Universitatea „Politehnica” Bucureşti
Facultatea de Electronică, Telecomunicaţii şi Tehnologia Informaţiei

Sinteza vorbirii pornind de la mişcarea buzelor

Proiect de diplomă

prezentat ca cerinţă parţială pentru obţinerea titlului de
Inginer în domeniul *Electronică, Telecomunicaţii şi Tehnologia Informaţiei*,
programul de studii de licenţă *Reţele şi software pentru Telecomunicaţii*

Conducători ştiinţifici
Dr. Ing. Dan ONEAȚĂ
Conf. Dr. Ing. Horia CUCU

Absolvent
Octavian PASCU

Universitatea "Politehnica" din Bucureşti
Facultatea de Electronică, Telecomunicaţii şi Tehnologia Informaţiei
Departamentul **Tc**

Anexa 1

TEMA PROIECTULUI DE DIPLOMĂ
a studentului **PASCU M. Octavian , 442D-RST.**

1. Titlul temei: Sinteza vorbirii pornind de la mișcarea buzelor

2. Descrierea contribuţiei originale a studentului (în afara părţii de documentare) și specificații de proiectare:
Scopul proiectului de sintetizare a vocii pe baza mișcării buzelor este de a oferi persoanelor ce nu mai pot vorbi din cauza unui accident posibilitatea de a comunica cu persoanele din jur folosind mișcarea buzelor.
Pentru realizarea acestei comunicări, o camera va efectua captura mișcării buzelor iar cu ajutorul unui program ce va conține un model antrenat de inteligență artificială, va fi sintetizata o voce ce va reproduce cuvintele mimate de utilizator.
Din punctul de vedere al structurii modelului, acesta va fi construit în limbajul de programare Python, cu ajutorul librăriei Pytorch pentru inteligenta artificiala, folosind tehnici de Deep Learning.
În continuarea creării modelului, acesta va fi antrenat pe un set de date în limba engleza și se va efectua optimizarea hiperparametrilor.

3. Resurse folosite la dezvoltarea proiectului:
Limbaj de programare Python, librăria Pytorch

4. Proiectul se bazează pe cunoştinţe dobîndite în principal la următoarele 3-4 discipline:
1. Programarea calculatoarelor 2. Structuri de date și algoritmi 3. Programarea orientata pe obiect

5. Proprietatea intelectuală asupra proiectului aparţine: U.P.B.

6. Data înregistrării temei: 2019-12-04 08:54:12

Conducător(i) lucrare,
Conf. dr. ing. Horia CUCU

Student,

Director departament,
Conf. dr. ing. Eduard POPOVICI

Decan,
Prof. dr. ing. Mihnea UDREA

Cod Validare: **1c27d5d981**

Anexa 5

Declarație de onestitate academică

Prin prezenta declar că lucrarea cu titlul „Sinteza vorbirii pornind de la mișcarea buzelor", Prezentată în cadrul Facultății de Electronică, Telecomunicații și tehnologia Informației a Universității „Politehnică" din București ca cerință parțială pentru obținerea titlului de *Inginer* în domeniul *Electronică, Telecomunicații și Tehnologia Informației*, programul de studii *Rețele și Software pentru Telecomunicații* este scrisă de mine și nu a mai fost prezentată niciodată la o facultate sau instituție de învățământ superior din țară sau străinătate.

Declar că toate sursele utilizate, inclusiv cele de pe Internet, sunt indicate în lucrare, ca referințe bibliografice. Fragmentele de text din alte surse, reproduse exact, chiar și în traducere proprie din altă limba, sunt scrise între ghilimele și fac referință la sursă. Reformularea în cuvinte proprii a textelor scrise de către alți autori face referință la sursă. Înțeleg că plagiatul constituie infracțiune și se sancționează conform legilor în vigoare.

Declar că toate rezultatele simulărilor, experimentelor și măsurătorilor pe care le prezint ca fiind făcute de mine, precum și metodele prin care au fost obținute, sunt reale și provin din respectivele simulări, experimente și măsurători. Înțeleg că falsificarea datelor și rezultatelor constituie fraudă și se sancționează conform regulamentelor în vigoare.

București, *25.06.2020*

Absolvent *Octavian PASCU*

(semnătură în original)

Cuprins

Lista figurilor

Lista tabelelor

Lista acronimelor

BPTT - Backpropagation through time

CPU – Central processing unit

ELU - Exponenţial linear unit

GRU - Gated recurrent unit

GPU – Graphical processing unit

IA - Inteligenţă artificială

LSTM - Long short-term memory

MCD – Mel cepstral distortion

MLP - Multilayer perceptron

MSE - Mean squared error

RGB - Red blue green

ReLU - Rectified linear unit

RNR - Reţea neurală recurentă

TTS - Text-to-speech

Introducere

Motivaţie

Comunicarea verbală este principalul mijloc de interacţiune dintre oameni şi reprezintă un mod de a transmite informaţii şi emoţii. Pentru persoanele ce şi-au pierdut abilitatea de a vorbi, comunicarea non-verbala nu poate oferi aceleaşi posibilităţi de a transmite un mesaj: limbajul prin semne nu este cunoscut de toată lumea iar prin scris este anevoios de comunicat faţă în faţă cu alte persoane.

Accidente ce duc la imposibilitatea vorbirii sunt des întâlnite în ziua de azi dar soluţile la această problemă nu oferă aceeaşi experienţă pentru persoanele din jur. Ideea de redare a vorbirii pentru persoanele cu deficienţe nu este una nouă, există aplicaţii care oferă acest lucru, de exemplu text-to-speech(TTS). Deşi aceste aplicaţii oferă posibilitatea comunicării verbale ele presupun existenţa unui mediu adiţional până la comunicare, în cazul TTS trebuiesc scrise cuvintele.

Pentru eliminarea acestui mediu o soluţie ar fi transmiterea directă a vorbirii prin captarea mişcărilor fetei. Acest lucru presupune folosirea unui aparat pentru înregistrarea fetei şi puterea de procesare necesară antrenării unei reţele neurale artificiale.

Studiul inteligenţei artificiale(IA) este un domeniu în dezvoltare iar popularitatea şi numărul de aplicaţii continuă să crească odată cu avansul tehnologic computational. Domeniile în care IA este folosită includ medicină, cu utilizări precum diagnosticarea unei boli dar şi tratarea acesteia.

Scopul lucrării este antrenarea şi folosirea unei reţele neurale artificiale pentru sinteza vocii bazate pe mişcarea buzelor. Pentru realizarea acestui scop şi a obţine o performanţă cât mai bună am utilizat 3 arhitecturi diferite ale reţelei şi diverse metode de preprocesare a datelor precum decuparea imaginilor şi folosirea derivatelor de ordinul 1 şi 2. Baza de date folosită pentru antrenare este "GRID corpus", limba vorbirii fiind engleză.

Structura lucrării

Capitolul 1 cuprinde noţiuni teoretice fundamentale din cadrul reţelelor neurale artificiale folosite cât şi descrierea metodelor de evaluare şi procesare a datelor.

Capitolul 2 prezintă tehnologiile folosite şi aplicabilitatea lor pentru realizarea scopului lucrării. Sunt explicate funcţiile folosite în cadrul lucrării şi în ce mod.

Capitolul 3 conţine cele 3 arhitecturi folosite pentru antrenarea reţelei: MLP, convolutional şi autoregresiv, şi sunt prezentate avantajele si dezavantajele lor.

Capitolul 4 descrie metodele de preprocesare a datelor pentru a fi folosite ca şi date de intrare în model.

Capitolul 5 se axează pe implementarea software a arhitecturilor, conţinând bucăţi de cod relevante în preprocesarea, antrenarea şi evaluarea lor.

Capitolul 6 cuprinde rezultatele obţinute şi diferenţele practice dintre arhitecturi şi metode de preprocesare a datelor.

Capitolul 1
Noţiuni teoretice

1.1Machine Learning

Machine Learning este studiul algoritmilor care se îmbunătăţesc automat prin experienţa. Algoritmii de Machine Learning construiesc un model matematic bazat pe un set de date şi sunt folosiţi pentru în număr ridicat de aplicaţii precum filtrarea de email, detecţia de feţe sau recunoaşterea automată a vorbirii.

Aceşti algoritmi sunt împărţiţi în 3 tipuri:

- Învăţarea supervizată se ocupă de algoritmi în care calculatorul primeşte un set de perechi intrare-iesire, la care trebuie să ajungă, prin antrenare.
- Învăţarea nesupervizată în care nu este specificat un rezultat corespunzător setului de date şi are ca scop aflarea structurii setului de date
- Reinforment learning prezintă algoritmi ce au ca scop interacţiunea cu un mediu înconjurător dinamic, urmărind un anumit rezultat.

Machine Learning şi Inteligenţa Artificială sunt domenii apropiate ca şi scop. La momentul actual, multe surse afirmă faptul că Machine Learning este un subdomeniu al Inteligenţei Artificiale.

Pentru a utiliza Machine Learning trebuie creat un model şi apoi antrenat pe un set de date pentru a face predicţii. Există mai multe tipuri de modele precum:

- Reţele neurale artificiale
- Reţele Bayesiene
- Algoritmi genetici

În continuare voi descrie algoritmi antrenaţi supervizat, de tip reţele neurale artificiale, folosiţi pentru performanţe bune.

1.2Reţele neurale artificiale

Reţele neurale artificiale sunt modele inspirate de reţelele neuronale biologice care constituie creierul. Modelele de acest timp sunt formate din neuroni artificiali care îşi transmit date între ei iar fiecare neuron transmite mai departe rezultatul unei funcţii liniare. Scopul acestor reţele este de a imita creierul uman în rezolvarea unei cerinţe predeterminate.
O reţea neurală este formată dintr-un număr de neuroni conectaţi între ei. Fiecare conexiune are o pondere asociată. Ponderile sunt principalul mod de stocare al informaţiei iar antrenarea se face prin schimbarea lor. Există neuroni ce primesc date de intrare, neuroni ascunşi ce prelucrează în continuare datele şi neuroni ce transmit mai departe predicţiile făcute de reţea.

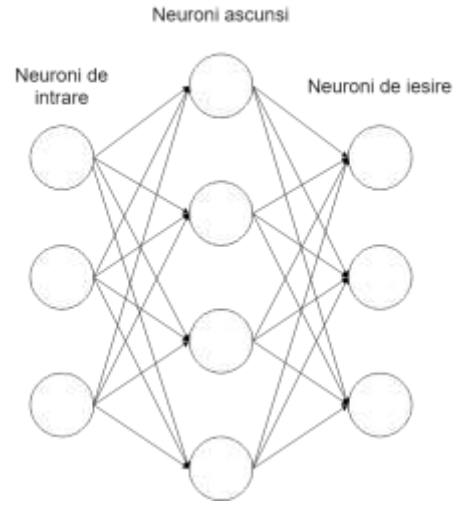


Figura 1.1: Reţea neurală artificială

Fiecare neuron are un set de date la intrare, un set de date la ieşire şi o metodă de calcul a următorului nivel de activare în timp. Neuronul face calculele local folosind datele de la intrare şi nu necesită un control global.

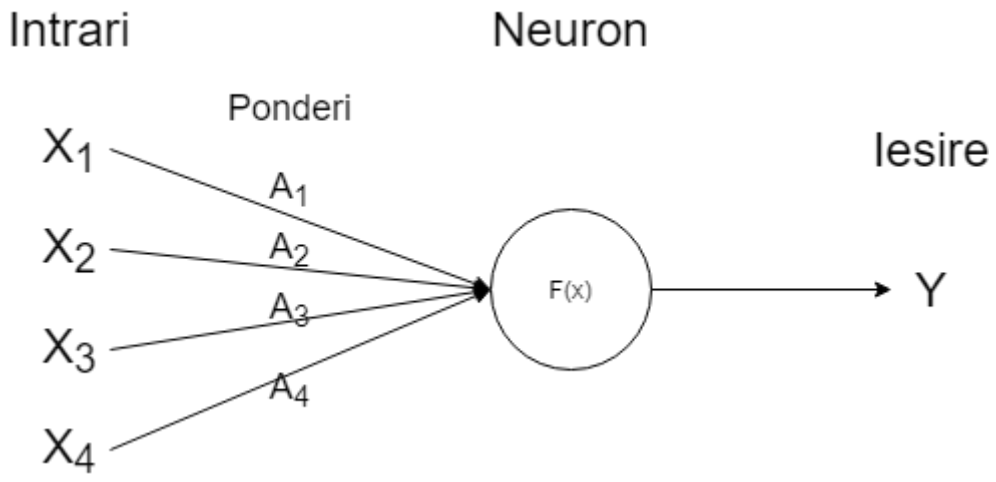


Figura 1.2: Structura unui neuron artificial

Pentru crearea unei reţele neurale în scopul îndeplinirii unei cerinţe trebuie stabilit numărul de neuroni, metodă de conexiune a neuronilor, trebuiesc iniţializate ponderile şi antrenarea lor folosind un algoritm pe un set de date.

Ca şi structură a reţelei neurale există mai multe tipuri:

- Reţea "feed-forward" , un tip de reţea în care conexiunile sunt unidirecţionale şi nu există cicluri
- Reţea recurentă , un tip de reţea în care conexiunile pot forma cicluri

Pentru ajustarea ponderilor şi scăderea erorii în antrenarea unui model se foloseşte un algoritm numit "Backpropagation". Backpropagation e o tehnică eficientă de calcul a gradienţilor unei reţele arbitrare

şi este folosit împreună cu algoritmi de optimizare (precum gradient descent) pentru minimizarea erorii.

La antrenarea unui model setul de date este împărţit în 3 părţi:

- Set de date pentru antrenare, în general între 80-90% din date
- Set de date pentru validare, pe acesta se calculează eroarea în timpul antrenării
- Set de date pentru testare utilizat pentru observarea performanţei modelului

În funcţie de eroarea pe fiecare din seturi se poate observa dacă modelul este construit adecvat pentru setul de date.

Un exemplu ar fi "overfitting", un fenomen ce apare în cazul în care modelul este prea complex pentru setul de date. În acest caz eroarea pe setul de antrenare scade către 0 dar cea de validare scade iar apoi creşte semnificativ faţă de eroarea de pe setul de antrenare.

Pentru a controla complexitatea modelului, se pot schimba "hyper-parametrii" ce specifică arhitectura reţelei, de exemplu numărul de celule dintr-un strat, numărul de straturi, funcţii de activare.

Reţelele neurale adânci sunt reţele neurale în care există mai multe straturi ascunse. În această lucrare sunt folosite următoarele structuri ale reţelelor neurale adânci:

- Reţele neurale convoluţionale
- Reţele neurale recurente

1.2.1 Reţele neurale convoluţionale

Reţelele neurale convoluţionale îşi iau numele de la operaţia de convoluţie din matematică şi au ca scop principal encodarea invarianţei la translaţii şi informaţia locală. Ca şi consecinţă numărul de parametrii folosiţi este redus. În cazul reţelelor neurale tradiţionale, numărul de parametrii necesari sunt direct proporţionali cu numărul de date la intrare. Cum fiecare pixel dintr-o imagine rezprezinta un număr (în cazul imaginilor alb-negru) sau 3 numere (în cazul imaginilor RGB), numărul de date la intrare este foarte ridicat, de exemplu pentru o imagine color cu rezoluţia 1080p vom avea 1920*1080*3 parametrii la intrare.

Formula convoluţiei:

$$(f * g)(t) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau)\, g(t-\tau)\, d\tau$$

Metoda prin care reţelele neurale convoluţionale reduc acest număr este de a folosi filtre de diferite dimensiuni pentru extragerea de parametrii din regiuni locale în loc de toată imaginea de la intrare.

Pentru exemplificarea diferenţei dintre numărul de parametrii, fie o imagine RGB de dimensiunea 32x32:

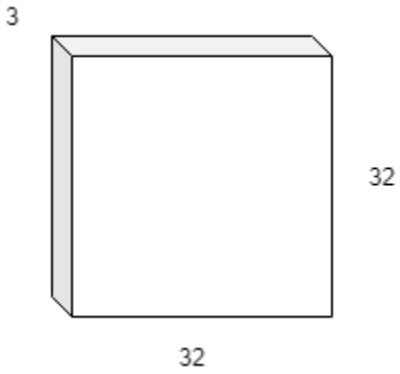


Figura 1.3: Dimensiunea imaginii RGB

- Pentru un tip de strat total-conectat ("fully-connected"), toate datele din imagine sunt conectate cu fiecare neuron din stratul următor din reţea. Asta înseamnă că fiecare neuron va avea 32*32*3 conexiuni şi deci 32*32*3 ponderi. La conexiunea cu următorul strat de 32*32 vom avea 32*32*3*32*32= 3,145,728 parametrii doar într-un singur strat.

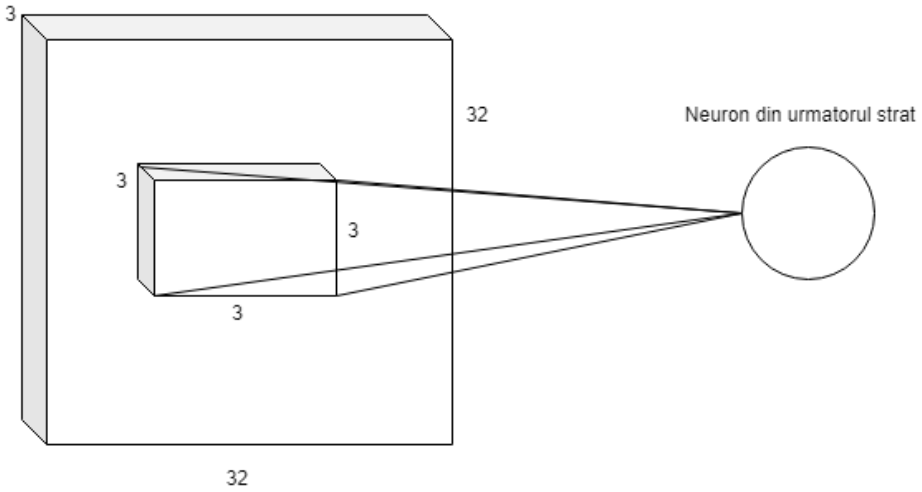


Figura 1.4: Operaţia de convoluţie din cadrul reţelei neurale convoluţionale

- În cazul unei reţele neurale convoluţionale, ne putem uita la o regiunea locală a imaginii şi să aplicăm operaţia de convoluţie pe acea regiune. Putem observa în figura că în cazul unui filtru de 3x3x3 vom avea 27 parametrii captaţi de către un neuron, având în total 27*32*32=27,648 parametrii, mult mai puţini comparativ cu metoda precedentă. O modalitate de a scădea şi mai mult numărul de parametrii este de a folosi aceleaşi ponderi pentru toată imaginea, filtrul având valori constante. Cu această metodă putem folosi doar 3*3*3 = 27 parametrii pentru conectarea unei imagini 32x32x3 la un strat 32x32.

Pe lângă numărul scăzut de parametrii (de la 3,145,728 la 27*k, k=numărul de filtre), folosind filtre speciale pentru toată imaginea putem extrage caracteristici oriunde s-ar afla în imagine.

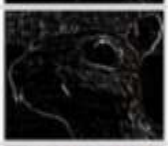
Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Figura 1.5: Exemple de filtre de convoluţie [1]

Folosirea filtrelor are multe avantaje dar există şi dezavantaje:

- Putem pierde informaţia de la marginile imaginii
- Multe filtre se suprapun şi captează aceleaşi caracteristici

Pentru rezolvarea acestor dezavantaje avem următoarele tehnici:

- Folosirea padding-ului (adăugarea de zerouri) la marginea imaginilor astfel încât filtrele să nu piardă informaţie dar acest lucru creşte dimensiunea ieşirii
- Introducerea noţiunii de "stride" ce reprezintă numărul de pixeli depărtare faţă de pixelul curent unde se va folosi următoarea convoluţie.

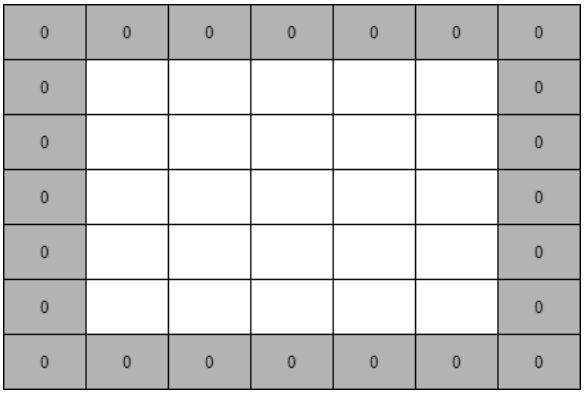


Figura 1.6: Padding în reţele neurale convoluţionale

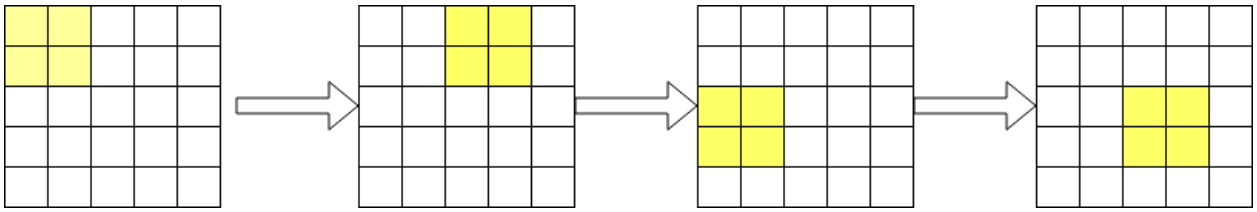


Figura 1.7: Stride=2 în reţele neurale convoluţionale

Folosind straturi de convoluţie dintr-o reţea neurală, modelul poate învăţa doar funcţii liniare. Pentru ca modelul să poată genera ieşiri cu o complexitate mai mare, sunt introduse straturi non-liniare numite şi funcţii de activare, precum ReLU, funcţia sigmoidală, funcţia tangentă hiperbolică.

Funcţia sigmoidală este o funcţie ce produce valori între 0 şi 1. Această funcţie introduce neliniaritati în reţea şi are proprietatea de a returna 1 pentru valori mai mari de o valoare arbitrară sau 0 pentru valori mai mici.

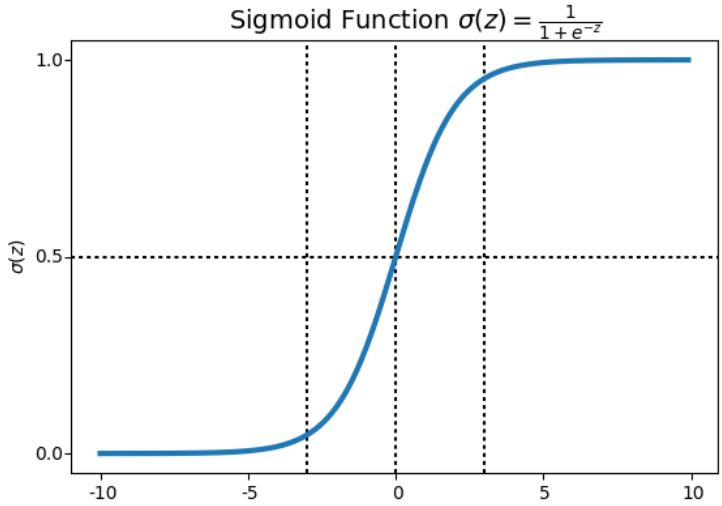


Figura 1.8: Funcţia Sigmoidala [2]

O altă funcţie neliniară folosită des este Tangenţă Hiperbolică. Această funcţie este asemănătoare cu funcţia sigmoidală dar returnează valori între [-1,1].

$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

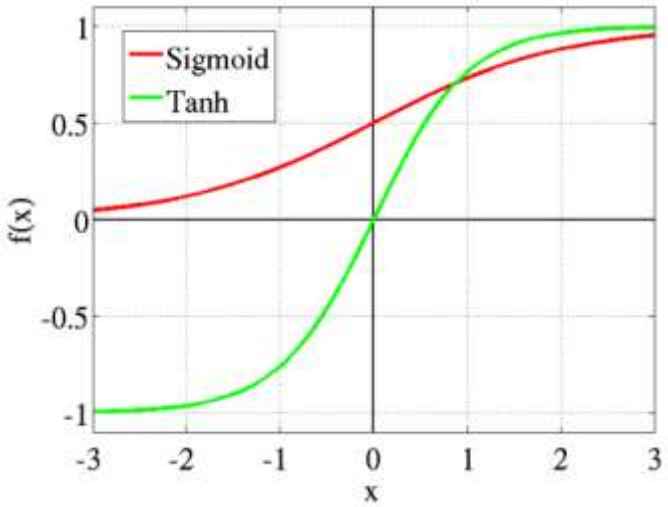


Figura 1.9: Funcţia Tangenţă Hiperbolică [3]

Rectified Linear Unit (ReLU) este cea mai folosită funcţie de activare cu ReLU(x) = max(0,x). Problema acestei este faptul că toate valorile mai mici de 0 devin 0 iar acest lucru scade abilitatea modelului de a învăţa din setul de date.

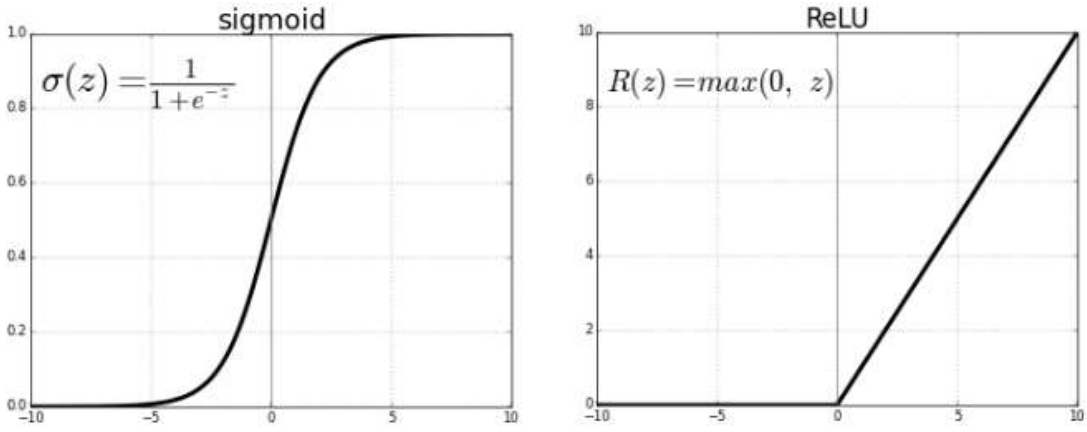


Figura 1.10: Comparaţie dintre funcţia sigmoidală şi ReLU [3]

Pentru a rezolva această problemă o soluţie este funcţia Exponenţial Linear Unit(ELU). Este o funcţie asemănătoare cu ReLU doar că poate returna valori negative .

$$R(z) = \begin{cases} z & z > 0 \\ \alpha.(e^z - 1) & z <= 0 \end{cases}$$

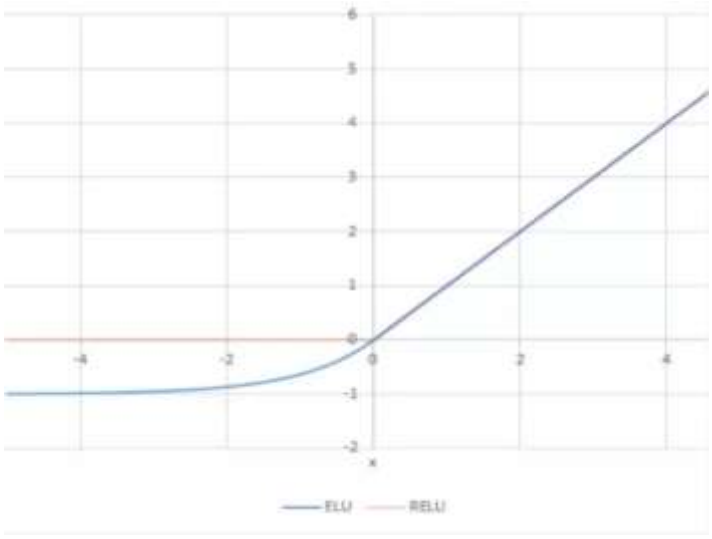


Figura 1.11: Comparaţie dintre funcţiile ELU şi ReLU [4]

Un alt strat folosit des în reţelele neurale convoluţionale este stratul de "Pooling". Scopul acestui strat este de a reduce dimensiunea intrării pentru următorul strat convoluţional prin folosirea unui "stride">1. Cel mai des folosit strat de Pooling este MaxPool, care alege valoarea maximă dintr-un filtru şi elimină celelalte valori.

Figura 1.12: Stratul MaxPooL [5]

Pentru combaterea fenomenului de overfitting, este folosit stratul Dropout. Acesta elimină un procentaj din intrări, în mod aleator la fiecare iteraţie din antrenare, şi reduce complexitatea reţelei. La testare sunt folosite toate conexiunile.

Toate aceste straturi pot fi folosite pentru intrări 1d , 2d sau 3d în funcţie de scopul reţelei.

1.2.2 Reţele neurale recurente

Reţelele neurale recurente sunt reţele neurale specializate pentru date secvenţiale. La fel cum reţelele neurale convoluţionale sunt folosite pentru a prelucra imagini largi, reţele neurale recurente au scopul de a procesa secvenţe foarte mari.

O caracteristică a RNR ce ajută la atingerea scopului este de a partaja parametrii, ce este foarte important când o porţiune din informaţie poate apărea în mai multe poziţii din aceeaşi secvenţă. Astfel, fiecare pas viitor din reţea este influenţat de o parte din parametrii folosiţi în paşii trecuţi.

În figură de mai jos este exemplificată o reţea neurală recurentă:

- X reprezintă input-ul
- O este output-ul
- L este funcţia de cost ce reprezintă diferenţa dintre ieşirea din reţea şi rezultatul dorit y
- Y este rezultatul dorit

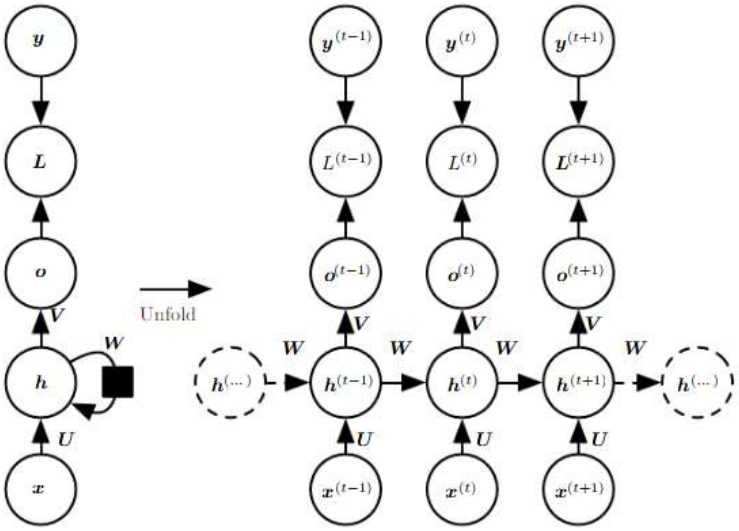


Figura 1.13: Structura reţelei neurale recurente [3]

Se poate observa faptul că odată desfăcută, reţeaua are conexiuni ascunse parametrizate de W ce influenţează rezultatul curent faţă de cel trecut. În această reţea secţiunea de input şi secţiunea de ouput sunt de aceeaşi lungime.

Propagarea în timp este descrisă de ecuaţia:

$$\boldsymbol{a}^{(t)} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)}$$

Funcţia h:

$$\boldsymbol{h}^{(t)} = \tanh(\boldsymbol{a}^{(t)})$$

Ieşirea:

$$\boldsymbol{o}^{(t)} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)}$$

Un exemplu în care lungimea intrării şi lungimea ieşirii sunt diferite:

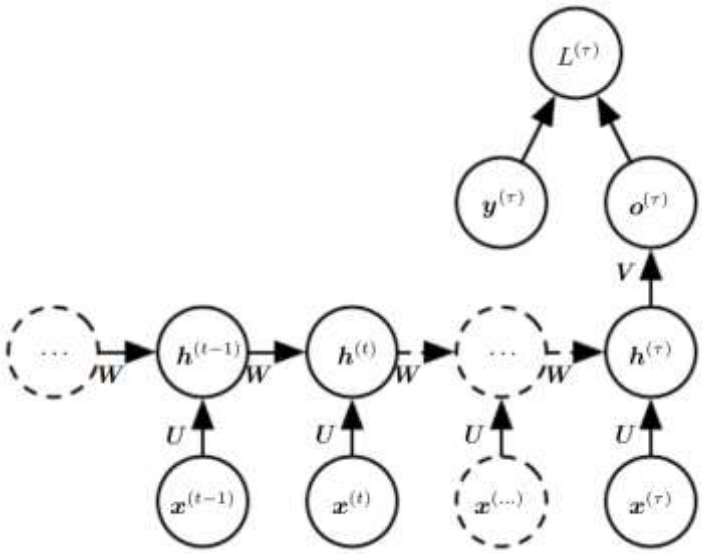


Figura 1.14: Reţea neurală recurentă cu o singură ieşire [6]

În cazul reţelei neurale convoluţionale, backpropagation este calculat că în cazul reţelelor neurale tradiţionale. Pentru reţelele neurale recurente , este compus "backpropagation through time"(BPTT). Această variantă de backpropagation desfăşoară toate conexiunile în timp, iar suma erorilor conexiunilor în timp este adăugată la eroarea totală.

Exemplele de până acum reprezintă reţele neurale recurente secvenţiale, iar output-ul este influenţat doar de momentele de timp trecute. Pentru unele aplicaţii, toată secvenţă poate fi folosită la calcularea output-urilor, şi implicit şi paşii din viitor.

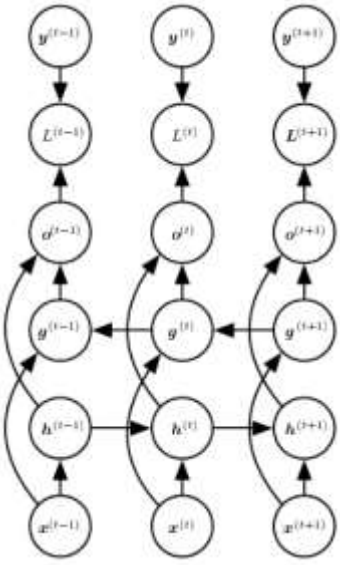


Figura 1.15: Reţea neurală recurentă bidirecţională [6]

În această figură recurentă h(t) propaga informaţia înspre viitor iar g(t) în trecut. Astfel ieşirea o(t) primeşte informaţii şi din trecut şi din viitor pentru fiecare pas.

O arhitectură populară pentru maparea unei secvenţe de intrare la o secvenţă de ieşire ce nu are aceeaşi lungime este arhitectura encoder-decoder şi este folosită în multe aplicaţii precum recunoaşterea vocii, traducere etc.

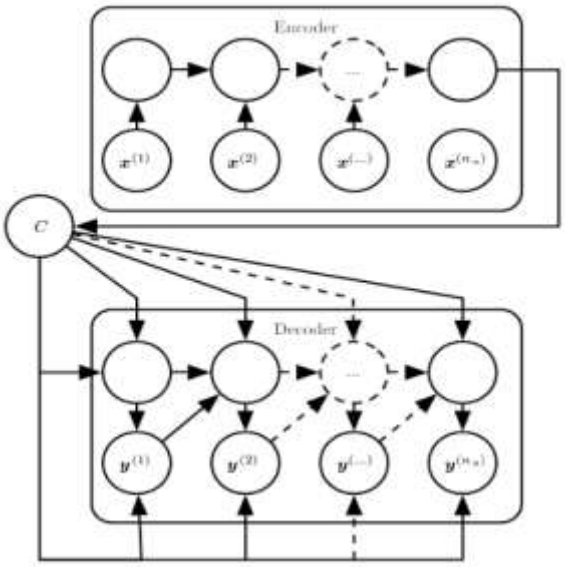


Figura 1.16: Reţea neurală recurentă cu arhitectură encoder-decoder [6]

În figură de mai sus codorul produce o reprezentare a secvenţei de intrare, C, ce poate fi un vector sau o secvenţă de vectori. Această secvenţă este folosită ca şi intrare pentru decodor ce o procesează într-o secvenţă de dimensiune variabilă.

La fel ca la reţelele neurale convoluţionale, folosind un număr mare de straturi poate duce la fenomenul de dispariţie a gradienţilor (en. Vanishing gradient) iar soluţiile acestei probleme sunt:

- De a forma conexiuni dintre trecutul distant şi prezent
- De a elimina conexiuni

Cele mai efective modele secvenţiale sunt "long short-term memory"(LSTM) şi reţelele bazate pe "gated recurrent unit" (GRU).

Aceste modele sunt bazate pe idea de a crea conexiuni în timp ce reţin derivate care nu dispar.

O celulă dintr-o reţea neurală LSTM are structură:

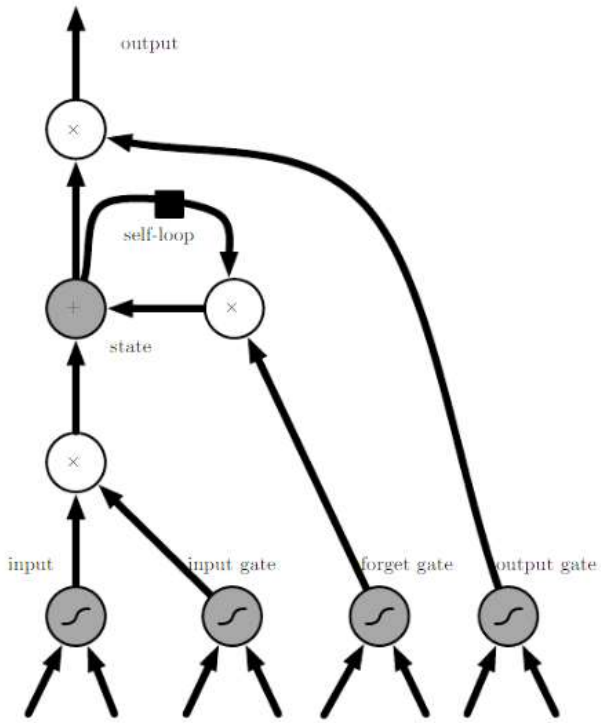


Figura 1.17: Structura LSTM [6]

Celulele sunt conectate cu ele însăşi, înlocuind parametrii ascunşi din reţelele neurale recurente tradiţionale, şi astfel informaţia nu dispare în timp. La intrare sunt folosiţi neuroni artificiali tradiţionali. Celulele cu recurentă (en. Self-loop) sunt controlate de către un parametru numit poartă de uitare (en. Forget-gate) ce setează ponderea ca fiind o valoare între 0 şi 1 prin folosirea unei funcţii sigmoidale.

Ca şi alternativă a reţelelor neurale LSTM sunt modelele bazate pe GRU. Diferenţa dintre cele două arhitecturi este faptul că o singură unitate de poartă (en. Gate unit) controlează factorul de uitare (en. Forgetting factor) şi updatează statusul celulei.

1.3 **Funcţia de cost**

Pentru că o reţea neurală artificială să formeze o predicţie bună este necesară o funcţie de cost ce minimizează eroarea prin conceptul de "gradient descent".

În cadrul lucrării sunt folosite două funcţii de cost, MSE în primele două arhitecturi şi FlowtronLoss în a treia:

- Mean Squared Error(MSE) este o funcţie des întâlnită . Fie "C" funcţia de cost, "N" numărul de exemple pentru antrenare, "y" un vector cu rezultatele adevărate şi "o" un vector cu predicţii date de reţea. În acest caz formula acestei funcţii este:

$$C(\mathbf{y}, \mathbf{o}) = \frac{1}{N} \sum_{i=1}^{N} (y_i - o_i)^2$$

- FlowtronLoss este o funcţie de cost folosită în cadrul arhitecturii cu decodor autoregressiv. MSE nu poate fi folosită în acest caz deoarece modul de funcţionare al reţelelor neurale autoregresive este diferit faţă de cele convoluţionale sau recurente.

1.4Algoritmi de optimizare

Un algoritm de optimizare este folosit pentru minimizarea costului f(x), unde x ∈ R. Gradientul este Δf(x) iar dimensiunea pasului k =t_k .

1.4.1 Batch Gradient Decent

Acest algoritm actualizează parametrii x după trecerea prin tot setul de antrenare:

$$x_{k+1} = x_k - t_k \Delta f(x_k)^{(1:n)}$$

Optimizatorul converge garantat către minimul global pentru o problemă convexă şi spre un minim local pentru probleme non-convexe. În cazul reţelelor neurale adânci, acest calcul ar dura prea mult. De asemenea, memoria computaţională este limitată şi de aceea este dificil să folosim tot setul de date deodată.

1.4.2 Stohastic Gradient Decent

Algoritmul calculează gradientul şi updatează parametrii pentru fiecare probă.

$$x_{k+1} = x_k - t_k \Delta f(x_k)^{(i)}$$

Actualizarea parametrilor cauzează funcţia să fluctueze deoarece există o variantă mare între diferitele date pentru antrenare şi deşi putem folosi un pas mic pentru convergenţă sigură acest lucru ar îngreuna antrenarea semnificativ.

1.4.3 Adam

Adam este un algoritm de optimizare bazat pe Stohastic Gradient Decent dar este mai eficient, utilizează mai puţină memorie şi poate fi folosit într-un număr mare de aplicaţii. m_k si v_k sunt media şi varianta necentrata.

$$m_k = \beta_1 m_{k-1} + (1-\beta_1)\Delta f(x_k)$$
$$v_k = \beta_2 v_{k-1} + (1-\beta_2)\Delta f(x_k)^2$$
$$\hat{m}_k = \frac{m_k}{1-\beta_1^k}, \hat{v}_k = \frac{v_k}{1-\beta_2^k}$$
$$x_{k+1} = x_k - \frac{t}{\sqrt{\hat{v}_k} + \varepsilon}\hat{m}_k$$

În general $\beta_1 = 0.9$ si $\beta_2 = 0.999$ iar $\varepsilon = 10 \cdot e^{-8}$.

Capitolul 2

Tehnologii folosite

2.1Limbajul de programare Python

Python este un limbaj de programare dinamic, de nivel înalt, creat de Guido van Rossum în anul 1991. Acesta oferă funcţionalităţi precum programare orientată pe obiecte şi programarea structurată. Scopurile acestui limbaj de programare sunt:

- De a oferi o flexibilitate crescută în scrierea aplicaţiilor
- De a fi uşor de înţeles şi a avea sintaxa simplificată

Din punct de vedere al sintaxei şi a semanticii, Python este uşor de înţeles. Formatarea este simplă şi sunt folosite cuvinte în engleză pentru comenzi.

Un aspect unic al limbajului de programare Python este folosirea identării pentru delimitarea blocurilor de cod în loc de acolade.

Câteva dintre comenzile ce pot fi folosite sunt:

- "if" pentru execuţia condiţională a unui bloc de cod
- "for" pentru a itera peste un obiect iterabil
- "try" pentru tratarea excepţiilor
- "continue" pentru trecerea la următoarea iteraţie
- "import" pentru adăugarea unor module sau librării
- "print" pentru a afişa

Majoritatea expresiilor din Python sunt similare cu cele din C sau Java, cu anumite diferenţe:

- Python foloseşte cuvintele "and", "or", "not" pentru operatorii booleani în loc de "&&", "||" şi "!".
- Python face diferenţa dintre liste şi tupluri. Listele sunt scrise precum [1,2,3] şi sunt mutabile, iar tuplurile sunt scrise precum (1,2,3) şi sunt imutabile.
- Python poate folosi indexi pentru iterarea peste liste, precum a[start:stop], a[start:stop:step].

Operatorii matematici (+,-,*,/) sunt folosiţi identic cu cei din alte limbaje de programare.

Unul dintre avantajele folosirii limbajului Python este numărul crescut de librării (peste 200.000) ce includ funcţii pentru multe domenii precum : "Machine learning", "Networking", "Multimedia", "Graphical user interface", "Dată analytics", "Databases" etc. .

2.2Librăria PyTorch pentru Machine Learning

PyTorch este o librărie open-source pentru Machine Learning bazată pe librăria Torch şi este folosită pentru aplicaţii precum "Computer vision" şi "Natural Language Processing", creată de "Facebook AI Research Lab".

Pytorch oferă 2 caracteristici:

- Tehnici de calcul bazate pe tensori, folosindu-se de unitatea grafica de procesare (GPU)
- Reţele neurale adânci bazate pe un sistem de calcul cu diferenţiere automată

Câteva dintre funcţiile folosite în cadrul acestei lucrări sunt:

- Torch.nn.Conv3d - aplică o convoluţie 3D asupra unei intrări
- Torch.nn.Conv1d - aplică o convoluţie 1D asupra unei intrări
- Torch.nn.BatchNorm1d - aplică normalizarea loturilor pe o intrare 2D sau 3D
- Torch.nn.BatchNorm3d - aplică normalizarea loturilor pe o intrare 5D
- Torch.nn.Linear - aplică o transformare liniară asupra unei intrări
- Torch.nn.Sigmoid - aplică funcţia Sigmoida asupra unei intrări
- Torch.nn.Dropout - transformă date la întâmplare în zerouri, folosit pentru regularizarea şi reducerea overfitting-ului

2.3Librăria Ignite

Librăria Ignite este o librărie de nivel înalt folosită pentru antrenarea reţelelor neurale în Pytorch. Ignite ajută la scrierea rapidă a codului pentru antrenarea unui model.

Câteva funcţionalităţi oferite sunt:

- Controlul metricilor
- Posibilitatea de oprire în funcţie de o condiţie arbitrară
- Salvarea automată a unui model

Funcţii folosite din cadrul librăriei Ignite sunt:

- Ignite.engine.create_supervised_model - Primeşte ca şi intrări modelul, optimizatorul, loss-ul şi returnează rezultatul antrenării folosind intrările
- Ignite.engine.create_supervisel_evaluator - Creează un evaluator al unui model
- Ignite.engine.Events - ajuta la controlul supervizării antrenării şi rezultatelor
- Ignite.handlers.ModelCheckpoint - salvează modelul curent în timpul antrenării

2.4Librăria Open-CV

Open-cv este o librărie ce conţine algoritmi pentru procesarea imaginilor.

Funcţii folosite din cadrul librăriei sunt:

- Cv2.VideoCapture - transformă un video într-un vector 4D (3,D,W,H)
- Cv2.cvtColor - setează culoarea unui input
- Cv2.COLOR-BGR2GRAY - transformă un input color în alb-negru

2.5Librăria Librosa

Librăria Librosa este o librărie folosită pentru procesare audio. În cadrul lucrării este folosită pentru transformarea semnalului audio în melspectrograme şi invers.

Câteva funcţii folosite:

- Librosa.feature.melspectrogram
- Librosa.power_to_db
- Librosa.feature.inverse.mel_to_audio

2.6Librăria Numpy

Librăria Numpy este o librărie utilizată pentru optimizarea operaţiilor cu matrici de dimensiuni mari

Câteva funcţii folosite:

- Np.pad - adaugă zerouri unui input
- Np.diff - compune diferenţiala unui input
- Np.empty - creează un vector format din zerouri

2.7CUDA

CUDA este o platformă pentru calculul paralel dezvoltată de NVIDIA. Fosind CUDA, viteza calculelor creşte dramatic în aplicaţii ce necesită multe calcule. Pentru utilizare este necesar un GPU dezvoltat de NVIDIA.

În inteligenţă artificială, numărul de parametrii poate fi la ordinul miliardelor ce trebuiesc ajustaţi prin back-propagation. Pentru reducerea timpului de rulare, paralelismul oferit de CUDA oferă un avantaj semnificativ faţă de CPU. De asemenea, deoarece importanţa reţelelor neurale a crescut în industrie, NVIDIA a format o librărie numită cuDNN ce creşte performanţa reţelelor.

În această lucrare librăria cuDNN este folosită la antrenarea reţelelor în Python.

Capitolul 3

Procesarea datelor

3.1 Video

Datele video folosite pentru antrenarea modelelor fac parte din setul de date "GRID corpus". Fiecare video conţine 75 cadre în 3 secunde, rezoluţia după preprocesare este 64x64 iar audio este capturat la 25khz. În total pentru fiecare persoană sunt folosite 900 video-uri de antrenare, 50 de validare şi 50 de testare.

Pentru procesarea datelor am folosit un model antrenat de detecţie a feţei, care extrage coordonatele caracteristicilor de la vorbitori. Rezultatele sunt sub forma text, având coordonatele punctelor conform imaginii:

Figura 3.1: Notaţii coordonatele feţei [7]

Am decupat video-urile originale folosind punctele [49,55,53,57] astfel încât conţin doar buzele plus 10 pixeli padding.

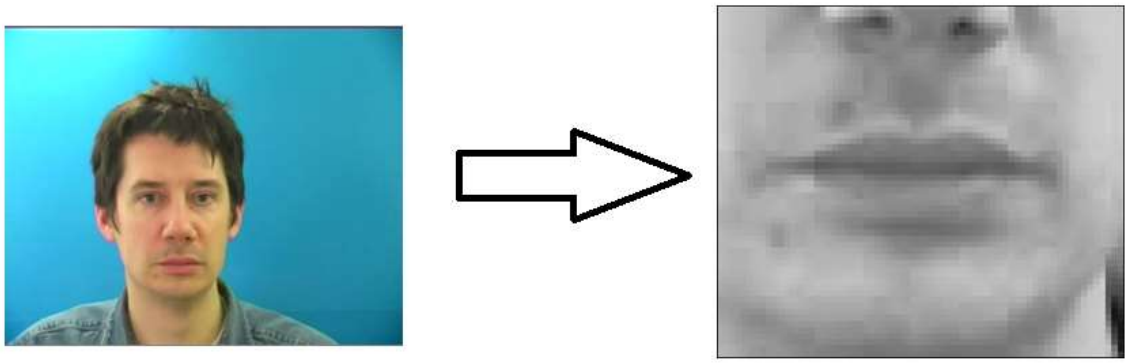


Figura 3.2: Preprocesarea fetei

Alte metode de procesare a datelor folosite sunt adăugarea de padding, calculul derivatelor de ordinul 1 şi 2 şi inserarea lor în setul de date, şi normalizarea valorilor inregistrarilor video şi derivatelor pentru a fi în intervalul [-1,1]. Toate aceste metode sunt explicate în capitolul 6, rezultate.

3.2 Audio

Spectrograma este o reprezentare vizuală a spectrului de frecvenţe al unui semnal ce variază în timp şi este compusă cu ajutorul transformatei Fourier de scurtă durată. Reprezentarea este o imagine ce pe orizontală are timpul, pe verticală are frecvenţele iar intensitatea culorilor reprezintă amplitudinea semnalului.

Oamenii nu percep schimbările de frecvente în mod liniar. Diferenţa dintre 500 Hz şi 1000 Hz este uşor de auzit dar nu putem observa diferenţa dintre 10000 Hz şi 10500 Hz. Pentru a lua în calcul acest lucru, în anul 1937 Stevens, Volkmann şi Newmann au propus o scară ce reprezintă diferenţele audio percepute de oameni, numită scara "mel".

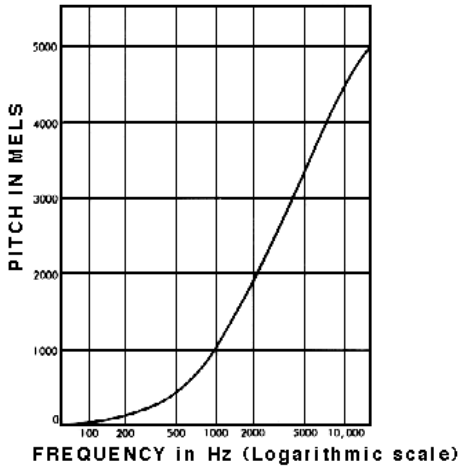


Figura 3.3: Scara mel [8]

Melspectrograma este o spectrogramă în care frecvenţele sunt convertite folosind scara mel.

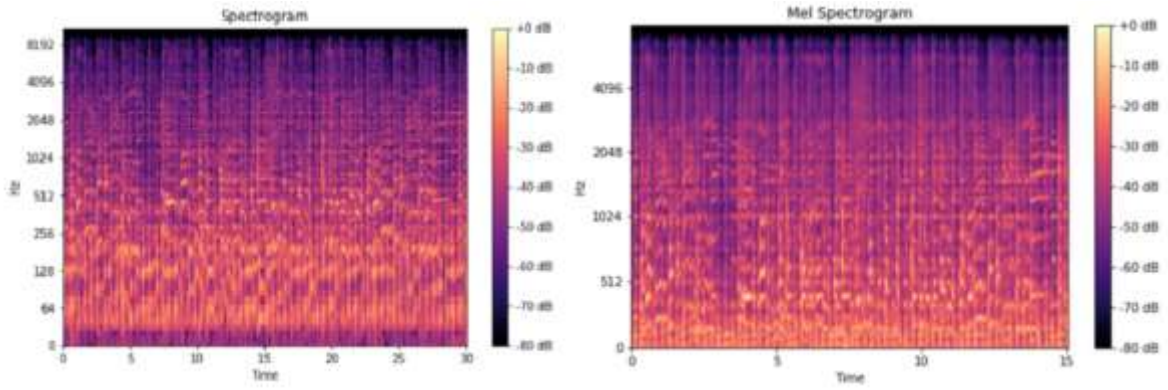


Figura 3.4: Comparaţie Spectrogram vs MelSpectrogram [8]

Audio din cadrul setului de date "GRID corpus" nu este aliniat temporal cu video, de aceea fişierele audio au fost extrase direct din video-urile sursă. Pentru transformarea din audio în melspectrograma s-a folosit o frecvenţă de eşantionare de 22050 Hz şi 80 de canale mel.

Metoda de transformare din spectograma în audio este bazată pe algoritmul Griffin-Lim şi este folosită cu ajutorul librăriei Librosa.

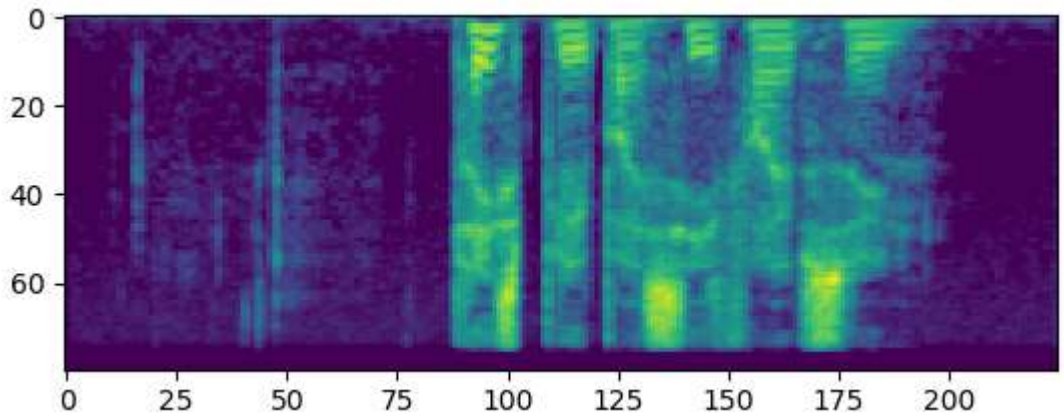


Figura 3.5: MelSpectrograma

Capitolul 4

Arhitecturi folosite

Arhitecturile folosite sunt de tipul encoder-decoder. Encoderul este folosit pentru transformarea contextului din video-urile de la intrare într-o secvenţă de vectori iar decoderul pentru procesarea secvenţei de vectori într-o matrice corespunzătoare mărimii spectrogramelor de la ieşire.

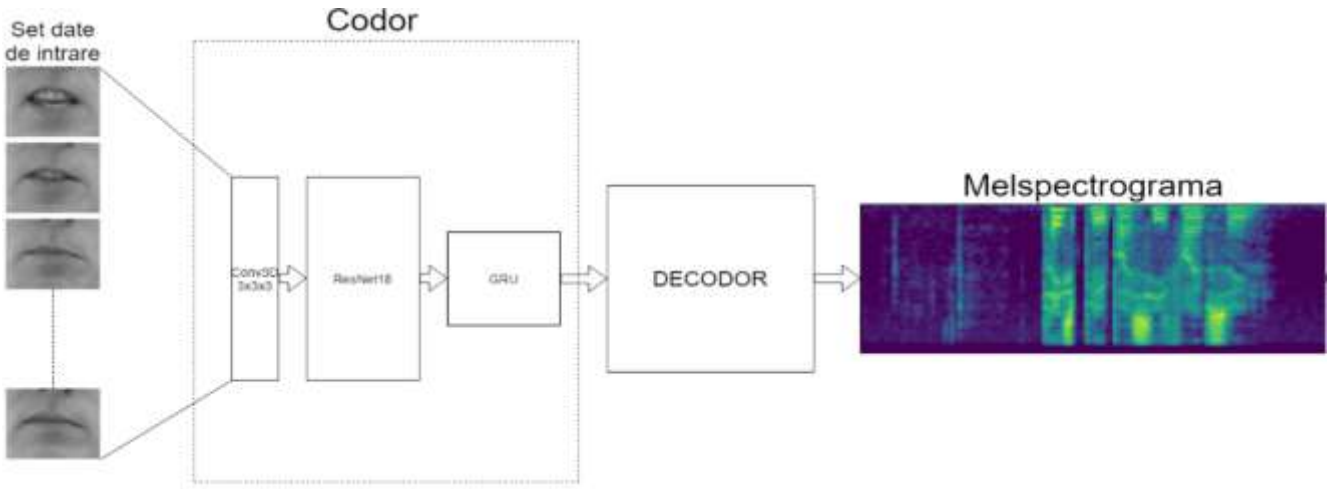


Figura 4.1: Arhitectură folosită

4.1Codor

Codorul este format dintr-un strat convoluţional 3D (1 canal la intrare şi 64 canale la ieşire), un strat BatchNorm3d, un strat ReLU , o versiune modificată a arhitecturii ResNet18 şi un strat GRU.

Resnet18 este o reţea neurală convolutionala ce conţine 18 straturi , folosită în aplicaţii pentru procesarea imaginilor. Numele acestei arhitecturi provine de la conexiunile reziduale folosite. O conexiune reziduală este o conexiune dintre straturi non-consecutive.

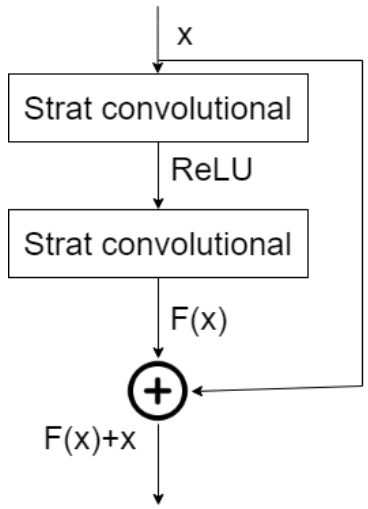


Figura 4.2: Conexiune reziduală

Pentru adaptarea în arhitectura de encoder am modificat primul strat într-un strat convolutional 2D cu 64 canale la intrare şi ieşire, şi am eliminat ultimul strat softmax.

Această arhitectură a decoderului este folosită în toate testele deoarece poate identifica toate caracteristicile datelor de la intrare, având o putere de procesare mare.

4.2Decoder

Ca şi decoder am folosit 3 arhitecturi, de tipul MLP, convoluţionale şi autoregresive.

4.2.1 Decoderul MLP

Termenul MLP(multilayer perceptron) descrie o reţea neurală artificială de tip "feed-forward" ce conţine straturi formate din perceptroni .

Arhitectura este formată din:

- 3 straturi liniare cu câte 2000 neuroni
- 2 straturi de normalizare a loturilor
- 2 straturi de Dropout
- 2 straturi ELU
- La ieşire se aplică funcţia sigmoidala

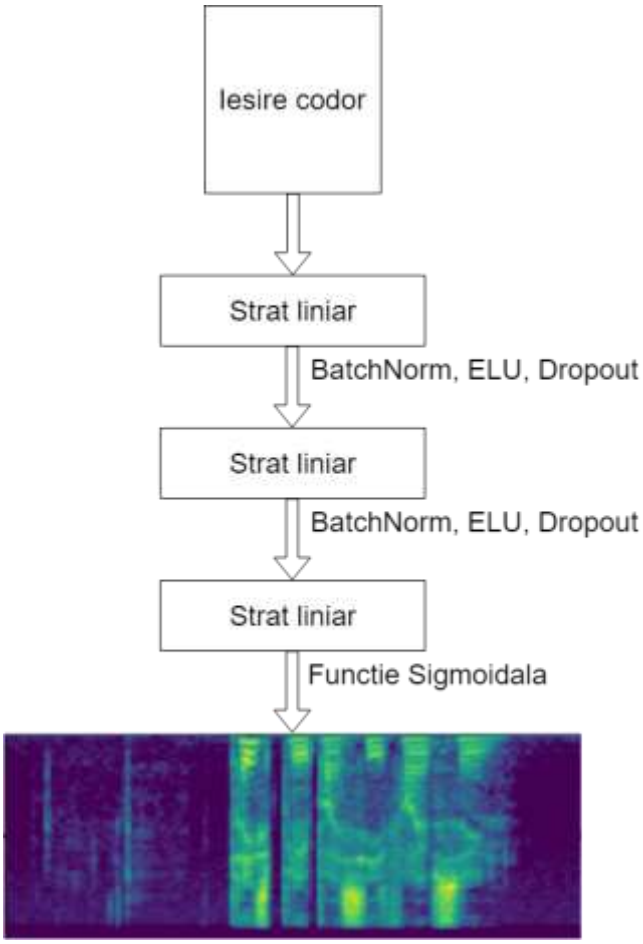


Figura 4.3: Decoder MLP

Avantajul acestei arhitecturi este uşurinţa înţelegerii şi implementării dar dezavantajul este ineficienţa şi performanţa scăzută.

4.2.2 Decoder convoluţional

Cea de-a doua arhitectură se foloseşte de reţele neurale convoluţionale şi este formată din:

- 1 strat de deconvoluţie pentru creşterea dimensiuni temporale. Înregistrările folosite pentru antrenare conţin 75 cadre. Pentru a ajuta la corelarea temporală dintre video-uri şi spectrograme am folosit un strat de deconvolutie cu un filtru de dimensiunea 3x3 şi stride = 3 şi astfel ajungem de la dimensiunea 75 la 225 a spectrogramelor.
- 2 straturi de convoluţie 1D, cu mărimea filtrului 3 şi folosind padding 1
- 3 straturi de normalizare a loturilor, 1 după fiecare convoluţie
- 3 straturi ELU, 1 după fiecare convoluţie

Această arhitectură are avantajul de a menţine dimensiunea temporală la folosirea convoluţiilor şi astfel învaţă mai eficient parametrii necesari.

Un alt pas necesar pentru menţinerea dimensiunii temporale a fost schimbarea frecvenţei de esantionare în 19300 Hz, astfel încât diemensiunea temporală a intrării 75 să poată fi înmulţită cu 3 şi să ajungă la dimensiunea temporală a spectrogramei 225.

Pentru un lot format din 8 video-uri:

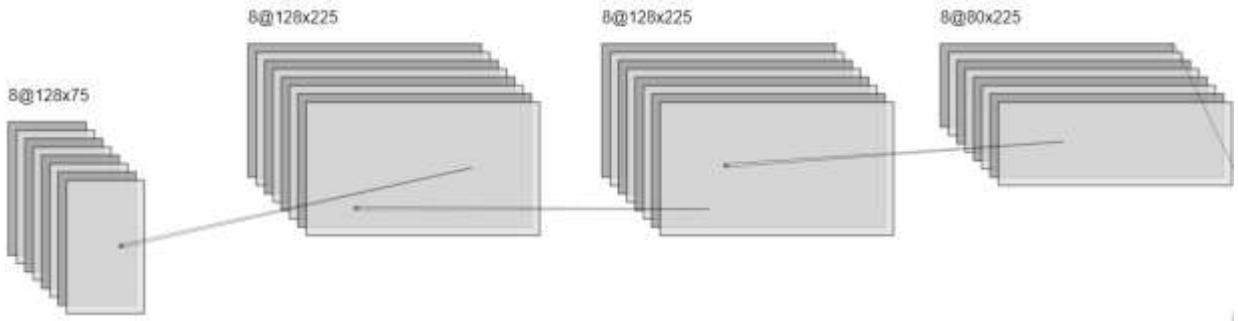


Figura 4.4: Decoder convolutional

4.2.3 Decodor autoregresiv

A treia arhitectură folosită este o variantă adaptată a decoderului folosit în cadrul Proiectului "Flowtron"[9]. Flowtron foloseşte o reţea neurală autoregresiva bazată pe fluxuri , de tip encoder-decoder pentru generarea spectrogramelor pornind din text.

O reţea neurală autoregresiva prezice cadrul curent folosind prezicerile anterioare.

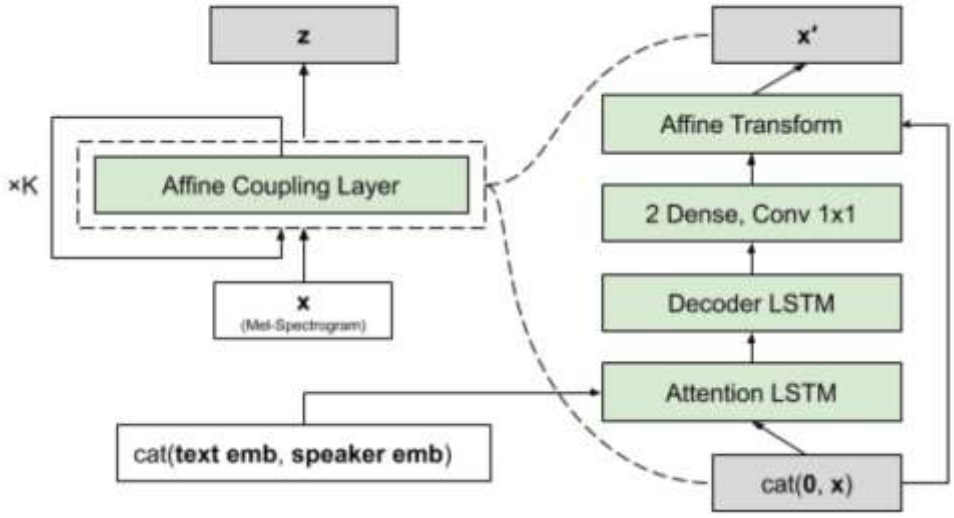


Figura 4.5: Arhitectură Flowtron [9]

În cazul arhitecturii reţelelor neurale autoregresive, ea are ca şi rol maparea unei variabile aleatoare ,cu o distribuţie gaussiana, în melspectrograma de la ieşire.

În antrenare, melspectrograma este dată la intrare iar ieşirea reprezintă o variabilă aleatoare cu distribuţia gaussiana. Generarea melspectrogramei rezultă din inversarea procesului.

Pentru adaptarea arhitecturii am eliminat partea de encoder, atenţia şi stratul de poartă şi am integrat codorul descris în capitolul 3.1 .

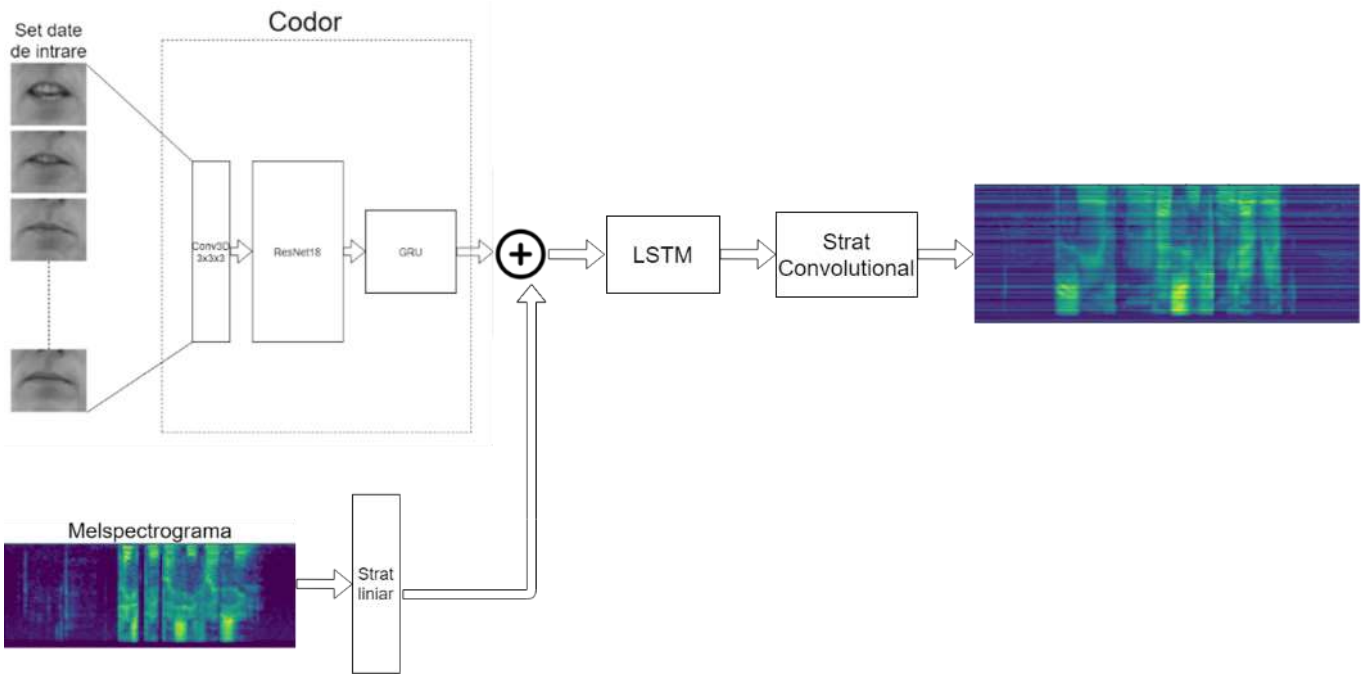


Figura 4.6: Adaptarea arhitecturii Flowtron

Capitolul 5

Implementarea software

Lucrarea conţine 5 fişiere scrise în totalitate de către mine (train.py, test.py, nn.py, dataset.py, Flowtron_infer.py) şi 3 fişiere ce conţin metode preluate din alte surse şi adaptate pentru a funcţiona în cadrul licenţei (Flowtron_train.py, Flowtron.py, audio.py)

5.1Train.py

Fişierul train.py conţine metoda de antrenare a unui model de inteligenţă artificială, la intrare primind un set de date şi returnând la ieşire modelul antrenat. Codul complet este scris în anexă A.

În metoda "main.py" întâi sunt prezentate argumentele ce pot fi folosite la rularea programului. "Mu" şi "sigma" reprezintă media şi abaterea medie standard a video-urilor şi derivatelor de ordinul 1 şi 2 pentru vorbitorii "s1", "s2", "s3", "s4", "s5", fiind calculate folosind fişierul "Compute_mean_std.py". Cele 2 variabile sunt folosite în cadrul metodei torchvision.transforms ce returnează o funcţie cu scopul normalizării datelor:

```
62      mu = [1.4024e+02, -4.9944e-03, 3.5836e-04]
63      sigma = [21.9047,  4.4827,  5.8206]
64      transform = torchvision.transforms.Normalize(mean=mu, std=sigma)
```

Figura 5.1: Funcţia de normalizare a datelor

Funcţia "transform" este dată ca şi intrare metodei "dataset.XTSDataset()" împreună cu numele fişierelor din setul de date şi locaţia lor.

```
77      train_dataset = src.dataset.xTSDataset(ROOT, "train", transform=transform)
78      valid_dataset = src.dataset.xTSDataset(ROOT, "valid", transform=transform)
```

Figura 5.2: Pregătirea datelor de antrenare

Metoda returnează conţinutul normalizat al fişierelor. În cazul datelor de antrenare şi validare, folosesc funcţia torch.util.dată.DataLoader() pentru organizarea datelor în loturi de câte 8 tupluri ce conţin imagini şi spectrogramele corespunzătoare.

```
80      train_loader = torch.utils.data.DataLoader(
81          train_dataset,
82          batch_size=8,
83          collate_fn=collate_fn,
84          shuffle=True)
```

Figura 5.3: Încărcarea datelor de antrenare

Pentru optimizarea parametrilor după fiecare iteraţie, am folosit funcţia "torch.optim.Adam" cu o rată de învăţare de 0.0001. Creearea funcţiilor de antrenare şi validare se face prin "engine.create_supervised_trainer" şi "engine.create_supervised_evaluator" cu intrările modelul, optimizatorul şi funcţia de cost. Apoi rularea se face prin comandă "trainer.run()" .

```
107     @trainer.on(engine.Events.ITERATION_COMPLETED)
108     def log_training_loss(trainer):
109         print("Epoch {:3d} Train loss: {:8.6f}".format(trainer.state.epoch,
110                                                        trainer.state.output))
111         train_loss.append(trainer.state.output)
```

Figura 5.4: Funcţie efectuată la fiecare iteraţie

Funcţia permite executare unei comenzi la fiecare iteraţie, în cazul meu afişarea costului. În mod similar pot executa o comandă la începutul unei epoci.

```
114     @trainer.on(engine.Events.EPOCH_COMPLETED)
115     def log_validation_loss(trainer):
116         evaluator.run(valid_loader)
117         metrics = evaluator.state.metrics
118         print("Epoch {:3d} Valid loss: {:8.6f} ←".format(
119             trainer.state.epoch, metrics['loss']))
120         valid_loss.append(metrics['loss'])
```

Figura 5.5: Funcţie efectuată la începutul fiecărei epoci

În cazul în care loss-ul modelului se apropie de convergenţă, rata de învăţare scade prin funcţia "lr_scheduler.ReduceLROnPlateau" şi astfel se asigură convergenţa.

```
123     lr_reduce = lr_scheduler.ReduceLROnPlateau(optimizer,
124                                                verbose=args.verbose,
125                                                **LR_REDUCE_PARAMS)
126
127     @evaluator.on(engine.Events.COMPLETED)
128     def update_lr_reduce(engine):
129         loss = engine.state.metrics['loss']
130         lr_reduce.step(loss)
```

Figura 5.6: Actualizarea ratei de învăţare

Antrenarea se opreşte automat în cazul în care costul modelului nu scade timp de 8 epoci, comandă executată prin "ignite.handlers.EarlyStopping()". În continuare la terminarea fiecărei epoci, modelul este salvat automat prin creearea unui "checkpoint".

```
140     checkpoint_handler = ignite.handlers.ModelCheckpoint(
141         "output/models/checkpoints",
142         model_name,
143         score_function=score_function,
144         n_saved=5,
145         require_empty=False,
146         create_dir=True)
147     evaluator.add_event_handler(engine.Events.EPOCH_COMPLETED,
148                                 checkpoint_handler, {"model": model})
149
150     trainer.run(train_loader, max_epochs=MAX_EPOCHS)
151     torch.save(model.state_dict(), model_path)
```

Figura 5.7: Salvarea modelului

Metoda "collate_fn" este folosită pentru organizarea structurii datelor la intrarea în funcţia de antrenare.

```
32  def collate_fn(batches):
33      videos = [batch[0] for batch in batches]
34      specs = [batch[1] for batch in batches]
35      max_v = max(video.shape[1] for video in videos)
36      max_s = max(spect.shape[1] for spect in specs)
37      videos = [F.pad(video, pad=(0, 0, 0, 0, 0, max_v - video.shape[1])) for video in videos]
38      specs = [F.pad(spect, pad=(0, max_s - spect.shape[1], 0, 0)) for spect in specs]
39      video = torch.stack(videos)
40      spect = torch.stack(specs)
41      return video, spect
```

Figura 5.8: Organizarea datelor din loturi

5.2 **Test.py**

Fişierul "test.py" este folosit pentru testarea unui model, returnând spectrogramele corespunzătoare setului de testare. Codul complet este scris în anexa B.

Prima parte a fişierului este identică cu cea din "main.py" deoarece sunt încărcate datele de testare dar în continuare modelul intră în starea de evaluare prin "model.eval()". Această metodă opreşte actualizarea parametrilor în timpul rulării modelului. Pentru testare nu folosesc librăria "ignite" ci definesc o instrucţiune repetitivă ce compune costul la fiecare iteraţie. Calculul costului mediu din cadrul setului de testare se face prin adunarea costului total şi împărţirea la numărul de iteraţii, şi apoi salvez spectrogramele de la ieşire.

```
48      with torch.no_grad():
49          for i, (data, target) in enumerate(tqdm(loader)):
50              data, target = data.to(device), target.to(device)
51              output = model(data)
52
53              loss = mse(output, target)
54              np.save(f"output/specs/{model_name}{i}.npy", output.cpu().numpy())
55              test_loss += loss.item()
56
57      test_loss = test_loss / len(loader)
58      print(test_loss)
```

Figura 5.9: Testarea unui model

5.3nn.py

Acest fişier conţine reţelele corespunzătoare primelor două arhitecturi prezentate. Codul complet este scris în anexa C.

Definirea reţelelor a fost făcută utilizând librăria PyTorch. Întâi sunt definite straturile ce vor fi folosite apoi modelul este construit în metoda model.forward(). Exemplu pentru definirea ResNet18:

```
168         self.encoder = resnet18()
169         self.encoder.conv1 = nn.Conv2d(
170             K,
171             64,
172             kernel_size=(7, 7),
173             stride=(2, 2),
174             padding=(3, 3),
175             bias=False,
176         )
177 
178         self.encoder = nn.Sequential(*list(self.encoder.children())[:-1])
```

Figura 5.10: Implementare ResNet18

La construirea reţelei, pe lângă straturile descrise, folosesc comenzi pentru schimbarea dimensiunilor intrării pentru funcţii ce necesită acest lucru.

```
69          x = self.conv1(x)  # B x C x S x H x W
70          x = self.batchnorm(x)
71          x = self.relu(x)
72 
73          x = self.dropout2(x)
74          x = x.transpose(1, 2)
75          x = x.reshape(B*S, 64, H, W)
76          x = self.encoder(x)
77          x = x.squeeze().view(B, S, 512)
78          x = x.permute(1, 0, 2)
79          x, _ = self.gru(x)
80          x = x.permute(1, 0, 2)  # B, S, D
81          x = self.elu(x)
82          x = self.dropout2(x)
```

Figura 5.11: Implementarea codorului

5.4dataset.py

În fişierul dataset.py sunt definite metodele pentru prelucrarea datelor. Metodele principale folosite sunt "xTSSample.load()" şi "xTSDataset.load()". Acestea returnează video-urile după procesare şi spectrogramele corespunzătoare.

```
86          while fc < frameCount and ret:
87              ret, frame = cap.read()
88              try:
89                  buf[fc] = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
90
91              except:
92                  print(self.person, self.file)
93              self.crop[fc] = buf[fc][top-k:bot + k, left-k:right + k]
94
95              fc += 1
```

Figura 5.12: Încărcarea şi decuparea video-urilor

Metoda "xTSSample.load()" primeşte la intrare un nume de fişier şi codul vorbitorului şi returnează video-ul şi spectrograma corespunzătoare. Metoda "xTSDataset.load()" primeşte la intrare un set de nume de fişiere şi codurile vorbitorilor pentru fiecare fişier, şi folosind "xTSSample.load()" încarcă şi returnează toate video-urile şi spectrogramele pentru fişierele specificate în setul de date. În plus, este aplicată o transformare peste setul de date înainte de returnare. Codul complet este scris în anexa D.

```
284         stream = xTSSample(self.root, self.folder[idx], self.file[idx])
285         stream.load()
286         stream.data = stream.data.type(torch.FloatTensor)
287         stream.data = stream.data.cuda()
288         D, C, H, W = stream.data.shape
289
290         for i in range(C):
291             stream.data[:, i, :, :] = self.transform(stream.data[:, i, :, :])
292
293         return stream.data[0,:,:,:], stream.spec
```

Figura 5.13: Încărcarea setului de date

5.5Flowtron.py

"Flowtron.py" conţine reţeaua neurală alterată din proiectul "NVIDIA/FlowTron" . Pentru adaptarea reţelei la această lucrare, am înlocuit codorul cu cel din arhitectura prezentată la capitolul 5.1, am schimbat funcţiile prezente pentru a accepta o intrare 2D. Codul complet este scris în anexă E.

Un exemplu de funcţie schimbată este "FlowTronLoss". Iniţial această funcţie de cost era compusă folosind dimensiunile vectorului de text pentru a forma o mască a intrării. Cum toate înregistrările video din setul de date sunt de aceeaşi dimensiune iar forma intrării este diferită, metoda de calcul a putut fi schimbată într-una mai simplă:

```
82          n_elements = np.prod(z.shape)
83          for i, log_s in enumerate(log_s_list):
84              if i == 0:
85                  log_s_total = torch.sum(log_s)
86              else:
87                  log_s_total = log_s_total + torch.sum(log_s)
88          loss = torch.sum(z * z) / 2 - log_s_total
89          loss = loss / n_elements
90          return loss
```

Figura 5.14: Implementarea funcţiei de cost FlowtronLoss

5.6Flowtron_train.py

Metoda de antrenare a reţelelelor neurale autoregresive este diferită faţă de primele două reţele prezentate şi de aceea este folosit alt fişier Python. Reţelele neurale autoregresive învaţă o funcţie pentru reprezentarea datelor şi de aceea spectrograma este de asemenea folosită la intrare iar costul nu poate fi calculat folosind metodele tradiţionale. Codul complet este scris în anexa F.

```
146     for epoch in range(epoch_offset, 40):
147         print("Epoch: {}".format(epoch))
148         for batch in train_loader:
149             model.zero_grad()
150             mel, video = batch
151             mel, video = mel.cuda(), video.cuda()
152             z, log_s_list, gate_pred,  mean, log_var, prob = model(
153                 mel, video)
154             loss = criterion.forward((z, log_s_list, gate_pred, mean, log_var, prob),
155                                      lengths=torch.from_numpy(np.repeat(225,8)).cuda())
156             reduced_loss = loss.item()
157             loss.backward()
158             optimizer.step()
```

Figura 5.15: Implementarea antrenării Flowtron

5.7Flowtron_infer.py

Inferenţă reprezintă compunerea unei spectrograme prin aplicarea unei funcţii peste o distribuţie simplă. Această metodă există doar pentru reţelele autoregresive şi returnează spectrograma compusă.

```
76      with torch.no_grad():
77          residual = torch.from_numpy(np.float32(np.zeros((1,80,225)))).to(device)
78
79          video = video.to(device)
80          mels = model.infer(
81              residual,  video[0].unsqueeze(0))
```

Figura 5.16: Inferenţă Flowtron

În cadrul acestui fişier este inclusă şi metodă de transformare din spectrogramă în audio folosind funcţia DeepConvTTS() din audio.py. Codul complet este scris în anexa G.

```
86      deep = DeepConvTTS(sampling_rate=19300)
87      audio = deep.mel_to_audio(mels)
88      audio *= 32767 / max(0.01, np.max(np.abs(audio)))
89      write(os.path.join('results/', 'sigma{}.wav'.format(_sigma)),
90            19300, audio.astype(np.int16))
```

Figura 5.17: Transformarea din spectrogramă în audio

Capitolul 6

Rezultate

6.1 Parametrii folosiţi

Antrenarea unui model s-a efectuat cu o rată de învăţare de 0.0001 ce scade odată cu apropierea erorii de convergenţă. Modelul termină antrenarea după 8 epoci în care eroarea de validare nu a scăzut. Deşi modelul este programat să se oprească automat, în cazul în care eroarea de validare continuă să scadă pentru un număr mare de epoci, numărul maxim de epoci este 128.

6.2 Metrici de evaluare

Pentru evaluarea rezultatelor am folosit eroarea medie pătrată şi Mel Cepstral Distortion pe acelaşi set de date în toate testele. Mel Cepstral Distortion a fost folosit pentru compararea rezultatelor din diferite arhitecturi încât costul este compus diferit pentru fiecare arhitectură.

Mel Cepstral Distortion este o măsură a diferenţei dintre 2 secvenţe mel cepstra. Este folosită pentru determinarea calităţii a vocii sintetizate. Cu cât valoarea MCD este mai mică, cu atât vocea sintetizată este mai apropiată de cea originală.

Pentru calculul MCD întâi se află coeficienţii cepstral C în cazul audio original şi ĉ în cazul audio sintetizat. MCD se calculează cu formula(Sursa: [10]):

$$\frac{10\sqrt{2}}{\ln 10}\frac{1}{T}\sum_{t=1}^{T}\sqrt{\sum_{i}\left(C_{ti}-\hat{C}_{ti}\right)^{2}}$$

Eroarea medie pătrată reprezintă pătratul diferenţei dintre valoarea adevărată şi valoarea prezisă de către reţea. Dacă y este valoarea adevărată şi ŷ este valoarea prezisă atunci eroarea medie pătrată este:

$$L(y,\hat{y})=\frac{1}{N}\sum_{i=0}^{N}(y-\hat{y}_i)^2$$

În cadrul arhitecturilor au fost testate metode populare de îmbunătăţire a reţelei precum creşterea padding-ului, normalizarea datelor şi adăugarea la intrare a derivatelor de ordinul 1 şi 2 a video-urilor.

Toate antrenările au rulat până la oprirea automată a modelului prin mecanismul explicat la capitolul 5.1.

6.3 Baza de date

Baza de date folosită este "GRID corpus". Ea conţine 39 de vorbitori, fiecare având 1000 de înregistrări video. Limba este engleza iar cuvintele vorbite nu sunt unice, acelaşi cuvânt poate apărea în mai multe înregistrări. Toţi cei 39 de vorbitori au faţa îndreptată spre camera video, iar cuvintele sunt clar rostite. Durata unui video este de 3 secunde.

Antrenarea reţelei s-a efectuat folosind primele 900 de înregistrări ale persoanei "s1" în cazul antrenării pe o singură persoană, pentru 2 persoane am folosit primele 900 de înregistrări ale persoanelor "s1" şi "s2".

Setul de validare este format din 50 de video-uri şi reprezintă video-urile 901-950 ale unui vorbitor.

Pentru evaluarea rezultatelor am compus setul de testare folosind fişierele 951-1000 ale vorbitorului "s1" în cazul erorii medii pătrate. Pentru calculul Mel Cepstral Distortion am folosit aceeaşi înregistrare, video-ul 951 din cadrul vorbitorului "s1", numele fişierului fiind "swav1a".

6.4 Creşterea padding-ului

Valoarea iniţială a padding-ului folosit este 10 şi cuprinde gura şi o porţiune din nas.

Figura 6.1: Imagine preprocesata cu padding 10

Prin creşterea paddingului putem vedea o parte mai mare din secţiunea obrajilor şi nasului, acestea schimbâdu-şi forma la pronunţarea diferitelor cuvinte şi putem testa impactul lor la antrenarea modelului.

Figura 6.2: Imagine preprocesata cu padding 25

Rezultatele sunt:

num. subjects	padding	first and second derivatives	normalization	decoder	sampling frequency	loss ↓
1	10	yes	standardization	MLP	22 KHz	0.0058
1	25	yes	standardization	MLP	22 KHz	0.0059

Tabel 6.1: Rezultate padding

Impactul adăugării padding-ului a rezultat a fi detrimental antrenării, şi nu am folosit padding crescut în continuare.

6.5 Adăugarea derivatelor de ordinul 1 şi 2

Adăugarea derivatelor de ordinul 1 şi 2 este o modalitatea populară în procesarea imaginilor pentru identificarea mai uşoară a marginilor de către model.

Derivatele au fost compuse folosit funcţia "numpy.diff()" .

```
44  def diff(buf_input):
45      buf_input = np.pad(buf_input, ((1, 0), (0, 0), (0, 0)), 'edge')
46      buf_output = np.diff(buf_input, axis=0)
47      return buf_output
100         diff_video[0, :, :, :] = self.crop
101         x = diff(self.crop)
102         diff_video[1, :, :, :] = diff(self.crop)
103         diff_video[2, :, :, :] = diff(diff_video[1, :, :, :])
```

Figura 6.3: Calcularea derivatelor de ordinul 1 şi 2

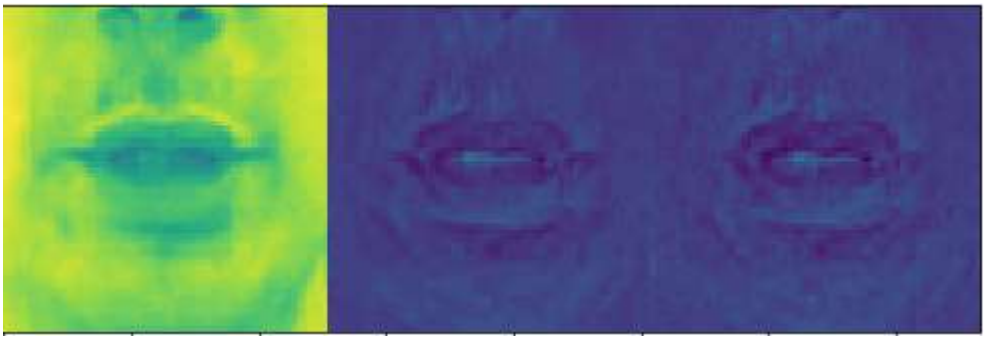


Figura 6.4: Derivatele de ordinul 1 şi 2

num. subjects	padding	first and second derivatives	normalization	decoder	sampling frequency	loss ↓
1	10	no	standardization	MLP	22 KHz	0.00575
1	10	yes	standardization	MLP	22 KHz	0.00577
2	10	no	standardization	MLP	22 KHz	0.0056
2	10	yes	standardization	MLP	22 KHz	0.0053
5	10	no	standardization	MLP	22 KHz	0.0056
5	10	yes	standardization	MLP	22 KHz	0.0053

Tabel 6.2: Rezultate obţinute prin adăugarea derivatelor de ordinul 1 şi 2, arhitectură MLP

Pentru o persoană adăugarea derivatelor nu prezintă un avantaj dar pentru un numai mai mare de persoane informaţiile despre marginile gurii ajută reţeaua să înveţe mai bine caracteristicile.

Această testare a fost efectuată şi pe arhitectura bazată pe reţele neurale convoluţionale.

num. subjects	padding	first and second derivatives	normalization	decoder	sampling frequency	loss ↓
1	10	no	standardization	conv	19.3 KHz	0.0052
1	10	yes	standardization	conv	19.3 KHz	0.0052
2	10	no	standardization	conv	19.3 KHz	0.0053
2	10	yes	standardization	conv	19.3 KHz	0.0052

Tabel 6.3: Rezultate adăugarea derivatelor de ordinul 1 şi 2, arhitectură convolutionala

Rezultatele sunt similare cu cele de la arhitectura MLP.

6.6 Normalizarea

Valorile din înregistrările video sunt între [0,255] dar în cazul derivatelor există şi valori negative, iar plaja lor de valori este diferită la fiecare video. Pentru ca reţeaua să lucreze cu valori similare pentru video-uri şi derivate am aplicat funcţia de normalizare pe toate cele 3, astfel încât valorile să fie între [-1,1].

Funcţia aplicată este torch.transforms.normalize iar pentru aplicare am calculat media şi abaterea standard a video-urilor şi derivatelor de ordinul 1 şi 2 pentru 5 persoane, în total 5000 video-uri.

Codul pentru calculul mediei şi abaterea standard:

```
57      for data , _ in train_loader:
58          batch_samples = data.size(0)
59          data = data.view(batch_samples, data.size(1), -1)
60          mean += data.mean(2).sum(0)
61          std += data.std(2).sum(0)
62          nb_samples += batch_samples
63
64      mean /= nb_samples
65      std /= nb_samples
```

Figura 6.5: Calculul mediei şi abaterea standard

Aplicând normalizarea avem rezultatele:

num. subjects	padding	first and second derivatives	normalization	decoder	sampling frequency	loss ↓
2	10	yes	[0, 1]	MLP	22 KHz	0.0060
2	10	yes	standardization	MLP	22 KHz	0.0053

Tabel 6.4: Rezultate normalizare a datelor

Aplicarea normalizării are un impact mare asupra costului iar în audio rezultat cuvintele sunt mai clare.

6.7 Diferenţa dintre arhitecturi

Din rezultatele anterioare am observat că pentru reproducerea cuvintelor, folosirea padding=10 derivatelor şi normalizării ajută la performanţa reţelei. Pentru compararea arhitecturilor am folosit Mel Cepstral Distortion pe vocea sintetizată rezultată comparativ cu cea originală.

num. subjects	padding	first and second derivatives	normalization	decoder	sampling frequency	MCD ↓
1	10	yes	standardization	MLP	22 KHz	183
1	10	yes	standardization	conv	19.3 KHz	169
1	10	yes	standardization	autoregressive	19.3 Khz	182

Tabel 6.5: Rezultate finale pentru o persoană

Putem observa că decodorul convoluţional reproduce cel mai bine audio pentru o persoană.

num. subjects	padding	first and second derivatives	normalization	decoder	sampling frequency	MCD
2	10	yes	standardization	conv	19.3 KHz	169
2	10	yes	standardization	autoregressive	19.3 Khz	191
2	10	yes	standardization	MLP	22 KHz	195

Tabel 6.6: Rezultate finale pentru două persoane

Acest lucru rămâne valabil şi pentru 2 persoane.

Concluzii

Obiectivul lucrării este antrenarea unei reţele neurale pentru a fi folosită cu scopul sintetizării vocii pornind de la mişcarea buzelor. Arhitecturile analizate în cadrul îndeplinirii scopului sunt de 3 tipuri:

- MLP
- Convoluţionale
- Autoregresive

Implementarea şi testarea s-a efectuat folosind limbajul de programare Python prin librării specifice construirii reţelelor neurale artificiale.

Alegerea arhitecturilor s-a bazat pe metode populare întâlnite în articole de specialitate. Primul pas l-a reprezentat compunerea acestora pe baza modelului codor-decodor întâlnit în aplicaţii cu scopul sintetizării vorbirii. În continuare am decis folosirea reţelelor neurale convoluţionale în codor pentru extragerea caracteristicilor din înregistrările video, iar reţelele neurale recurente deoarece informaţiile temporale sunt importante în redarea vocii. Decodorul MLP reprezintă varianta cea mai uşoară a implementării unui decodor, urmat de decodorul convoluţional iar apoi cel autoregresiv.

Pe lângă alegerea arhitecturilor am căutat şi metode de procesare a datelor pentru îmbunătăţirea performanţelor modelelor. Adăugarea derivatelor de ordinul 1 şi 2 pentru detecţia marginilor şi normalizarea datelor au avut un efect pozitiv asupra performanţei arhitecturii.

Vocea sintetizată din prezicerile arhitecturilor este în toate cazurile inteligibilă, dar folosind 2 sau mai multe persoane la antrenare, vocea este reprodusă folosind toate persoanele. Acest lucru face audio să pară nenatural.

Ca şi dezvoltări ulterioare, primul pas ar fi creearea unei voci comune pentru fiecare persoană astfel încât vocea reprodusă să nu reprezinte un amalgam de voci din setul de antrenare. În continuare există mai multe direcţii posibile:

- Prefecţionarea arhitecturii astfel încât vocea rezultată să pară cât mai naturală
- Integrarea posibilităţii de a funcţiona in timp real
- Identificarea emoţiilor şi influenţarea vocii pe baza lor

Bibliografie

[1] Saad Albawy, Saad Alzawy, Osman N. Ucan, Oguz Bayat,
"Social touch gesture recognition using deep neural network "

[2] Nahua Kang, Multi-Layer Neural Networks with Sigmoid Function
https://towardsdatascience.com/multi-layer-neural-networks-with-sigmoid-function-deep-learning-for-rookies-2-bf464f09eb7f, accesat la data: 15.06.2020

[3] Activation Functions în Neural Networks
https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6 , accesat la data: 15.06.2020

[4] Activation functions,
https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html#elu, accesat la data: 13.06.2020

[5] "Tensorflow maxpool: Working with CNN Max Pooling layers în TensorFlow"
https://missinglink.ai/guides/tensorflow/tensorflow-maxpool-working-cnn-max-pooling-layers-tensorflow/ , accesat la data: 11.06.2020

[6] Ian Goodfellow, Yoshua Bengio, Aaron Courville. Deep learning. MIT press, 2016. URL: https://www.deeplearningbook.org/

[7] Devangini Patel, Facial Landmarks Detection
https://devanginiblog.wordpress.com/2017/09/05/facial-landmark-detection/ , accesat la data: 08.06.2020

[8] Leland Roberts. „Understanding the Mel Spectrogram”.
https://medium.com/analytics-vidhya/understanding-the-mel-spectrogram-fca2afa2ce53 accesat la data de: 13.06.2020

[9] Rafael Valle, Kevin Shih, Ryan Prenger, Bryan Catanzaro. Flowtron: an autoregressive flow-based generative network for text-to-speech synthesis, 2020

[10] Dan Oneaţă,
https://dsp.stackexchange.com/questions/56391/mel-cepstral-distortion accesat la data: 13.06.2020

Anexe

Anexa A – Train.py

```
import argparse
import os
import os.path
import pdb
import sys
import numpy as np

import torch
import torch.nn as nn
import torch.optim.lr_scheduler as lr_scheduler
import torch.utils.data
from torch.nn import functional as F
import ignite.engine as engine
import ignite.handlers
import torchvision

import src.dataset

from models import MODELS

ROOT = os.environ.get("ROOT", "")

SEED = 1337
MAX_EPOCHS = 128
PATIENCE = 8
LR_REDUCE_PARAMS = {
    "factor": 0.2,
    "patience": 4,
}


def collate_fn(batches):
    videos = [batch[0] for batch in batches]
    spects = [batch[1] for batch in batches]
    max_v = max(video.shape[1] for video in videos)
    max_s = max(spect.shape[1] for spect in spects)
    videos = [F.pad(video, pad=(0, 0, 0, 0, 0, max_v - video.shape[1])) for video in videos]
    spects = [F.pad(spect, pad=(0, max_s - spect.shape[1], 0, 0)) for spect in spects]
    video = torch.stack(videos)
    spect = torch.stack(spects)
    return video, spect


def main():
    parser = argparse.ArgumentParser(description="Evaluate a given model")
    parser.add_argument("--model-type",
                        type=str,
                        required=True,
                        choices=MODELS,
                        help="which model type to train")
    parser.add_argument("-m",
                        "--model",
                        type=str,
                        default=None,
                        required=False,
                        help="path to model to load")
    parser.add_argument("-v",
                        "--verbose",
                        action="count",
                        help="verbosity level")
    args = parser.parse_args()
    mu = [1.4024e+02, -4.9944e-03, 3.5836e-04]
    sigma = [21.9047,  4.4827,  5.8206]
    transform = torchvision.transforms.Normalize(mean=mu, std=sigma)
    print(args)
    train_loss = []
```

```
valid_loss = []
model = MODELS[args.model_type]()
if args.model is not None:
    model_path = args.model
    model_name = os.path.basename(args.model)
    model.load(model_path)
else:
    model_name = f"{args.model_type}"
    model_path = f"output/models/{model_name}.pth"

train_dataset = src.dataset.xTSDataset(ROOT, "train", transform=transform)
valid_dataset = src.dataset.xTSDataset(ROOT, "valid", transform=transform)

train_loader = torch.utils.data.DataLoader(
    train_dataset,
    batch_size=8,
    collate_fn=collate_fn,
    shuffle=True)

valid_loader = torch.utils.data.DataLoader(
    valid_dataset,
    batch_size=8,
    collate_fn=collate_fn,
    shuffle=False)

# ignite_train = DataLoader(train_loader, shuffle=True)

optimizer = torch.optim.Adam(model.parameters(), lr=0.0001)
loss = nn.MSELoss()

device = 'cuda'

trainer = engine.create_supervised_trainer(model, optimizer, loss, device=device)

evaluator = engine.create_supervised_evaluator(
    model,
    metrics={'loss': ignite.metrics.Loss(loss)},
    device=device,
)

@trainer.on(engine.Events.ITERATION_COMPLETED)
def log_training_loss(trainer):
    print("Epoch {:3d} Train loss: {:8.6f}".format(trainer.state.epoch,
                                                   trainer.state.output))
    train_loss.append(trainer.state.output)


@trainer.on(engine.Events.EPOCH_COMPLETED)
def log_validation_loss(trainer):
    evaluator.run(valid_loader)
    metrics = evaluator.state.metrics
    print("Epoch {:3d} Valid loss: {:8.6f} ←".format(
        trainer.state.epoch, metrics['loss']))
    valid_loss.append(metrics['loss'])


lr_reduce = lr_scheduler.ReduceLROnPlateau(optimizer,
                                           verbose=args.verbose,
                                           **LR_REDUCE_PARAMS)

@evaluator.on(engine.Events.COMPLETED)
def update_lr_reduce(engine):
    loss = engine.state.metrics['loss']
    lr_reduce.step(loss)

def score_function(engine):
    return -engine.state.metrics['loss']

early_stopping_handler = ignite.handlers.EarlyStopping(
    patience=PATIENCE, score_function=score_function, trainer=trainer)
evaluator.add_event_handler(engine.Events.EPOCH_COMPLETED,
                            early_stopping_handler)
```

```
    checkpoint_handler = ignite.handlers.ModelCheckpoint(
        "output/models/checkpoints",
        model_name,
        score_function=score_function,
        n_saved=5,
        require_empty=False,
        create_dir=True)
    evaluator.add_event_handler(engine.Events.EPOCH_COMPLETED,
                                checkpoint_handler, {"model": model})

    trainer.run(train_loader, max_epochs=MAX_EPOCHS)
    torch.save(model.state_dict(), model_path)
    print("Model saved at:", model_path)
    np.save('train_loss', np.asarray(train_loss))
    np.save('valid_loss', np.asarray(valid_loss))


if __name__ == "__main__":
    main()
```

Anexa B – Test.py

```
import argparse
import os
import os.path
import pdb
import sys
import numpy as np

import numpy as np
import torch
import torch.nn as nn
import torch.optim.lr_scheduler as lr_scheduler
import torch.utils.data
from tqdm import tqdm
import torchvision
from models import MODELS
import src.dataset
from train import collate_fn



BATCH_SIZE = 8
DEVICE = "cuda"

ROOT = os.environ.get("ROOT", "")


def predict(args):
    mu = [1.4024e+02, -4.9944e-03, 3.5836e-04]
    sigma = [21.9047, 4.4827, 5.8206]
    transform = torchvision.transforms.Normalize(mean=mu, std=sigma)
    dataset = src.dataset.xTSDataset(ROOT, "test", transform=transform)
    loader = torch.utils.data.DataLoader(
        dataset, batch_size=BATCH_SIZE, collate_fn=collate_fn, shuffle=False
    )
    model = MODELS[args.model_type]()
    model_name = f"{args.model_type}"
    model_path = f"output/models/{model_name}.pth"
    model.load_state_dict(torch.load(model_path))

    n_samples = len(dataset)
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model = model.to(device)
    model.eval()
    mse = nn.MSELoss()
    preds = np.zeros((1025, 225))
    test_loss = 0
```

```
    with torch.no_grad():
        for i, (data, target) in enumerate(tqdm(loader)):
            data, target = data.to(device), target.to(device)
            output = model(data)

            loss = mse(output, target)
            np.save(f"output/spects/{model_name}{i}.npy", output.cpu().numpy())
            test_loss += loss.item()

    test_loss = test_loss / len(loader)
    print(test_loss)


def main():
    parser = argparse.ArgumentParser(description="Test a given model")
    parser.add_argument("--model-type",
                        type=str,
                        required=True,
                        choices=MODELS,
                        help="which model type to use")
    args = parser.parse_args()
    predict(args)


if __name__ == "__main__":
    main()
```

Anexa C – nn.py

```
from typing import List, Union, Dict
import collections
import enum

from types import SimpleNamespace
import numpy as np
import torch
import torch.nn as nn
from torchvision.models import resnet18
import torchvision
import pdb
from hparams import hparams
import src.dataset
get_same_padding = lambda s: (s - 1) // 2


class resnet(nn.Module):

    def __init__(self):
        super(resnet, self).__init__()
        kwargs = dict(kernel_size=3, stride=1, padding=1, bias=True)
        K = 64
        D_gru = 128
        self.conv1 = nn.Conv3d(3, K, **kwargs)
        self.conv2 = nn.Conv3d(K, K, **kwargs)
        self.batchnorm = nn.BatchNorm3d(K)
        self.batch1 = nn.BatchNorm1d(2000)
        self.batch2 = nn.BatchNorm1d(2000)
        self.batch3 = nn.BatchNorm1d(2000)
        self.relu= nn.LeakyReLU(inplace=True)
        self.elu = nn.ELU(alpha=1.0)
        self.gru = nn.GRU(512, D_gru)
        self.linear = nn.Linear(75*128, 2000)
        self.linear2_1 = nn.Linear(2000, 2000)
        self.linear2_2 = nn.Linear(2000, 2000)
        self.linear2_3 = nn.Linear(2000, 2000)

        self.linear3 = nn.Linear(2000, 257*80)
        self.dropout2 = nn.Dropout(0.25)
        self.dropout4 = nn.Dropout(0.4)
        self.sigmoid = nn.Sigmoid()
```

```
        self.encoder = resnet18()
        self.encoder.conv1 = nn.Conv2d(
            K,
            64,
            kernel_size=(7, 7),
            stride=(2, 2),
            padding=(3, 3),
            bias=False,
        )

        self.encoder = nn.Sequential(*list(self.encoder.children())[:-1])

    def forward(self, x):
        try:
            B, D, S, H, W = x.shape
        except:
            B, S, H, W = x.shape
            x = x.unsqueeze(1)

        x = x.float()  # scale data in [0, 1]

        #x = x.float()
        x = self.conv1(x)  # B x C x S x H x W
        x = self.batchnorm(x)
        x = self.relu(x)

        x = self.dropout2(x)
        x = x.transpose(1, 2)
        x = x.reshape(B*S, 64, H, W)
        x = self.encoder(x)
        x = x.squeeze().view(B, S, 512)
        x = x.permute(1, 0, 2)
        x, _ = self.gru(x)
        x = x.permute(1, 0, 2)  # B, S, D
        x = self.elu(x)
        x = self.dropout2(x)

        #Flatten data
        x = x.reshape(B, 75*128)

        #1st linear layer
        x = self.linear(x)
        x = self.batch1(x)
        x = self.elu(x)
        x = self.dropout2(x)

        # 2nd linear layer
        x = self.linear2_1(x)
        x = self.batch2(x)
        x = self.elu(x)
        x = self.dropout2(x)

        # 3rd linear layer
        x = self.linear2_2(x)
        x = self.batch3(x)
        x = self.elu(x)
        x = self.dropout2(x)

        # 257, 80
        x = x.unsqueeze(1)
        x = x.reshape(B, 257, 80)

        return x


class resnet2(nn.Module):

    def __init__(self):
        super(resnet2, self).__init__()
        kwargs = dict(kernel_size=3, stride=1, padding=1, bias=True)
        K = 64
        D_gru = 128
```

```
        self.conv1 = nn.Conv3d(3, K, **kwargs)
        self.conv2 = nn.Conv3d(K, K, **kwargs)
        self.batchnorm = nn.BatchNorm3d(K)
        self.batch1 = nn.BatchNorm1d(128)
        self.batch2 = nn.BatchNorm1d(128)
        self.batch3 = nn.BatchNorm1d(128)
        self.dropout2 = nn.Dropout(0.25)
        self.dropout4 = nn.Dropout(0.4)
        self.relu = nn.LeakyReLU(inplace=True)
        self.elu = nn.ELU(alpha=1.0)
        self.gru = nn.GRU(512, D_gru)
        self.conv1d1 = nn.Conv1d(128, 128, kernel_size=3, padding=1)
        self.conv1d2 = nn.Conv1d(128, 128, kernel_size=3, padding=1)
        self.conv1d3 = nn.Conv1d(128, 80, kernel_size=3, padding=1)
        self.convtranspose1d = nn.ConvTranspose1d(128, 128, 3, stride=3)
        self.sigmoid = nn.Sigmoid()
        self.encoder = resnet18()
        self.encoder.conv1 = nn.Conv2d(
            K,
            64,
            kernel_size=(7, 7),
            stride=(2, 2),
            padding=(3, 3),
            bias=False,
        )

        self.encoder = nn.Sequential(*list(self.encoder.children())[:-1])

    def forward(self, x):
        try:
            B, D, S, H, W = x.shape
        except:
            B, S, H, W = x.shape
            x = x.unsqueeze(1)

        x = x.float()  # scale data in [0, 1]

        x = self.conv1(x)  # B x C x S x H x W

        x = self.batchnorm(x)
        x = self.relu(x)

        x = self.dropout2(x)

        x = x.transpose(1, 2)

        x = x.reshape(B*S, 64, H, W)

        x = self.encoder(x)

        x = x.squeeze().view(B, S, 512)
        x = x.permute(1, 0, 2)

        x, _ = self.gru(x)

        x = x.permute(1, 2, 0)  # B, D, C   8, 128, 75

        x = self.convtranspose1d(x) # 8,128,225
        x = self.batch3(x)
        x = self.elu(x)


        x = self.conv1d1(x)  # 8, 128, 225
        x = self.batch1(x)
        x = self.elu(x)

        x = self.conv1d2(x)  #8, 128, 225
        x = self.batch2(x)
        x = self.elu(x)
```

```
        x = self.conv1d3(x)  #8, 80, 225
        x = self.sigmoid(x)

        x = x.permute(0, 2, 1)  # B, C, D  8,80,225
        return x
```

Anexa D – dataset.py

```
from typing import List, Callable, Union, Tuple
import datetime
import inspect
import math
import os
import random


from torch.nn import functional as F
from moviepy.editor import *
import json
import cv2
import numpy as np
import h5py
import torch
import torch.utils.data
from typing import List, Callable, Union, Tuple
import torchvision
import datetime
import inspect
import math
import os
import random

import librosa
from PIL import Image

from torch.nn import functional as F
from moviepy.editor import *
import json
import cv2
import numpy as np
import torch
import torch.utils.data
import pdb
import scipy
import matplotlib.pyplot as plt
import scipy.io.wavfile
from scipy import signal
from scipy.io import wavfile
from audio import DeepConvTTS


def diff(buf_input):
    buf_input = np.pad(buf_input, ((1, 0), (0, 0), (0, 0)), 'edge')
    buf_output = np.diff(buf_input, axis=0)
    return buf_output


class xTSSample(object):
    def __init__(self,
                 root: str,
                 person: str,
                 file: str):
        self.root = root
        self.person = person
        self.data = None
        self.file = file
        self.crop = None
        self.spec = None

        self.paths = {
            "face": os.path.join(root, "face-landmarks"),
```

```
        "audio": os.path.join(root, "audio-from-video"),
        "video": os.path.join(root, "video"),
    }

def load(self):
    """ Crop lips"""
    k = 10
    f = open(os.path.join(self.paths["face"], self.person, self.file + ".json"))
    fl = json.load(f, strict=False)
    top = fl[0][51][1] - k
    bot = fl[0][58][1] + k
    left = fl[0][49][0] - k
    right = fl[0][55][0] + k
    cap = cv2.VideoCapture(os.path.join(self.paths["video"],  self.person, self.file + ".mpg"))
    frameCount = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
    frameWidth = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
    frameHeight = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
    buf = np.empty((frameCount, frameHeight, frameWidth), np.dtype('float32'))
    self.crop = np.empty((frameCount, bot - top + 2*k, right - left + 2*k), np.dtype('float32'))
    fc = 0
    ret = True

    while fc < frameCount and ret:
        ret, frame = cap.read()


        try:
            buf[fc] = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        except:
            print(self.person, self.file)
        self.crop[fc] = buf[fc][top-k:bot + k, left-k:right + k]

        fc += 1



    cap.release()
    self.crop = [np.array(Image.fromarray(im).resize((64, 64))) for im in self.crop]
    diff_video = np.empty((3, np.shape(self.crop)[0], np.shape(self.crop)[1], np.shape(self.crop)[2]))
    diff_video[0, :, :, :] = self.crop
    x = diff(self.crop)
    diff_video[1, :, :, :] = diff(self.crop)
    diff_video[2, :, :, :] = diff(diff_video[1, :, :, :])
    a, b, c, d = diff_video.shape
    x = diff_video[2]
    x2 = np.sum(x)
    x2 = x2/(b*c*d)
    self.crop = np.stack(self.crop)
    self.data = torch.from_numpy(diff_video)   #for derivatives
    #self.data = torch.from_numpy(self.crop) #without derivatives


    """ Create spectrogram"""
    path = os.path.join(self.paths["audio"], self.person, self.file + ".mpg.wav")
    # sample_rate, samples = scipy.io.wavfile.read(path)
    #wav = load_wav(path)
    #self.spec = spectrogram(wav)
    # frequencies, times, spectrogram = signal.spectrogram(samples, sample_rate)
    # fmin = 10
    # fmax = 4000
    # freq_slice = np.where((frequencies >= fmin) & (frequencies <= fmax))
    "frequencies = frequencies [freq_slice]"
    "spectrogram = spectrogram[freq_slice, :][0]"
    deep = DeepConvTTS(sampling_rate=19300)
    wav = deep.load_audio(path)
    self.spec = deep.audio_to_mel(wav)



    # self.spec = spectrogram
```

```
class xTSDataset(torch.utils.data.Dataset):
    """ Implementation of the pytorch Dataset. """

    def __init__(self,
                 root: str,
                 type: str,
                 transform: Callable
                 ):
        """ Initializes the xTSDataset
        Args:
            root (string): Path to the root data directory.
            type (string): name of the txt file containing the data split
        """
        self.root = root
        self.transform = transform
        path = os.path.join(self.root, "src", type + ".txt")
        with open(path, 'r') as f:
            content = f.read()

        self.folder = []
        self.file = []
        res = content.split()
        i = 0
        for idx in res:
            if i % 2 == 0:
                self.file.append(idx)
            if i % 2 == 1:
                self.folder.append(idx)
            i = i + 1
        self.size = len(self.file)

    def __len__(self):
        return self.size

    def __getitem__(self, idx: int):
        if idx >= self.size:
            raise IndexError
        stream = xTSSample(self.root, self.folder[idx], self.file[idx])
        stream.load()
        stream.data = stream.data.type(torch.FloatTensor)
        stream.data = stream.data.cuda()
        D, C, H, W = stream.data.shape

        for i in range(C):
              stream.data[:, i, :, :] = self.transform(stream.data[:, i, :, :])

        return stream.data[0,:,:,:], stream.spec
```

Anexa E – Flowtron.py

```
class Flowtron(torch.nn.Module):
    def __init__(self,
                 n_attn_channels=128, n_lstm_layers=2,
                 use_gate_layer=False):
        super(Flowtron, self).__init__()
        norm_fn = nn.InstanceNorm1d
        self.lstm = nn.LSTM(512,
                            int(512 / 2), 1,
                            batch_first=True, bidirectional=True)
        self.speaker_embedding = torch.nn.Embedding(1, 1)
        n_flows = 2
        K = 64
        D_gru = 128
        n_mel_channels = 80
        n_speaker_dim = 0
        n_hidden = 128
        kwargs = dict(kernel_size=3, stride=1, padding=1, bias=True)
        self.conv1 = nn.Conv3d(1, K, **kwargs)
```

```
        self.flows = torch.nn.ModuleList()
        self.dummy_speaker_embedding = True
        self.encoder = resnet18()
        self.encoder.conv1 = nn.Conv2d(
            K,
            64,
            kernel_size=(7, 7),
            stride=(2, 2),
            padding=(3, 3),
            bias=False,
        )
        self.gru = nn.GRU(512,  int(128 / 2), 1,
                          batch_first=True, bidirectional=True)
        self.encoder = nn.Sequential(*list(self.encoder.children())[:-1])
        self.batchnorm = nn.BatchNorm3d(K)
        self.relu = nn.LeakyReLU(inplace=True)
        self.dropout2 = nn.Dropout(0.25)
        for i in range(n_flows):
            add_gate = True if (i == (n_flows-1) and use_gate_layer) else False
            if i % 2 == 0:
                self.flows.append(AR_Step(n_mel_channels, n_speaker_dim,
                                          D_gru,
                                          n_mel_channels+n_speaker_dim,
                                          n_hidden, n_attn_channels,
                                          n_lstm_layers, add_gate))
            else:
                self.flows.append(AR_Back_Step(n_mel_channels, n_speaker_dim,
                                               D_gru,
                                               n_mel_channels+n_speaker_dim,
                                               n_hidden, n_attn_channels,
                                               n_lstm_layers, add_gate))
        self.batch_sizes = 8

    def forward(self, mel, x):

        mel = mel.permute(1, 0, 2)


        try:
            B, D, S, H, W = x.shape
        except:
            B, S, H, W = x.shape
            x = x.unsqueeze(1)

        x = x.float()  # scale data in [0, 1]

        x = self.conv1(x)  # B x C x S x H x W
        x = self.batchnorm(x)
        x = self.relu(x)

        x = self.dropout2(x)
        x = x.transpose(1, 2)
        x = x.reshape(B*S, 64, H, W)
        x = self.encoder(x)

        x = x.squeeze().view(B, S, 512)

        encoder_outputs, _ = self.gru(x)

        encoder_outputs = encoder_outputs.transpose(0, 1)



        log_s_list = []

        mask = None

        for i, flow in enumerate(self.flows):
            mel, log_s= flow(
                mel, encoder_outputs, mask, torch.from_numpy(np.repeat(225, 8)))
            log_s_list.append(log_s)
        gate = None
```

```
        return mel, log_s_list,  gate,  mean, log_var, prob

    def infer(self, residual, x, temperature=1.0,
              gate_threshold=0.5):


        residual = residual.permute(2, 0, 1)
        try:
            B, D, S, H, W = x.shape
        except:
            B, S, H, W = x.shape
            x = x.unsqueeze(1)

        x = x.float()  # scale data in [0, 1]

        x = self.conv1(x)  # B x C x S x H x W
        x = self.batchnorm(x)
        x = self.relu(x)

        x = self.dropout2(x)
        x = x.transpose(1, 2)
        x = x.reshape(B * S, 64, H, W)
        x = self.encoder(x)

        x = x.squeeze().view(B, S, 512)

        encoder_outputs, _ = self.gru(x)

        encoder_outputs = encoder_outputs.transpose(0, 1)


        for i, flow in enumerate(reversed(self.flows)):
            self.set_temperature_and_gate(flow, temperature, gate_threshold)
            residual= flow.infer(residual, encoder_outputs)


        return residual.permute(1, 2, 0)
```

Anexa F – Flowtron_train.py

```
import argparse
import os
import os.path
import pdb
import sys
import numpy as np

import torch
import torch.nn as nn
import torch.optim.lr_scheduler as lr_scheduler
import torch.utils.data
from torch.nn import functional as F
import ignite.engine as engine
import ignite.handlers
import torchvision
from models.Flowtron import FlowtronLoss
import src.dataset
from models.Flowtron import Flowtron
from models import MODELS

ROOT = os.environ.get("ROOT", "")

SEED = 1337
MAX_EPOCHS = 128
PATIENCE = 8
LR_REDUCE_PARAMS = {
    "factor": 0.2,
    "patience": 4,
}

def save_checkpoint(model, optimizer, learning_rate, iteration, filepath):
```

```
    print("Saving model and optimizer state at iteration {} to {}".format(
          iteration, filepath))
    model_for_saving = Flowtron().cuda()
    model_for_saving.load_state_dict(model.state_dict())
    torch.save({'model': model_for_saving,
                'iteration': iteration,
                'optimizer': optimizer.state_dict(),
                'learning_rate': learning_rate}, filepath)


def collate_fn(batches):
    videos = [batch[0] for batch in batches]
    spects = [batch[1] for batch in batches]

    max_v = max(video.shape[0] for video in videos)
    max_s = max(spect.shape[1] for spect in spects)
    videos = [F.pad(video, pad=(0, 0, 0, 0, 0, max_v - video.shape[0])) for video in videos]
    spects = [F.pad(spect, pad=(0, max_s - spect.shape[1], 0, 0)) for spect in spects]
    video = torch.stack(videos)
    spect = torch.stack(spects)
    return (spect,video)


def compute_validation_loss(model, criterion, collate_fn):
    model.eval()
    with torch.no_grad():
        mu = [1.4024e+02, -4.9944e-03, 3.5836e-04]
        sigma = [21.9047, 4.4827, 5.8206]
        transform = torchvision.transforms.Normalize(mean=mu, std=sigma)
        valid_dataset = src.dataset.xTSDataset(ROOT, "valid", transform=transform)
        valid_loader = torch.utils.data.DataLoader(
            valid_dataset,
            batch_size=8,
            collate_fn=collate_fn,
            shuffle=False)
        val_loss = 0.0
        for i, batch in enumerate(valid_loader):
            mel, video = batch
            mel, video = mel.cuda(), video.cuda()

            z, log_s_list, gate_pred,  mean, log_var, prob = model(
                mel, video)

            loss = criterion((z, log_s_list, gate_pred, mean, log_var, prob),
                             np.repeat(225,8))

            reduced_val_loss = loss.item()
            val_loss += reduced_val_loss
        val_loss = val_loss / (i + 1)

    print("Mean {}\nLogVar {}\nProb {}".format(mean, log_var, prob))
    model.train()
    return val_loss

def main():
    parser = argparse.ArgumentParser(description="Evaluate a given model")
    parser.add_argument("--model-type",
                        type=str,
                        required=True,
                        choices=MODELS,
                        help="which model type to train")
    parser.add_argument("-m",
                        "--model",
                        type=str,
                        default=None,
                        required=False,
                        help="path to model to load")
    parser.add_argument("-v",
                        "--verbose",
                        action="count",
                        help="verbosity level")
    args = parser.parse_args()
```

```
mu = [1.4024e+02, -4.9944e-03,  3.5836e-04]
sigma = [21.9047,  4.4827,  5.8206]
mu1 = [1.4024e+02]
sigma1 = [21.9047]
transform = torchvision.transforms.Normalize(mean=mu, std=sigma)
print(args)
train_loss = []
valid_loss = []
model = MODELS[args.model_type]()
if args.model is not None:
    model_path = args.model
    model_name = os.path.basename(args.model)
    model.load(model_path)
else:
    model_name = f"{args.model_type}"
    model_path = f"output/models/{model_name}.pth"
model = model.cuda()
train_dataset = src.dataset.xTSDataset(ROOT, "train2", transform=transform)
valid_dataset = src.dataset.xTSDataset(ROOT, "valid2", transform=transform)

train_loader = torch.utils.data.DataLoader(
    train_dataset,
    batch_size=8,
    collate_fn=collate_fn,
    shuffle=True)

valid_loader = torch.utils.data.DataLoader(
    valid_dataset,
    batch_size=8,
    collate_fn=collate_fn,
    shuffle=False)

# ignite_train = DataLoader(train_loader, shuffle=True)

optimizer = torch.optim.Adam(model.parameters(), lr=0.0001)
criterion = FlowtronLoss()

device = 'cuda'
iteration = 0
model.train()
epoch_offset = max(0, int(iteration / len(train_loader)))
# ================ MAIN TRAINNIG LOOP! ===================
for epoch in range(epoch_offset, 40):
    print("Epoch: {}".format(epoch))
    for batch in train_loader:
        model.zero_grad()
        mel, video = batch
        mel, video = mel.cuda(), video.cuda()
        z, log_s_list, gate_pred,  mean, log_var, prob = model(
            mel, video)
        loss = criterion.forward((z, log_s_list, gate_pred, mean, log_var, prob),
                                 lengths=torch.from_numpy(np.repeat(225,8)).cuda())
        reduced_loss = loss.item()
        loss.backward()
        optimizer.step()
        print("{}:\t{:.9f}".format(iteration, reduced_loss), flush=True)
        iters_per_checkpoint = 113
        if (iteration % iters_per_checkpoint == 0):
            val_loss = compute_validation_loss(
                model=model, criterion=criterion,  collate_fn=collate_fn)
            print("Validation loss {}: {:9f}  ".format(iteration, val_loss))


            checkpoint_path = "{}/model_{}".format(
                    "output/models/checkpoints", iteration)
            save_checkpoint(model, optimizer, 0.0001, iteration,
                            checkpoint_path)
        iteration += 1
```

Anexa G – Flowtron_infer.py

```
import matplotlib
matplotlib.use("Agg")
import matplotlib.pylab as plt

import os
import argparse
import json
import sys
import torch
import numpy as np
import torch
import torch.nn as nn
import torch.optim.lr_scheduler as lr_scheduler
import torch.utils.data
from tqdm import tqdm
import torchvision
from models import MODELS
import src.dataset
from train import collate_fn
from audio import DeepConvTTS
from models.Flowtron import Flowtron
from torch.utils.data import DataLoader

import pdb
ROOT = os.environ.get("ROOT", "")
from scipy.io.wavfile import write
BATCH_SIZE = 8
DEVICE = "cuda"


def infer(flowtron_path,  output_dir,  n_frames=75, seed=1234):
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)

    mu = [1.4024e+02, -4.9944e-03, 3.5836e-04]
    sigma = [21.9047, 4.4827, 5.8206]
    transform = torchvision.transforms.Normalize(mean=mu, std=sigma)

    model_path = f"output/models/checkpoints/model_3503"

    model = torch.load(model_path)["model"]

    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

    model = model.to(device)
    model.eval()
    print("Loaded checkpoint '{}')" .format(flowtron_path))


    dataset = src.dataset.xTSDataset(ROOT, "test", transform=transform)
    loader = torch.utils.data.DataLoader(
        dataset, batch_size=BATCH_SIZE, collate_fn=collate_fn, shuffle=False
    )

    video = torch.from_numpy(np.empty((8, 75, 64, 64)))
    video[0] = dataset[0][0]
    video[1] = dataset[1][0]
    video[2] = dataset[2][0]
    video[3] = dataset[3][0]
    video[4] = dataset[4][0]
    video[5] = dataset[5][0]
    video[6] = dataset[6][0]
    video[7] = dataset[7][0]
    """
    spec = dataset[0][1].to(device)
    video = video.to(device)
    spec = spec.unsqueeze(0)
    model.zero_grad()
```

```
z, log_s_list, gate_pred, attn, mean, log_var, prob = model.forward(
    spec, video[0].unsqueeze(0))

mel, att = model.infer(z.transpose(0,2).transpose(0,1), video[0].unsqueeze(0))
"""

with torch.no_grad():
    residual = torch.from_numpy(np.float32(np.zeros((1,80,225)))).to(device)

    video = video.to(device)
    mels = model.infer(
        residual,  video[0].unsqueeze(0))

mels = mels.squeeze()

mels = mels.transpose(1, 0).cpu().numpy()
deep = DeepConvTTS(sampling_rate=19300)
audio = deep.mel_to_audio(mels)
audio *= 32767 / max(0.01, np.max(np.abs(audio)))
write(os.path.join('results/', 'sigma{}.wav'.format( sigma)),
      19300, audio.astype(np.int16))
```