VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

NÍZKO-DIMENZIONÁLNÍ FAKTORIZACE PRO "END-TO-END" ŘEČOVÉ SYSTÉMY

LOW-DIMENSIONAL MATRIX FACTORIZATION IN END-TO-END SPEECH RECOGNITION SYSTEMS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. MATÚŠ GAJDÁR**
AUTHOR

VEDOUCÍ PRÁCE **Ing. MARTIN KARAFIÁT, Ph.D.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2019/2020

Zadání diplomové práce

Student: **Gajdár Matúš, Bc.**
Program: Informační technologie Obor: Počítačová grafika a multimédia
Název: **Nízko-dimenzionální faktorizace pro "End-To-End" řečové systémy**
Low-Dimensional Matrix Factorization in End-To-End Speech Recognition Systems
Kategorie: Zpracování řeči a přirozeného jazyka

Zadání:

1. Seznamte se s problematikou neuronových sítí (NN) a s jejich použitím v oblasti automatického přepisu řeči.
2. Prostudujte End-To-End přístup a techniku nízko dimenzionální maticové faktorizace.
3. Prostudujte toolkity "kaldi", "espnet" a knihovnu "pytorch" umožňující trénovaní/testování neuronových sítí.
4. Implementujte nízko-dimenzionální maticovou faktorizaci v pytorchi a porovnejte výsledky se stávající Kaldi implementaci.
5. Implementujte tuto techniku do "End-To-End" systému.
6. Otestujte různé architektury.

Literatura:

- Geoffrey Hinton, Li Deng, Dong Yu... **Deep Neural Networks for Acoustic Modeling in Speech Recognition**, IEEE Signal Processing Magazine (2012)
- LeCun, Y., Bengio, Y. and Hinton, G. E. (2015), **Deep Learning,** Nature, Vol. 521, pp 436-444.
- Povey, D., Cheng, G., Wang, Y., Li, K., Xu, H., Yarmohammadi, M., Khudanpur, S. (2018) **Semi-Orthogonal Low-Rank Matrix Factorization for Deep Neural Networks**. Proc. Interspeech 2018, 3743-3747, DOI: 10.21437/Interspeech.2018-1417.
- Shinji Watanabe, Takaaki Hori, Suyoun Kim, John R. Hershey and Tomoki Hayashi, "**Hybrid CTC/Attention Architecture for End-to-End Speech Recognition**," *IEEE Journal of Selected Topics in Signal Processing*, vol. 11, no. 8, pp. 1240-1253, Dec. 2017
- https://espnet.github.io/espnet/tutorial.html

Při obhajobě semestrální části projektu je požadováno:

- Splnění prvních 4 bodů zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Karafiát Martin, Ing., Ph.D.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 20. května 2020
Datum schválení: 1. listopadu 2019

Abstrakt

Práca sa zaoberá problematikou rozpoznávania reči s pomocou učenia neurónových sietí, na ktoré je aplikovaný algoritmus nízko-dimenzionálnej faktorizácie. V práci je popísaná implementácia časovo oneskorených neurónových sietí s faktorizáciou (TDNN-F) a bez nej (TDNN) v jazyku Pytorch. Následne je porovnávaná s už existujúcou implementáciou v nástroji Kaldi, kde boli dosiahnuté podobné výsledky v rámci experimentovania s rôznymi architektúrami. V poslednej kapitole popisujeme dopad nízko-dimenzionálnej faktorizácie na 'End-to-End' (E2E) rečové systémy a taktiež modifikovanie systému s TDNN(-F) sieťami. Pri experimentoch sa nám v určitých nastaveniach sietí s faktorizáciou podarilo zlepšiť výsledky.Súčasne sme pomocou TDNN(-F) sietí dokázali zmenšiť komplexnosť učenia redukciou veľkosti siete.

Abstract

The project covers automatic speech recognition with neural network training using low-dimensional matrix factorization. We are describing time delay neural networks with factorization (TDNN-F) and without it (TDNN) in Pytorch language. We are comparing the implementation between Pytorch and Kaldi toolkit, where we achieve similar results during experiments with various network architectures. The last chapter describes the impact of a low-dimensional matrix factorization on End-to-End speech recognition systems and also a modification of the system with TDNN(-F) networks. Using specific network settings, we were able to achieve better results with systems using factorization. Additionally, we reduced the complexity of training by decreasing network parameters with the use of TDNN(-F) networks.

Kľúčové slová

Automatické rozpoznávanie reči, konvolučné neurónové siete, TDNN, nízko-dimenzionálna faktorizácia, E2E, TDNN-F, Pytorch, Kaldi, ESPnet

Keywords

Automatic speech recognition, convolution neural networks, TDNN, low-dimensional matrix factorization, E2E, TDNN-F, Pytorch, Kaldi, ESPnet

Citácia

GAJDÁR, Matúš. *Nízko-dimenzionální faktorizace pro "End-To-End" řečové systémy.* Brno, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Martin Karafiát, Ph.D.

Nízko-dimenzionální faktorizace pro "End-To-End" řečové systémy

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pána Ing. Martina Karafiáta, Ph.D. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Matúš Gajdár
1. júna 2020

Poďakovanie

Chcel by som sa poďakovať vedúcemu práce pánovi Ing. Martinovi Karafiátovi, Ph.D. za jeho pomoc a poskytnutie užitočných rád. Táto práca vznikla za podpory projektu CERIT Scientific Cloud, reg.č.: CZ.02.1.01/0.0/0.0/16_013/0001802 financovaného z EFRR.

Obsah

Kapitola 1

Úvod

Pokroky dnešnej doby v oblasti výpočtovej techniky a algoritmizácie vytvorili priestor pre aplikáciu neurónových sietí do rôznych oblastí. Jednou z nich je prezentovaná problematika rozpoznávania reči. Neurónové siete sú oproti starším modelom na rozpoznávanie reči, ako napríklad modely založené na skrytých markovových modeloch (HMM) [5], väčšinou rýchlejšie a efektívnejšie. Ich aplikácia sa postupne začleňovala do jednotlivých komponentov v architektúre automatického rozpoznávania reči (ASR), ako sú napríklad akustický a jazykový model. Na trénovanie modelov sa používajú neurónové siete s časovými závislosťami, ktoré si dokážu zapamätať súvislosti a extrahovať potrebné informácie zo vstupnej sekvencie dát. Ich všeobecný názov je rekurentné neurónové siete (recurrent neural networks - RNN) [40]. Do tejto kategórie ale spadá niekoľko typov architektúr, ktoré však majú tú nevýhodu, že trénovanie je časovo náročné.

V tretej kapitole si preto predstavíme neurónové siete s časovým obmedzením (time delay neural networks - TDNN) odvodené od konvolučných neurónových sietí, ktoré sa vyznačujú časovo menej náročným trénovaním. Následne ukážeme princíp výpočtu nízko-dimenzionálnej faktorizácie v štvrtej kapitole. V ďalšej kapitole otestujeme vytvorené TDNN siete s faktorizáciou (TDNN-F) v jazyku *Pytorch*[1] [33] a dosiahnuté výsledky porovnáme s existujúcim riešením v nástroji *Kaldi*[2] [37].

V šiestej kapitole popíšeme princíp 'End-to-End' (E2E) [7] systémov, ktoré používajú k učeniu hybridný mechanizmus. Pozostávajú z dvoch hlavných častí nazývaných enkodér a dekodér. Práve na enkodér aplikujeme nízko-dimenzionálnu faktorizáciu a otestujeme jej vplyv na celkové výsledky trénovania. Následne sa pokúsime nahradiť časti celého systému implementovanou TDNN a TDNN-F (TDNN(-F)) sieťou.

V úvodnej časti si najprv priblížime základné pojmy [10], ktoré je potrebné vedieť pre pochopenie problematiky v zvyšku práce.

1.1 Súvisiace práce

Ideu neurónových sietí s časovým obmedzením, ktoré vychádzajú z konvolučných neurónových sietí predstavili už v roku 1989 [47]. Redukciu počtu prepojení v týchto sieťach predstavili v práci [34]. Tento typ neurónových sietí je na trénovanie efektívnejší a časovo nenáročný a zároveň dosahuje podobné výsledky ako predošlé modely. Metóda nízko-dimenzionálnej

[1] nástroj na strojové učenie - https://pytorch.org/

[2] nástroj na rozpoznávanie reči - https://kaldi-asr.org/doc/

faktorizácie pre rozšírenie týchto sietí bola predstavená v [36]. Tento typ siete k svojmu učeniu potrebuje menšie množstvo dát k dosiahnutiu výborných výsledkov.

Vďaka postupnému presunu jednotlivých komponentov ASR na neurónové siete sa zjednodušil proces vytvárania nových systémov na spracovanie reči, čo prispelo k vytvoreniu E2E systémov, kde sa jednotlivé komponenty prepájajú do jednej hlbokej neurónovej siete. Princíp učenia týchto sietí pomocou enkodéru a dekodéru predstavili v [31]. Tento systém bol následne rozšírený o *connectionist temporal classification* (CTC) [14] v práci [15]. E2E systém zo sebou prináša napríklad zjednodušenie prípravy dát pre trénovanie a testovanie, naopak jeho nevýhodou je potreba veľkého množstva dát na trénovanie. K zmenšeniu množstva dát potrebných k natrénovaniu tejto vcelku robustnej neurónovej siete nám poslúži práve nízko-dimenzionálna faktorizácia, predstavená v už spomínanej práci [36].

1.2 Cieľ práce

Hlavným cieľom práce je zredukovať množstvo dát a zefektívniť trénovanie v komplexnom E2E systéme pre trénovanie automatického rozpoznávania reči. K tomu nám práve poslúži nízko-dimenzionálna faktorizácia. Tú aplikujeme na jednu z dvoch hlavných častí systému a to presnejšie na enkodér vystavaný zo sietí typu RNN. V ňom sa už nachádzajú komponenty, na ktoré je vhodné tento princíp použiť. Taktiež celý enkodér nahradíme architektúrou TDNN(-F) sietí, ktoré túto myšlienku faktorizácie predstavili. Jednotlivými experimentami sa budeme snažiť optimalizovať nastavenia pre zlepšenie výsledkov. Očakávame ho najmä pri použití menšieho množstva trénovacích dát. V prípade TDNN-F sietí očakávame aj redukciu počtu učiacich parametrov bez toho, aby sme stratili efektívnosť systému.

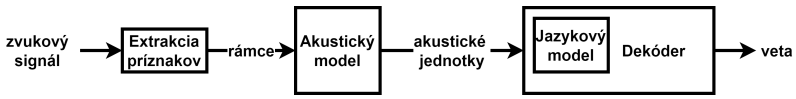
Kapitola 2

Základné pojmy problematiky automatického rozpoznávania reči

K správnemu uchopeniu problematiky si na začiatok predstavíme pár základných pojmov, pretože v problematike rozpoznávania reči sa s nimi budeme často stretávať:

- rámec - parametrizovaná časť zvukového signálu
- fonéma - významotvorná hláska v jazyku
- dekodér - hľadá najlepšiu sekvenciu slov zo vstupných foném

Hlavnou podstatou ASR je premeniť postupnosť rámcov na postupnosť foném a tie následne transformovať pomocou dekodéru na ucelené vety. V nasledujúcich častiach si predstavíme hlavné časti potrebné k tomuto procesu, zobrazené aj na obrázku 2.1. Okrem toho priblížime základné pojmy pre učenie pomocou neurónových sietí.



Obr. 2.1: Schéma základného procesu pre rozpoznávanie reči.

2.1 Základný proces pri rozpoznávaní reči

Rečník, teda osoba, vysloví myšlienku a vytvorí tak zvukový signál. Tým sa vygeneruje istá sekvencia slov W, ktorá prejde komunikačným kanálom. Priestorom sa teda šíri myšlienka v podobe akustického signálu X. Systém, ktorý nazývame rozpoznávač reči (môže ním byť aj osoba) tento signál zachytí, spracuje a následne dekóduje na slovnú sekvenciu W_2, pričom sa snaží o čo najväčšiu podobnosť s pôvodne vyslovenou sekvenciou W .

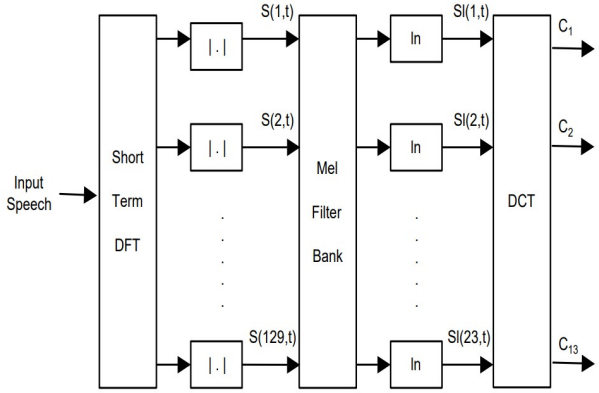
Pravdepodobnosť $P(W_2|X)$, vieme počítať priamo len pomocou E2E systému (bližšie vysvetlené v kapitole 6). Klasické systémy postavené na HMM, danú pravdepodobnosť faktorizujú pomocou Bayesovho pravidla:

$$P(W_2|X) = \frac{P(X|W_2)P(W_2)}{P(X)}, \tag{2.1}$$

kde $P(X|W_2)$ je pravdepodobnosť generovaná akustickým modelom. $P(W_2)$ je pravdepodobnosť generovaná jazykovým modelom a $P(X)$ je pravdepodobnosť akustického signálu, ktoré je konštantná pre celý zvukový signál, môžeme ju teda zanedbať.

Systémový rozpoznávač reči obsahuje niekoľko častí. Jednou z prvých častí je akustický model. Zahŕňa znalosti o zvukových dátach, teda o akustike, fonetike, vplyve prostredia na reč, o pohlaví rečníka, jeho dialekte a mnoho ďalších. Na predspracovanie signálu pre akustický model sa používa extrakcia významných častí signálu. Ďalšia časť v systéme sa nazýva dekodér, ktorý zahŕňa znalosť o správnej sekvencii foném v danom jazyku. Pomocou tohto modelu sa s prispením jazykového modelu z výsledkov na výstupe akustického modelu hľadá najpravdepodobnejšia sekvencia slov.

Všetky komponenty v systéme obsahujú veľa neznámych, ktoré súvisia s jednotlivými vlastnosťami rôznych rečníkov, ako je ich odlišný štýl reči, rýchlosť rozprávania, rôzna slovná zásoba, neznáme slová, rozličné nedokonalosti reči a mnoho iných neznámych, ktoré vplývajú na nedokonalosť systému. Systém, ktorý si poradí s čo najväčším počtom týchto nedostatkov môžeme označiť za *state-of-the-art*[1] systém.



Obr. 2.2: Schéma výpočtu MFCC koeficientov. Prevzaté z [55].

Extrakcia príznakov

Rečový signál je hneď na začiatku systému, a teda pred vstupom do akustického modelu upravený. To znamená, že akustický signál je segmentovaný posuvným oknom typicky na dĺžku 20-25 ms s posunom 10 ms. Na jednotlivé rámce následne aplikujeme operácie v nasledujúcom poradí (viď. obrázok 2.2):

1. diskrétnu fourierovu transformáciu (DFT) na rámec
2. absolútnu hodnotu na výstupné koeficienty DFT, tým získame energiu
3. *mel-scale filterbank*, urobí súčet energií v okolí významných frekvencií
4. logaritmus výstupných hodnôt
5. diskrétna kosínusová transformácia (DCT)

[1] najlepší systém v súčasnosti v danej problematike

Týmto postupom získame z rámca vektor C, zachovávajúci najmä užitočné informácie zo signálu pre následné použitie, taktiež nazývané MFCC koeficienty. Pri trénovaní pomocou neurónových sietí sa už posledná operácia DCT nepoužíva a za koeficienty sa považuje už vektor Sl, ktorého veľkosť je typicky 23. Poznáme rôzne metódy extrakcie príznakov, z ktorých najčastejšie používanými sú spomínaná *MFCC*, *PLP* a *FBank* (hlbšie popísané v knihe [19]). V našom testovaní budeme používať metódy *MFCC* a *FBank*.

Akustický model

Slovo W rozkladáme na menšie akustické jednotky, typicky na fonémy alebo ich kontextovo závislé varianty (bi-phony,tri-phony). Akustický model počíta pravdepodobnosť s akou prislúchajú akustické jednotky k zvukovému rámcu rečového signálu X. Pre určovanie týchto pravdepodobností sa prevažne používali HMM, v prípade hybridných modelov sa stále ešte používajú. V dnešnej dobe sa k tejto problematike pristupuje už prevažne pomocou neurónových sietí. Úlohou akustického modelu $P(X|W)$ je určiť s akou pravdepodobnosťou prislúcha akustická jednotka A k vstupnému rámcu X v čase t.

$$P(X|A) = \prod_{t=1}^{T} p(x_t|a_t), \tag{2.2}$$

Následne na to sa pre slovo W skladajúce z S akustických jednotiek $W = \{A_1, ..., A_S\}$, počíta pravdepodobnosť k signálu X:

$$P(X|W) = \prod_{s=1}^{S} P(X|A_s) \tag{2.3}$$

Dekodér

Dekodér generuje ucelené vety z posteriórnych pravdepodobnostní (u ktorých platí že ich suma je rovná 1) foném z akustického modelu. Je zostavený zo štyroch prevodníkov (hlbšie popísané v [27]), ktoré majú znalosť o kontexte, gramatike a výslovnosti slov.

Súčasťou dekodéru je jazykový model, ktorý pomáha určiť sekvenciu slov bez informácie o zvukových dátach. Na vstupe má k dispozícii postupnosť slov W, z ktorých určuje pravdepodobnosť danej sekvencie, rovnica 2.4. V tejto problematike sa najviac dáva prednosť používaniu N-gramových modelov, kde $N-1$ určuje počet predchádzajúcich slov určených na predpoveď aktuálneho slova.

$$P(W) = \prod_{i=1}^{k} P(w_i|w_{i-1}^{i-n+1}), \tag{2.4}$$

kde k určuje veľkosť predpovedanej sekvencie slov a w označuje jednotlivé slová. Najčastejšie sa používajú tri-gramové modely, ktoré modelujú postupnosť z troch slov. Rovnako ako v prípade akustického modelu sa už aj tento model implementuje ako neurónová sieť [26], ktorá sa učí na jednotlivých slovách zo vstupného korpusu.

Hodnotenie chybovosti

K porovnávaniu jednotlivých natrénovaných modelov bolo potrebné určiť si správnu hodnotiacu metriku. V prípade rozpoznávania reči je najčastejšie používanou metrikou chybovosti

takzvaná *Word Error Rate* (WER), ktorá je odvodená od metriky *edit distance* [29]. Počíta sa v nej s tromi druhmi chýb. Prvým druhom chyby je **substitúcia (S)** slova, pri ktorej je slovo nahradené iným nesprávnym slovom. Ďalším druhom je **zmazanie (Z)** správneho slova. Posledným druhom chyby je **vloženie (V)**, kedy je nesprávne slovo pridané navyše. Percentuálny úspech rozpoznávača vyjadrený ako hodnota WER sa dá vyjadriť ako súčet troch typov chýb podelený **celkovým počtom slov vo vete (All)** .

$$WER = 100\% \times \frac{S + V \ + Z}{All} \tag{2.5}$$

Pri hodnotení sa taktiež môže brať do úvahy aj hodnotenie správnosti celých viet. Každý E2E systém nám na výstupe dáva sekvenciu značiek, ktoré si na začiatku trénovania určíme. Tie môžu označovať jednotlivé písmená, ale aj slová alebo ich časti. Preto budeme na porovnávanie používať značky namiesto slov výpočtom *Character Error Rate* (CER), ktorého výpočet je totožný s WER.

2.2 Základné pojmy pre neurónové siete

V systémoch rozpoznávania reči sa používajú neurónové siete na tvorenie všetkých typov komponentov. Akustický model tvorený neurónovou sieťou dostane na vstup parametrizovaný zvukový rámec a vypočíta vektory posteriórnych pravdepodobností pre jednotlivé triedy. U problematiky rozpoznávania reči týmto myslíme prevažne fonémy. Vďaka univerzálnosti neurónových sietí a ich veľkým potenciálom môžeme povedať, že ich využitie je rozsiahle. Tento model môžeme výhodne použiť aj pri iných problematikách, ako je napríklad rozpoznávanie jazyka, akustické vyhľadávanie kľúčových slov, identifikácia rečníka a pod. Pre učenie neurónovej siete je kľúčovým faktorom príprava trénovacích a testovacích dát, k čomu je potrebné čo najväčšie množstvo kvalitných dát.

Neurónové siete

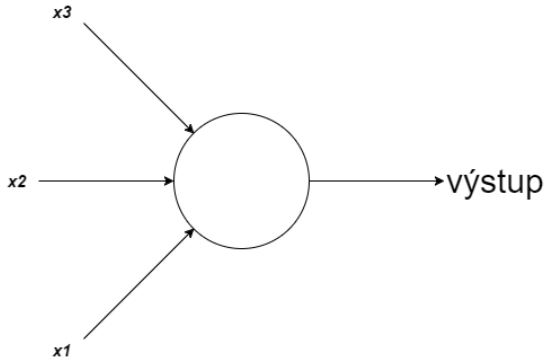
Matematický model neurónových sietí vznikol na základe inšpirácie z biologických neurónových sietí. Neurón je ich základným prvkom. Z veľkého množstva sekvencie neurónov tvoríme jednotlivé vrstvy siete. Tie sa delia na vstupné a výstupné. Často sa medzi nimi nachádza niekoľko skrytých vrstiev, čím vytvoríme hĺbkovú neurónovú sieť. Neuróny spolu komunikujú prostredníctvom posielania signálov cez váhové spojenia, na základe ktorých potom počas trénovania neustále upravujeme ich hodnoty - váhy. Po natrénovaní modelu trénovacími dátami váhy nadobudnú fixné hodnoty.

Neurón

Perceptron na obrázku 2.3 je binárny prvok, z ktorého vznikol neurón. Ten má niekoľko vstupov a vyprodukuje jeden binárny výstup. Každý vstup má pridelené reálne číslo, ktoré hodnotí dôležitosť vstupnej hodnoty, takzvanú váhu.

Threshold je významný parameter v rámci perceptronu. Porovnáva sa so sumou všetkých váh násobených príslušnými vstupmi vchádzajúcimi do perceptronu a určuje hodnotu výstupu: 0 alebo 1.

$$output = \begin{cases} 0 & ak \quad \sum_j w_j x_j \leq threshold \\ 1 & ak \quad \sum_j w_j x_j > threshold \end{cases} \tag{2.6}$$



Obr. 2.3: Príklad vstupného vektoru x do perceptronu. Prevzaté z [10].

Neurón je prvok s výstupom v obore reálnych čísel. Ako už názov napovedá, neurón je základným kameňom neurónových sietí. Výpočet výstupnej hodnoty je takmer totožný *perceptronu*. K súčtu vstupov sa často pripočítava ešte konštanta *bias*, ktorá pomáha vyhnúť sa problému, ak sú na vstupe nulové hodnoty, ktoré spôsobujú vynulovanie váhových spojení. Často tak docielime lepšie výsledky modelu. Ak by sa napríklad na vstup neurónu vložil nulový vektor x , tak by na výstupe bola hodnota nula. Tá by ďalej nevhodne ovplyvňovala ďalšie vrstvy neurónov. Súčet je nakoniec ešte transformovaný vhodnou aktivačnou funkciou *f()* používanou v neurónových sieťach.

$$y = f(\sum_{j=1}^{N} w_j x_j + bias) \tag{2.7}$$

Aktivačné funkcie

K transformácii sa využíva niekoľko aktivačných funkcií. Jednoduchšie funkcie sú používané častejšie pre ich efektívnosť a rýchlosť pri výpočte. Jednou z najznámejších je *logistický sigmoid*. Táto funkcia pri akomkoľvek vstupe nadobúda hodnoty medzi 0 až 1.

Ďalšou často používanou funkciou je *rektifikovaná lineárna jednotka (ReLU)* (obrázok 2.4) najmä vďaka jej jednoduchosti. Je výpočtovo rýchlejšia ako sigmoid a je tou najjednoduchšou nelineárnou funkciou, ktorá sa používa v architektúrach neurónových sietí s dosiahnutím dobrých výsledkov. Pri tejto funkcii získame z každého vstupu rozmedzie hodnôt taktiež medzi 0 až 1.

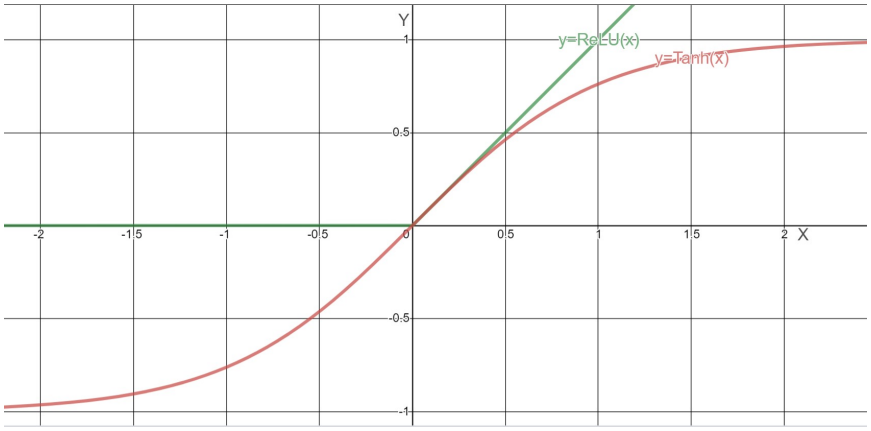
$$\sigma(x) = max(0, x) \tag{2.8}$$

V prípade, že veľa vstupných hodnôt obsahuje záporné hodnoty, mali by sme zvážiť použitie funkcie *hyperbolický tangens* (Tanh) (obrázok 2.4). Je výpočtovo náročnejší ako sigmoid ale na výstupe získame rozmedzie hodnôt medzi -1 až 1.

$$\sigma(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.9}$$

Hodnotiaca funkcia

Na úpravu už spomínaných váh potrebujeme algoritmus, ku ktorému je potrebné zvoliť funkciu, ktorá zhodnotí ako a o koľko by sa mali upraviť jednotlivé váhové parametre. Po-



Obr. 2.4: Porovnanie priebehu aktivačných funkcií. Zelená čiara reprezentuje priebeh funkcie ReLU. Červená čiara reprezentuje priebeh funkcie Tanh.

stupnou aproximáciou sa snažíme aby vstupné dáta x čo najlepšie sedeli do hypotézy $h()$. Pre zhodnotenie ako blízko správnemu nastaveniu váh sme, môžeme použiť jednu z hodnotiacich funkcií:

$$C(w,b) = \frac{1}{n}\sum_{x=1}^{n}[a_x \ln h(x) + (1-a_x)\ln(1-h(x))], \tag{2.10}$$

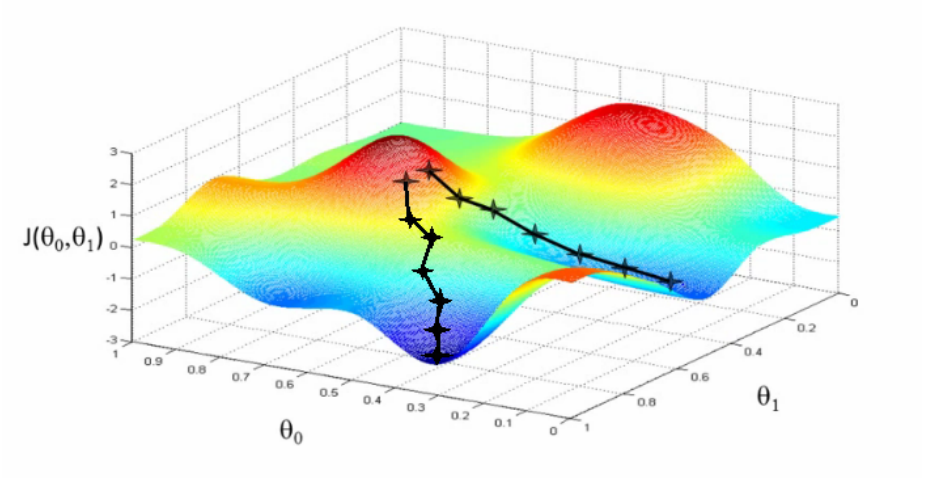
kde váhy neurónovej siete označujeme w , písmenom b zase označujeme všetky upravované biasy. Počet všetkých vstupných trénovacích dát je n . Cieľom každej úpravy je, aby hypotéza s aktuálnym vstupným vektorom $h(x)$ čo najlepšie odpovedala očakávanému výstupnému vektoru a_x. Výsledná hodnota je normalizovaná suma cez všetky trénovacie dáta. Snažíme sa teda dopracovať k tomu, aby $C(w,b) \approx 0$. Funkcia C sa nazýva *cross entropy error* [28].

Trénovanie

Trénovaním neurónovej siete je potrebné nastaviť váhové koeficienty tak, aby čo najlepšie klasifikovali dáta z trénovacej sady. Využíva sa k tomu niekoľko algoritmov, jeden zo základných je *gradient descent* (obrázok 2.5), ktorý hľadá minimum funkcie.

Trénovanie prebieha v niekoľkých krokoch. Najprv je potrebné vybrať sadu trénovacích a validačných dát, bežne v pomere 10 ku 1. V rámci sady sa istý počet dát spája do dávok, takzvaných *batch*. Následne na to je potrebné pre každú dávku urobiť nasledujúce kroky.

Dávky postupne po jednej vkladáme na vstup neurónovej siete. Urobíme dopredný výpočet, čo znamená, že vstupné vektory jednej dávky prejdú jednotlivými funkciami vo všetkých vrstvách siete až po výstupnú vrstvu. Z výstupu dostaneme odhad, ten porovnáme s očakávaným výstupom a vypočítame rozdiel pomocou hodnotiacej funkcie. Podľa toho ako veľmi sa líšil odhad od očakávaného výstupu budeme spätne upravovať jednotlivé váhy v sieti a to tak, že budeme postupovať presne opačným smerom od výstupnej až k vstupnej vrstve.



Obr. 2.5: Gradient descent, čierne úsečky znázorňujú jednotlivé kroky, ktoré sa blížia k minimám funkcie. Prevzatý z [42].

Jednotlivé váhy Θ upravíme nasledovne:

$$\Theta_{new} = \Theta_{old} + \lambda \nabla \frac{C}{\Theta_{old}}, \tag{2.11}$$

kde λ označuje učiaci faktor, ktorý určuje dĺžku kroku. C označuje hodnotiacu funkciu. $\nabla \frac{C}{\Theta}$ označuje výpočet *gradientu* z hodnotiacej funkcie v závislosti na váhach Θ pre jednotlivé vrstvy. Vo výpočte *gradientu* sa používa *chain rule* [13]. Učiaci faktor sa s väčším počtom iterácií zmenšuje. Ak by ostal nezmenený, v posledných krokoch učenia by sme mohli urobiť nesprávny krok výpočtu a preskočiť tak hľadané minimum. Naopak, ak by bol spočiatku veľmi malý, budeme konvergovať veľmi pomaly. K aktualizácii váh sa taktiež používajú rôzne optimalizačné algoritmy *Adam* [22] alebo *Adadelta* [54], ktoré sú založené na princípe *stochastic gradient descent* [39].

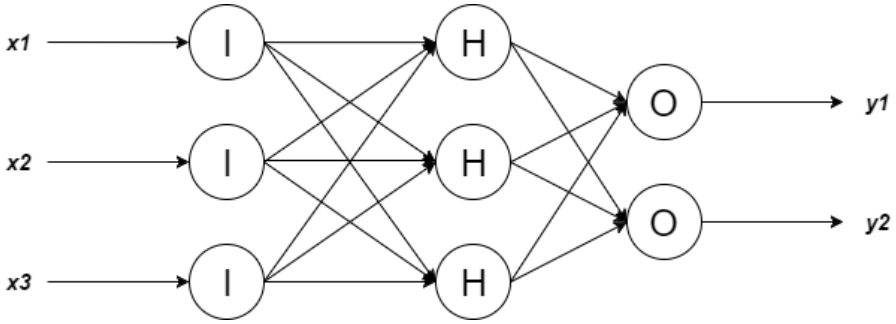
Typy neurónových sietí

Neurónové siete majú niekoľko delení. Priblížime si delenie z pohľadu typu prepojenia neurónov.

Prvou z nich je základná **doprednú neurónová sieť**, zobrazená na obrázku 2.6. V nej sú neuróny usporiadané do niekoľkých vrstiev, a to tak, aby sa neurón z nižšej vrstvy spájal so všetkými neurónmi vrstvy vyššej, nie naopak. Vstupný signál je teda postupne prenášaný na výstup cez skryté vrstvy. Algoritmy v tomto type siete sú jednoduchšie a ľahšie implementovateľné.

V **rekurentnej neurónovej sieti** sú neuróny prepojené v oboch smeroch a taktiež je každý neurón prepojený sám so sebou. Algoritmy sú preto zložitejšie a ťažšie na implementáciu. Taktiež sú pamäťovo a časovo náročnejšie. Grafický nákres takejto siete je neprehľadný, čo naznačuje jeho väčšiu zložitosť oproti klasickej doprednej sieti.

Konvolučná neurónová sieť sa začala používať najmä v oblasti spracovania obrazu. Jej princíp spočíva v násobení vstupnej matice z ďalšou, menšou maticou a následným sčítaním týchto hodnôt. Výsledkom je jedna hodnota, ktorá nám zmenší vstupnú maticu s hodnotami, ktoré budú verne reprezentovať pôvodný vstupný obrázok. Hlavnou časťou



Obr. 2.6: **Vrstvy doprednej neurónovej siete**: symboly **I** označujú vstupnú vrstvu, **H** skrytú vrstvu a **O** výstupnú vrstvu. Vstupný vektor má veľkosť 3 a začína písmenom x. Výstupný vektor veľkosti 2 začína písmenom y. Prevzaté z [10].

algoritmov pre učenie je násobenie matíc. Táto operácia je pri učení rýchla najmä pri použití grafických kariet. Viac detailov je v kapitole 3.1.1.

Kapitola 3

Architektúry neurónových sietí

V tejto časti predstavíme architektúru dvoch typov neurónových sietí. Jednou z nich sú konvolučné neurónové siete (CNN) [23] od ktorých sú odvodené TDNN. Druhou sú siete RNN, ktorých súčasťou je blok *Long-Short Term Memory* (LSTM) [18].

3.1 Popis časovo oneskorených neurónových sietí

V úvode spomínané TDNN (časovo oneskorené neurónové siete) boli predstavené už v roku 1989 [47]. Tieto siete sú v podstate hlboké neurónové siete vyskladané z 1-dimenzionálnych konvolučných vrstiev, ktoré si predstavíme ďalej v texte. Potom si priblížime použitie týchto sietí v rozpoznávaní reči, ktoré bolo vyvinuté z pôvodného použitia sietí pri spracovaní obrazu.

3.1.1 Konvolučné neurónové siete a regularizačné metódy

V tejto časti si priblížime problematiku CNN. Spočiatku boli používané najmä v odboroch pre spracovanie obrazu. S použitím týchto sietí dokážeme vo vstupnom obraze rozlíšiť rôzne typy vzorov a objektov. Dosiahneme to najmä vďaka aplikácií filtrov na jednotlivé časti obrazu, ktorých hodnoty sa sieť postupne učí. Tento prístup sa líši od starších algoritmov tým, že filtre sa upravovali ručne tak, aby našli hľadaný vzor. Všeobecne povedané, sú to v podstate dopredné neurónové siete s pevnou štruktúrou, kde ako váhy rozumieme hodnoty konvolučných filtrov. Aby sieť dosahovala dobré výsledky, musí obsahovať viacero vrstiev, čím získame priestorové, a v prípade spracovania videa, aj časové závislosti.

Konvolučná vrstva

Konvolučná vrstva sa skladá z niekoľkých filtrov, ktoré majú rovnaký rozmer, tiež nazývaných aj konvolučné matice. Daným filtrom prechádzame vstupnú maticu podľa toho, ako si nastavíme jednotlivé parametre danej vrstvy. Tie si priblížime v sekcii 3.1.1.

Výpočet konvolúcie

Pri výpočte hodnôt výstupu z konvolučnej vrstvy postupujeme jednotlivými násobeniami vstupnej matice s filtrom a následným sčítaním. Táto operácia sa volá diskrétna konvolúcia.

$$G[m,n] = (f * h)[m,n] = \sum_j \sum_k h[j,k] f[m-j, n-k], \tag{3.1}$$

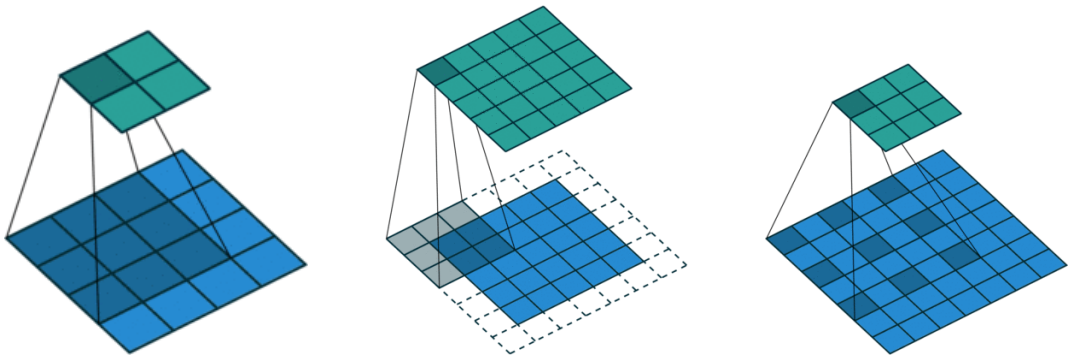
kde vstupná matica je označená ako *f*, matica filtra je označená ako *h*. Parametre *m, n* označujú bod výstupnej matice s rozmermi *M*N*, kde pre jednotlivé hodnoty postupujeme zľava doprava, teda: $n = n+1, n = (0, 1, \ldots, M)$. Pri dosiahnutí hodnoty M sa vrátime na začiatok a posunieme sa zhora dole, pre $m = m+1, m = (0, 1, \ldots, M)$. Úplne rovnako sa postupuje aj v prípade matice filtru s rozmermi *J*K*, kde opäť hodnoty *j* a *k* nadobúdajú postupne hodnoty: $j = (0, 1, \ldots, J)$ a $k = (0, 1, \ldots, K)$.

Jedným krokom výpočtu rozumieme vynásobenie vstupnej matice *f* s filtrom *h* v príslušnom bode *m,n* na výstupnej matici, ako to môžeme vidieť na obrázku 3.1 vľavo. Výpočty konvolúcie môžu prebiehať v rôznych dimenziách. Nami prezentovaný výpočet bol pre výpočet konvolúcie v 2D priestore, teda najbežnejšie používaný pri obrazoch. Pri spracovaní reči používame konvolúcie v 1D priestore.

Parametre konvolúcie

Pri cykle výpočtu konvolúcie môžeme nastaviť rôzne parametre, ktoré ovplyvňujú výpočet výstupnej matice [1]:

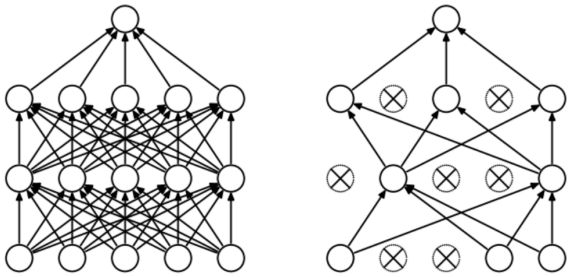
- *padding* - parameter označuje rozšírenie vstupnej matice nulovými alebo náhodnými hodnotami, napríklad za účelom zachovania rozmerov vstupného obrázku. Ukážka na obrázku 3.1 v strede.
- *stride* - parameter označuje o aký krok sa posunieme vo vstupnom obrázku. Základnou hodnotou je 1, to znamená, že sa budeme posúvať o jeden krok vpravo. Na konci riadku sa posunieme na začiatok a zároveň o jeden riadok dole.
- *dilation* - parameter označuje, ktorý krok v poradí máme vypočítať. Vo vzorovom prípade, kde je hodnota nastavená na 2, bude matica počítať z každou druhou hodnotou zo vstupu. Ukážka na obrázku 3.1 vpravo.



Obr. 3.1: Vľavo - klasická konvolúcia. Stred - padding o veľkosti 0. Vpravo - dilation o hodnotu 2. Prevzaté z [1].

Dropout

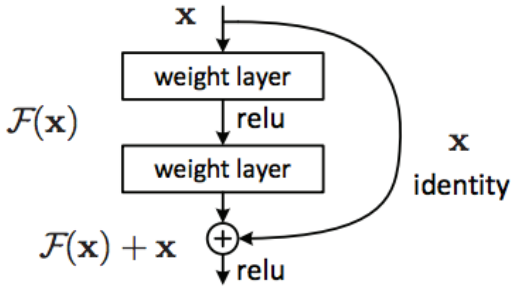
Regularizačná metóda *dropout* [45] sa používa buď pri doprednej alebo spätnej propagácii, kde v jednotlivých vrstvách nebudeme používať v aktuálnom výpočte niektoré uzly. Tie sú vybrané náhodne na základe pravdepodobnosti, ktorú zadávame na vstupe. Zobrazenie tejto metódy je na obrázku 3.2. Táto metóda sa používa najmä pri hlbokých sieťach s viacerými vrstvami. Napomáha to k tomu, aby sme nepretrénovali našu sieť. Pri pretrénovaní sieť takmer bezchybne predpovedá výstupy z trénovacích dát, ale bude dosahovať slabé výsledky pri testovacích dátach, teda tých, ktoré ešte nevidela.



Obr. 3.2: Zobrazenie rozdielu medzi štandardnou sieťou (vľavo) a sieťou, kde použijeme *dropout* metódu (vpravo). Prevzaté z [45].

Skip connections

Princíp *skip connections* [17] sa používa pre regulovanie toku informácií v hlbokých neurónových sieťach z dôvodu možnej straty informácií pri spätnej propagácii, keďže výpočet prechádza veľkým množstvom vrstiev. V bežnej sieti sa prepájajú jednotlivé vrstvy za sebou. Pomocou *skip connections* prepojíme medzi sebou vstupný vektor predchádzajúcej vrstvy s výstupným vektorom z nasledujúcej vrstvy, prípadne s výstupom vyšších vrstiev (viď obrázok 3.3). Následne tieto vektory sčítame alebo ich môžeme aj konkatenovať.



Obr. 3.3: Zobrazenie *skip connections*, kde x je vstupný vektor a $F(x)$ je výstupný vektor danej vrstvy. Na tomto obrázku sa tieto dva vektory na výstupe sčítavajú. Prevzaté z [17].

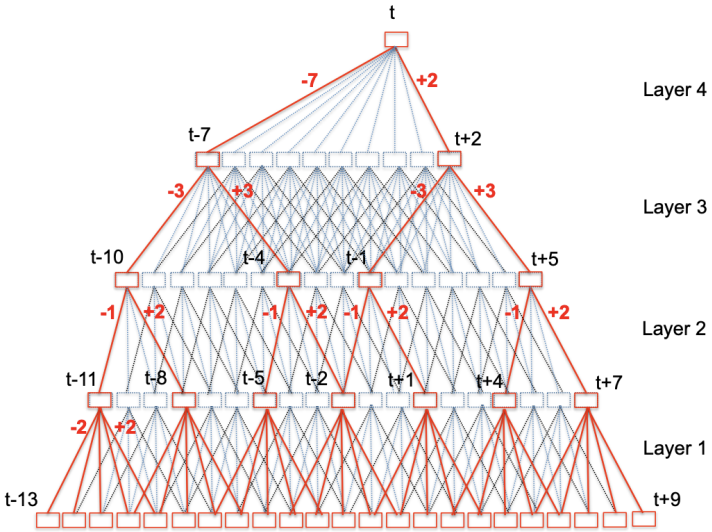
Konvolučné neurónové siete v rozpoznávaní reči

Hlavnou požiadavkou pri trénovaní rozpoznávania reči je udržanie dlhých časových závislostí medzi vstupnými rámcami počas trénovania. Túto potrebu dlho naplňovali rekurentné

neurónové siete, ale ich výpočet je časovo náročný a nedá sa paralelizovať. Jednoduché dopredné neurónové siete by sa mohli zdať efektívnejšie na trénovanie, tie ale pre udržanie časových závislostí potrebujú byť dostatočne veľké na zachytenie dlhých závislostí. S narastajúcou dĺžkou vstupu im narastá počet neurónov a zároveň aj celková náročnosť výpočtu váh v sieti. Preto na výpočet časových závislostí použijeme TDNN. Tieto siete sú efektívnosťou trénovania podobné štandardným dopredným sieťam. Hlavnou výhodou tejto siete je využívanie redukcie prepojení neurónov v sieti pre zrýchlenie výpočtu. K tomu sa používajú 1D konvolučné siete, kde nastavíme potrebné parametre pre jednotlivé vrstvy.

3.1.2 Architektúra časovo oneskorených neurónových sietí

Základná TDNN sieť pozostáva z niekoľkých vrstiev [34]. Na rozdiel od základnej doprednej neurónovej siete neberie prvá vrstva na vstup celý vstupný kontext rámcov. Pri sieti TDNN sú aplikované menšie operácie na užšie kontexty, takzvané posuvné okno. Tieto menšie operácie sú vlastne aplikáciou konvolučného filtra na 1D vstup. Čím hlbšiu sieť máme, tým dlhšie závislosti vieme našu sieť naučiť. Typicky sa používa kontextové okno, ktoré prijíma od aktuálneho rámca viac predchádzajúcich rámcov ako tých budúcich. Sieť je po inicializácii nemenná, je preto dôležité správne nastaviť počet vrstiev.



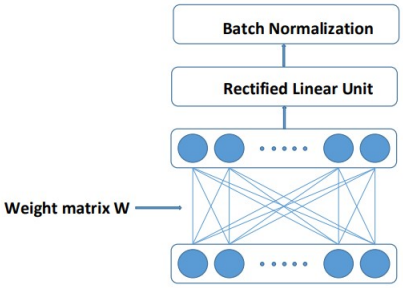
Obr. 3.4: Diagram zobrazuje jednotlivé vrstvy TDNN siete a medzery medzi rámcami, ktoré sú posielané do nasledujúcich vrstiev. Prevzaté z [34].

Jedným z hlavných parametrov pri budovaní TDNN vrstiev je veľkosť vstupného kontextu pre výpočet výstupných hodnôt pre ďalšiu vrstvu. Vstupné kontexty môžu byť definované ako postupnosť čísel alebo ako interval. Ako môžeme vidieť na obrázku 3.4, prvá vrstva berie na vstup kontext o veľkosti $[-2, 2]$, to znamená že berie 5 rámcov zo vstupnej vrstvy na výpočet vektoru pre ďalšiu vrstvu, čo môže byť taktiež zapísané ako $\{-2, -1, 0, 1, 2\}$. Nasledujúca vrstva berie už kontext veľkosti 2, teda rámce: $\{-1, 2\}$. Ako vstup je prvý rámec oneskorený o 1 a druhá hodnota určuje rámec posunutý od aktuálneho o 2 rámce. Tento princíp nám dovoľuje medzi jednotlivými vstupnými rámcami robiť medzery, vďaka čomu dosiahneme redukciu siete a rýchlejší výpočet. V prípade počítania bez medzier by

sme spôsobili zbytočné prekrývania, ktoré by mali za následok redundanciu výpočtov pre rovnaké rámce.

V TDNN sieti je najlepšie, aby prvá vrstva brala kontext bez medzier a každá nasledujúca vrstva kontext s väčšími medzerami. Na vstup sieti na obrázku 3.4 vyskladanej z blokov TDNN dávame zvukové rámce v rozpätí kontextu, ktorý sa nám zmestí do zobrazenej siete. To znamená, že k aktuálnemu rámcu v čase t musíme vyselektovať 13 predchádzajúcich rámcov od času t a 9 nasledujúcich rámcov. Následne urobíme dopredný výpočet cez všetky vrstvy a na výstupe dostaneme vektor s pravdepodobnosťami, ktorý porovnáme so správnym výstupom z trénovacích dát. Urobíme výpočet hodnotiacej funkcie a spätne spropagujeme vypočítanú chybu. V okrajových prípadoch $t = 0$ môžeme buď doplniť vstupný kontext nulovými rámcami zľava alebo skopírovať prvý rámec v sekvencii. V prípade $t = T$ urobíme rozširujúce operácie naopak.

Každý výstupný vektor z TDNN vrstvy prechádza nelineárnou funkciou $ReLU$ a následne je na výstup použitá dávková normalizácia [20]. Normalizácia slúži na zrýchlenie výpočtu a k celkovej stabilite učenia. Na koniec vrstvy pripájame ešte regularizačnú metódu *dropout*. Tento model je zobrazený na obrázku 3.5.



Obr. 3.5: Štandardná TDNN vrstva s aktivačnou funkciou a normalizáciou. Prevzaté z [36].

3.2 Long-short term memory

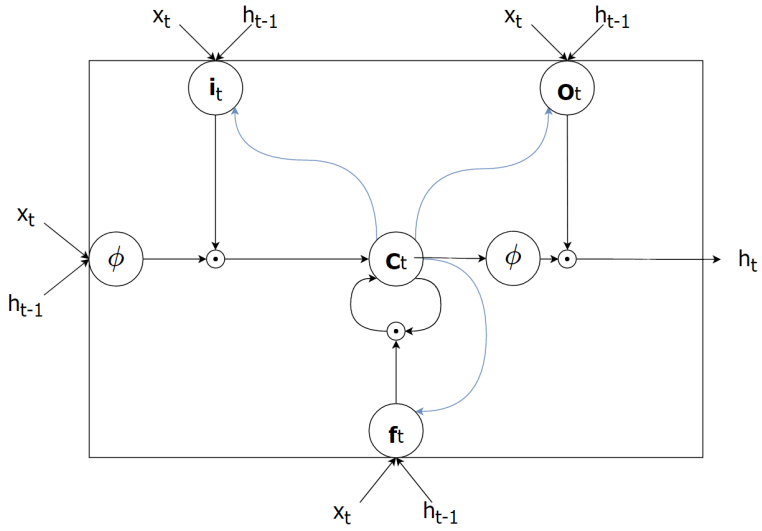
LSTM prvok je rozšírením klasických RNN. V prípade RNN sa sieť dokáže učiť časové závislosti zo vstupných sekvencií, pretože si udržuje vnútorný stav, ktorý reprezentuje už videné hodnoty. Narozdiel od klasických dopredných neurónových sietí, ktoré závisia len na aktuálnych hodnotách na vstupe. Problém v RNN ale nastáva pri dlhších časových závislostiach. Pri počítaní spätnej propagácie strácame hodnotu *gradientu*, čo môže spôsobiť problémy s aktualizáciou váh v nižších vrstvách siete, presnejšie dochádza k *vanishing gradientu* [12]. Riešením tohto problému je práve prvok LSTM. Na udržanie informácie o stave prvku používa pamäťovú bunku **c**. Informačný tok, ktorý ovplyvňuje túto bunku, kontroluje vstupná brána **i**, výstupná brána **o** a zabúdacia brána **f** (viď. obrázok 3.6).

Výpočet jedného prechodu v čase t je popísaný ako:

$$\begin{aligned} i_t &= \phi(W_{ix}x_t + W_{ih}h_{t-1} + W_{ic}c_{t-1} + b_i) \\ f_t &= \phi(W_{fx}x_t + W_{fh}h_{t-1} + W_{fc}c_{t-1} + b_f) \\ c_t &= f_t \odot c_{t-1} + i_t \odot \sigma(W_{cx}x_t + W_{ch}h_{t-1} + b_c) \\ o_t &= \phi(W_{ox}x_t + W_{oh}h_{t-1} + W_{oc}c_t + b_o) \\ h_t &= o_t \odot \sigma(c_t), \end{aligned} \tag{3.2}$$

kde znaky ϕ a σ označujú nelineárne aktivačné funkcie. Znak $\mathbf{b}_{\cdot}$ označuje *bias* použitý na regulovanie váhových spojení. Váhové matice $\mathbf{W}_{\cdot\mathbf{h}}$ spájajú predchádzajúcu skrytú vrstvu s LSTM prvkom, matice $\mathbf{W}_{\cdot\mathbf{x}}$ spájajú vstupy s LSTM prvkom a $\mathbf{W}_{\cdot\mathbf{c}}$ matice sú špeciálne diagonálne prepojenia vychádzajúce zvnútra prvku, teda z pamäťovej bunky, a sú dané na vstup bránam $\mathbf{i_t}$, $\mathbf{o_t}$, $\mathbf{f_t}$ v danom LSTM prvku.

Jednosmerné LSTM bloky príjmajú vstup postupne od času $t = 1$ do T. V prípade obojsmerných LSTM sa používa ďalšia vrstva LSTM blokov, ktoré prechádzajú vstup od $t = T$ do 1. Je preto potrebné mať na vstupe celú vstupnú sekvenciu. Po prechode vstupu obomi smermi sa výsledná hodnota určuje prevažne konkatenáciou výstupných hodnôt h_t z oboch LSTM blokov, na ktoré je aplikovaná aktivačná funkcia. Ďalšou možnosťou je použiť lineárnu vrstvu. Na jej vstup vložíme konkatenované výstupy blokov LSTM a ako výstup následne dostaneme vektor o nastavenej dĺžke. Ak je vstupný vektor väčší ako výstupný, hovoríme o projekčnej vrstve, viac v časti 6.2).



Obr. 3.6: Prvok LSTM obsahuje jednu pamäťovú bunku obsahuje 4 vstupné brány, ktorých vstupom je aktuálny vstupný vektor x_t a stav siete v čase *t-1* označený h_{t-1}. Vo vnútri bunky sa nachádza prvok c_t, ktorý udržiava vnútorný stav bunky. Jeho hodnota ovplyvňuje hodnotu vstupov, výstupov, ale taktiež aj seba samu. Prevzaté z [10].

Kapitola 4

Výpočet nízko-dimenzionálnej faktorizácie

Nízko-dimenzionálna faktorizácia [36] je založená na princípe singulárneho rozkladu a hľadaní semi-ortogonality. Tá nám pomáha pri odstránení redundancie v dátach, respektíve redundantných vektorov v matici.

Semi-ortogonálne matice sú neštvorcové matice s rozmerom $m * n$, kde $m! = n$ a zároveň vektory v riadku, respektíve v stĺpci, sú ortonormálne. Ortonormálne vektory sú zároveň lineárne nezávislé, čo nám pomáha práve pri odstránení redundancie dát pri učení. Najjednoduchšiu semi-ortogonálnu maticu si môžeme predstaviť ako maticu s unikátnymi jednotkovými vektormi v riadku.

Aby bola váhová matica semi-ortogonálna, potrebujeme ju počas učenia postupne upravovať. Výpočet tejto úpravy vykonávame po približne štyroch iteráciách učiaceho algoritmu *SGD* (2.2) a každým výpočtom ju priblížime k tomu aby bola semi-ortogonálna.

Predpokladajme maticu M u ktorej platí, že má menší počet riadkov ako stĺpcov. Túto maticu chceme po jej vynásobení s transponovanou maticou M , teda M^T, čo najviac priblížiť jednotkovej matici a splniť podmienku $P = I$, kde:

$$P \equiv MM^T \tag{4.1}$$

K tomuto účelu si zadefinujeme rovnicu $Q \equiv P - I$. Odvodíme si funkciu $f = tr(QQ^T)$, ktorej minimum budeme hľadať. Operácia $tr()$ znamená suma elementov vstupnej matice na druhú. Po výpočte parciálnych derivácií ako v [36] pre zistenie minimalizačnej funkcie dospejeme k rovnici:

$$M \leftarrow M - 4vQM \tag{4.2}$$

Pre učiaci faktor $v = \frac{1}{8}$ po expandovaní dostávame rovnicu na aktualizovanie matice $M \leftarrow M - \frac{1}{2}(MM^T - I)M$.

Učiaci faktor v sa v našich implementačných výpočtoch označuje ako α. Každou iteráciou túto hodnotu upravíme na základe aktuálnej matice P:

$$\alpha = \sqrt{\frac{tr(PP^T)}{tr(P)}} \tag{4.3}$$

Na kontrolu správneho konvergovania matice používame rovnicu 4.4 na výpočet *Frobenius norm*[1].

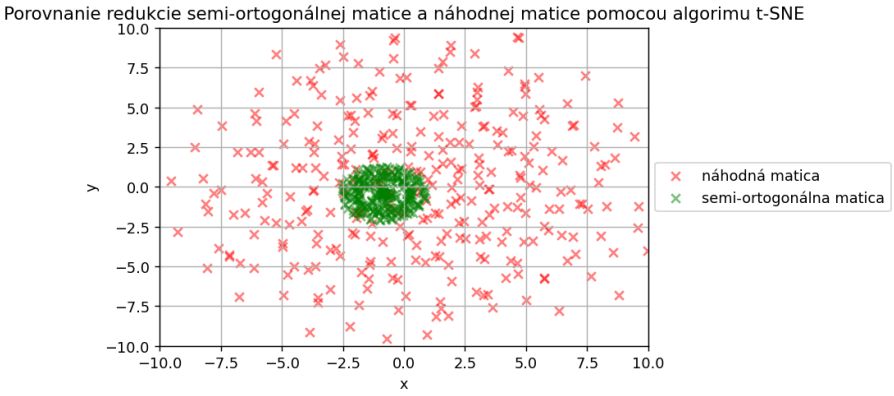
$$\| A \|_F = \sqrt{Tr(AA^T)} \tag{4.4}$$

[1] wolfram definícia Frobenius norm - http://mathworld.wolfram.com/FrobeniusNorm.html

Táto rovnica nám pri každej iterácii v trénovacom algoritme ukazuje nakoľko sme sa priblížili k tomu, aby bola matica semi-ortogonálna. Z tejto operácie by sme mali dostávať hodnoty blízke 0.

Ak na jednotkovú maticu aplikujeme výpočet 4.4, dostaneme hodnotu 0. Pre vizualizáciu rozdielu medzi náhodnou maticou a semi-ortogonálnou maticou využijeme algoritmus na redukciu dimenzií t-SNE [25]. Obe matice majú veľkosť *320*640* a sú zredukované na veľkosť *320*2*. Na obrázku 4.1 sú hodnoty náhodnej matice rozptýlené, na rozdiel od semi-ortogonálnej, kde sú hodnoty akumulované v okolí hodnoty 0.

Následne sme porovnali aj kosínusové vzdialenosti [53] medzi všetkými dvojicami vektorov v oboch maticiach. Ak sú dva vektory ortogonálne, kosínusová vzdialenosť je rovná 0. Sčítaním vzdialeností dostaneme sumu S_{all} reprezentujúcu priemernú kosínusovú vzdialenosť vektorov v matici. V prípade semi-ortogonálnej matice je $S_{all} = -1.9712e^{-07}$, čo nám potvrdzuje ortogonalitu vektorov. Pri náhodnej matici je hodnota výrazne vyššia, $S_{all} = 38266.2109$.



Obr. 4.1: Porovnanie redukcie semi-ortogonálnej matice a náhodnej matice pomocou algorimu t-SNE. Červenou farbou je označená náhodná matica. Zelenou farbou je označená semi-ortogonálna matica.

Tento princíp ortogonality a jej aplikovanie na sieť TDNN predstavíme v ďalšej kapitole. Aplikovanie na E2E systém predstavíme v časti 6.2.

Kapitola 5

Porovnanie nástrojov na učenie s nízko-dimenzionálnou faktorizáciou

Pre otestovanie správnej funkčnosti nízko-dimenzionálnej faktorizácie v jazyku Pytorch použijeme pre porovnanie existujúcu implementáciu v nástroji Kaldi. V prvej časti kapitoly si predstavíme architektúru TDNN s faktorizáciou. Následne si popíšeme spôsob implementácie v oboch nástrojoch a predstavíme výsledky experimentov.

5.1 Architektúra časovo oneskorených neurónových sietí s faktorizáciou

Zavedením nízko-dimenzionálnej faktorizácie do modelu TDNN [36] vytvoríme TDNN-F model. Váhovú maticu W vo vrstve TDNN na obrázku 3.5 rozložíme na dve matice M, N (obrázok 5.1), pre ktoré platí $W = MN$. Jednu z dvoch matíc budeme počas učenia upravovať pomocou algoritmu popísaného v časti 4. Táto matica bude teda spĺňať podmienky semi-ortogonálnej matice. Na obrázku 5.1 môžeme vidieť, že medzi maticami M a N je vložený *bottleneck* s rozmermi menšími ako je veľkosť parametrov matice W . Napríklad, predpokladajme maticu váh W veľkosti 100*50. Túto maticu rozdelíme pomocou *bottlenecku* veľkosti 10 na 2 matice vo veľkostiach 100*10 a 10*50, a týmto znížime celkový počet parametrov z pôvodných 5000 na 1500, čo je výrazná redukcia.

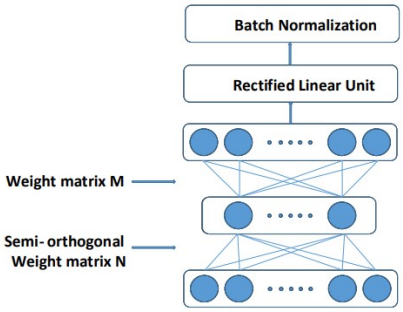
5.2 Implementácia v nástroji Kaldi a Pytorch

Našou úlohou bolo z naštudovanej problematiky a z existujúceho kódu v nástroji Kaldi implementovať TDNN a TDNN-F sieť v jazyku Pytorch.

Kaldi

V rámci nástroja Kaldi sa siete implementujú pomocou konfiguračných súborov. Hlavnými stavebnými blokmi sú:

- `linear-component` - je to klasická lineárna vrstva, u ktorej potrebujeme určiť veľkosť a vstup. Pre potreby TDNN bloku sa vstup do tejto siete zapisuje ako zoznam rámcov z predchádzajúcej vrstvy



Obr. 5.1: Faktorizovaná TDNN vrstva s *bottleneckom* a jednou semi-orthogonálnou maticou. Prevzaté z [36].

- `relu-batchnorm-layer` - vrstva, ktorá vykoná najprv lineárny výpočet a k tomu následne vykoná výpočet aktivačnej funkcie ReLU a ako posledné vypočíta dávkovú normalizáciu
- `tdnnf-layer` - TDNN-F vrstva, určujeme jej veľkosť, zároveň aj rozmer *bottlenecku* a taktiež `time-stride`, ktorý určuje ktoré rámce sa majú zobrať z predchádzajúcej vrstvy. Berie sa stredný rámec a potom rámec posunutý súmerne o danú hodnotu vľavo a vpravo.
- `output-layer` - je klasická lineárna výstupná vrstva, ktorá zo vstupu vypočíta logaritmický softmax[1]

Napríklad, pre model TDNN s veľkosťou vrstvy 520 a s 3 vstupnými rámcami z predchádzajúcej vrstvy bude aktuálny, predchádzajúci a nasledujúci rámec v konfiguračnom súbore zapísaný nasledovne:

```
relu-batchnorm-layer name=tdnn dim=520 input=Append(-1,0,1)
```

Pre model TDNN-F s veľkosťou vrstvy 520 a veľkosťou *bottlenecku* 96 a s 3 vstupnými rámcami z predchádzajúcej vrstvy je aktuálny, predchádzajúc a nasledujúci rámec v konfiguračnom súbore zapísaný nasledovne:

```
tdnnf-layer name=tdnnf dim=520 bottleneck-dim=96 time-stride=1
```

Pytorch

Pri implementácii TDNN sietí sme vychádzali najmä z existujúcich implementácií [6][9]. Hlavný rozdiel v týchto implementáciách je to, akým spôsobom sa pristupovalo k skladaniu rámcov v prípade načítavania rámcov z indexov -3,0 a 3 predchádzajúcej vrstvy.

Najprv sme používali funkciu *unfold*, ktorú aplikujeme na maticu x:

```
x.unfold(dimension, size, step)
```

Funkcia simuluje operáciu posuvného okna. Parameter `dimension` znamená, na akej dimenzii matice sa má operácia vykonať, parameter `size` určuje, na aké veľké časti sa bude deliť matica x a nakoniec `step`, ktorý určí, o koľko sa bude posúvať pri vytváraní jednotlivých okien v danej dimenzii.

[1] funkcia softmax - https://en.wikipedia.org/wiki/Softmax_function

Počas testovania implementácie sme prišli na to, že dané riešenie nie je veľmi optimálne. Toto bolo spôsobené tým, že funkcia `unfold` nie je dobre optimalizovaná a výpočet aj pri menšej sieti trval príliš dlho v porovnaní s druhým prístupom, ktorý je založený na 1D konvolučných neurónových sieťach. V Pytorchi sa na to používa modul s názvom `nn`[2] a jeho funkcia `Conv1d`:

```
torch.nn.Conv1d(in_channels, out_channels, kernel_size
                stride=1, padding=0, dilation=1)
```

Táto funkcia má parametre pre nastavenie veľkosti vstupu a výstupu `in_channels`, `out_channels`. Veľkosť konvolučnej matice označuje parameter `kernel_size` a posledné tri parametre ovplyvňujúce spôsob výpočtu konvolúcie: `stride, padding, dilation`, podrobnejšie popísané v časti 3.1.1. V tomto spôsobe implementácie sme dosiahli požadovanú funkcionalitu preskakovania rámcov za použitia parametra `dilation`, ktorý nastavuje medzery medzi prvkami v konvolučnej matici, vďaka čomu budeme môcť zo vstupu extrahovať napríklad rámce z indexov -3,0 a 3 predchádzajúcej vrstvy a spojiť ich do jedného vektoru, ktorý dáme ako vstup do lineárnej vrstvy. Tento prístup môžeme použiť len na symetrické indexovanie, napríklad indexy -7,2,4 sa nám pomocou tejto funkcie nepodarí extrahovať do jedného vektoru.

V našej implementácii je blok TDNN definovaný funkciou, ktorá na vstupe očakáva kontext ako list indexov, vstupný a výstupný rozmer, napríklad s hodnotou 520. Ďalší parameter `full_context` má buď hodnotu `true`, kedy sa berú hodnoty aj medzi hodnotami v liste z parametra `context`, alebo hodnotu `false`, kedy sa berú len hodnoty v liste:

```
TDNN(context=[-1, 1], input_channels=520,
     output_channels=520, full_context=True)
TDNN(context=[-3, 0, 3], input_channels=520,
     output_channels=520, full_context=False)
```

Ako výstupnú vrstvu používame funkciu, u ktorej sa nastavuje veľkosť vstupného a výstupného vektoru:

```
OUTPUT_TDNN(input_dim=520, output_dim=2032)
```

V rámci implementácie modelu TDNN sme použili nasledovné funkcie z Pytorch modulu `nn` pre stavbu neurónových sietí:

- `nn.ReLU()` - funkcia s aktivačnou funkciou typu ReLU
- `nn.BatchNorm1d()` - funkcia vypočíta dávkovú normalizáciu vstupnej matice
- `nn.Dropout()` - funkcia aplikuje dropout metódu na váhy danej vrstvy (3.2)

Funkcie sú aplikované na výstupné matice z konvolučnej vrstvy v poradí `ReLU()`, `BatchNorm1D()` a `Dropout()`.

Pri výstavbe celého modelu je dôležité dbať na to, aby celý kontext, ktorý dostaneme na vstupe, prešiel sieťou a posledná výstupná vrstva nám dala len jeden výsledný vektor. To znamená, že ak pridáme ďalšiu vrstvu s hodnotou `context=[-3,0,3]`, budeme musieť vstupný kontext rozšíriť o 3 rámce vľavo, respektíve vpravo.

Blok TDNN-F je v našej implementácii definovaný funkciou, ktorá na vstupe berie rovnaké argumenty ako blok TDNN, rozšírené o argument veľkosti *bottleneck*:

[2]Pytorch modul pre neurónové siete - https://pytorch.org/docs/stable/nn.html

```
TDNNF(context=[-1, 1], input_channels=520, bottleneck_channels=96,
      output_channels=520, full_context=True)
TDNNF(context=[-3, 0, 3], input_channels=520, bottleneck_channels=96,
      output_channels=520, full_context=False)
```

Blok TDNN-F je implementovaný podobne ako blok TDNN, kde pri výstupe z bloku rovnako používame funkcie `ReLU()`, `BatchNorm1D()` a `Dropout()` v danom poradí. Narozdiel od bloku TDNN ale výstupnú veľkosť konvolučnej vrstvy nastavujeme na hodnotu `bottleneck_channels`, následne nato používame funkciu transponovanej konvolúcie `ConvTranspose1d`, ktorej konvolučný filter nastavíme na veľkosť 1, takže z toho urobíme klasickú lineárnu sieť, ktorá len propaguje vstup o veľkosti *bottleneck* na výstup s veľkosťou rovnou hodnote parametra `output_channels`.

Pre výpočet semi-ortgonálnej matice je potrebné, aby sa vektory v stĺpci alebo riadku danej matice približovali k vzájomnej ortonormalite. Stĺpec alebo riadok sa vyberie podľa toho, ktorý rozmer je väčší. V implementácii očakávame na vstupe maticu s väčším počtom stĺpcov. Na výpočet použijeme nasledujúcu funkciu implementovanú v jazyku Pytorch, kód je inšpirovaný implementovaným kódom z nástroja Kaldi[3]:

```
def orthonormal_matrix_conv(M):
    update_speed = 0.125
    #P = M~M^T
    P~= torch.mm(M, torch.transpose(M,0,1))
    trace_p = torch.trace(P)
    trace_p_p = torch.trace(torch.mm(torch.transpose(P,0,1), P))
    scale = torch.sqrt(trace_p_p / trace_p)
    ratio = (trace_p_p * P.shape[0] / (trace_p * trace_p))

    #ked to je nad 1.0 tak by sme mali konvergovat pomalsie
    if ratio > 1.02:
      update_speed *=0.5
      if ratio > 1.1:
        update_speed *=0.5

    #P = (P - scale^2 * I).
    subs = torch.eye(P.shape[0]).cuda()*(-1*scale*scale)
    P~= P~+ subs

    #vypocet frobenius norm, teda kontroly konvergovania
    fb = torch.sqrt(torch.trace(torch.mm(torch.transpose(P,0,1), P)))

    alpha = update_speed / (scale * scale);

    #vypocet aktualizacnej matice
    M_update = -4 * alpha * torch.mm(P, M)
    M~= M~+ M_update
    return M
```

[3] funkcia ConstrainOrthonormalInternal() - https://github.com/kaldi-asr/kaldi/blob/5ca36b9adc6a2f85a921323f31313cccf41e599d/src/nnet3/nnet-utils.cc

V kóde sa používajú nasledovné funkcie jazyku Pytorch:

- `torch.transpose` - funkcia transponuje vstupnú maticu
- `torch.trace` - funkcia vypočíta sumu elementov na diagonále
- `torch.mm` - funkcia dostane na vstup dve matice, ktoré medzi sebou vynásobí
- `torch.sqrt` - funkcia odmocní každý element vstupnej matice
- `torch.eye` - funkcia vytvorí maticu danej veľkosti, kde na diagonále sú jednotky, vo zvyšku sú nuly.

Predchádzajúci kód obsahuje aj komentáre, ktoré vysvetľujú jednotlivé kroky. Vo funkcii sa používa konštantná premenná `update_speed`. Touto hodnotou sa vypočíta premenná `alpha`, ktorá určuje veľkosť kroku pri úprave vektorov matice. Výpočet je možné vidieť vo vyššie spomenutej rovnici 4.3. Na kontrolu konvergovania matice k správnemu riešeniu sa používa výpočet z rovnice 4.4.

V trénovacom cykle aplikujeme funkciu `orthonormal_matrix_conv` až po výpočte výstupných hodnôt a spätnej propagácii. Upravujú sa ňou príslušné váhy TDNN-F vrstiev. Váhy upravujeme v priemere každú 4 iteráciu v rámci epochy a zároveň každú druhú epochu.

```
weight_matrix = m.temporal_conv.weight.data[:,:,c].detach()
updated_weight_matrix = orthonormal_matrix_conv(weight_matrix)
```

Pomocou premennej `weight.data` zavolanej na danú vrstvu dostaneme váhovú maticu. Aktualizácia váh bude prebiehať práve na tejto matici a jednotlivými výpočtami bude jej tvar konvergovať k semi-ortogonálnemu tvaru. Operácie s týmito váhovými maticami sa v implementácii trénovania v jazyku Pytorch zaznamenávajú do celkovej trénovacej štruktúry neurónovej siete. V dôsledku toho je veľmi dôležité použitie funkcie `detach()` na požadovanú maticu, ktorá túto vlastnosť pre danú maticu vypne. Ak by sme ju nepoužili, spôsobilo by to problémy vo výpočtoch spätnej propagácie.

Pre načítavanie dát v Kaldi formáte existuje implementovaný modul pre Pytorch s názvom `kaldiio`[4]. V implementácii využívam funkciu na načítanie `.scp` súboru `feats_path`, ten odkazuje na jednotlivé `.ark` súbory obsahujúce vektory jednotlivých rámcov a zároveň súbor `alis_path`, ktorý obsahuje výstupné hodnoty porovnávané s výstupom trénovanej siete:

```
kaldiio.load_scp(feats_path)
kaldiio.load_scp(alis_path)
```

Vypočítaný model v Pytorch musíme napojiť na dekódovanie pomocou nástroja Kaldi a na tento účel použijeme už spomínaný modul `kaldiio`. Modul zapíše získané pravdepodobnosti do `.ark` formátu. Tieto pravdepodobnosti následne poslúžia ako vstup pre binárny súbor Kaldi s názvom `latgen-faster-mapped`. Ten na vstup očakáva už predom natrénovaný jazykový model. Nakoniec Kaldi vygeneruje potrebné dáta pre ďalšie utility nástroja Kaldi generujúce konečnú vetu, z ktorej sa vypočíta výsledný WER.

[4] kaldiio python modul pre čítanie a zapisovanie súborov typu `.ark` a `.scp` - https://pypi.org/project/kaldiio/

typ a poradie vrstiev	indexy kontextu	veľkosť výstupnej vrstvy
1. tdnn	-2,-1,0,1,-2	520
2. tdnn	-1,0,1	520
3. tdnn	-3,0,3	520
4. tdnn	-3,0,3	520
5. tdnn	-3,0,3	520
6. lineárna	X	2032

Tabuľka 5.1: Nastavenie neurónovej siete a hodnoty argumentov jej jednotlivých vrstiev.

5.3 Experimenty

Experimenty sme vykonávali spočiatku na službe Google Cloud Compute[5]. Pri prvej registrácií dostane užívateľ istý počet kreditov na počítanie zadarmo. Nakoniec sme sa ale presunuli na Metacentrum[6], kde boli koncom roka spustené nové uzly s grafikami umožňujúce vykonávať nové trénovania takmer okamžite bez zdĺhavého čakania v rade, čo nás pôvodne odrádzalo od používania tejto infraštruktúry.

Dáta

Pre naše potreby vytvorenia siete v Pytorch a dosiahnutia podobných výsledkov ako v Kaldi sme sa rozhodli použiť dataset *Mini librispeech ASR Corpus*[7] s dĺžkou trénovacích dát približne 5,2 hodín.

Na experimentovanie nad týmto dátovým setom je v Kaldi k dispozícii implementovaný návod. Jednotlivé rámce sú reprezentované ako *high-resolution MFCC*, sú to klasické rozšírené *MFCC* príznaky, ktorých veľkosť je 40 namiesto klasickej veľkosti 13. Tieto príznaky nám poslúžia ako vstup do neurónovej siete. Ako výstup neurónovej siete budeme mať vektor pravdepodobnostných hodnôt *pdf-id* pre každý vstupný rámec. Sekvencia týchto *pdf-id* sa používa ako vstup do dekodéru, ktorý vytvára cieľovú vetu.

Konfigurácie a výsledky

Ako základný model pre porovnanie implementácie Kaldi a Pytorch sme zvolili 5-vrstvovú architektúru vystavanú z TDNN vrstiev. Jednotlivé hodnoty siete sú nastavené podľa tabuľky 5.1. Veľkosť vstupného kontextu danej siete je 25 rámcov, presnejšie veľkosť ľavého kontextu je 12 a pravého taktiež 12. Skripty na učenie v nástroji Kaldi sme upravili a zjednodušili tak, aby sme dostali čo najvernejšie výsledky, bez optimalizačných prvkov, ktoré ponúka nástroj Kaldi. Jedným z týchto prvkov bolo použitie *iVec* [16] a taktiež sme vypli niektoré optimalizačné nastavenia.

Ako hodnotiacu funkciu sme použili funkciu `cross-entropy` (2.2) a ako optimalizačný algoritmus na úpravu váh sme použili klasický SGD (2.2). Pre účely porovnania sme museli zjednotiť trénovaciu a testovaciu sadu, ktorá sa vytvára v oboch prípadoch za behu. Pre tento účel sme vytvorili súbor, ktorý obsahuje indexy pre trénovacie a testovacie dáta.

Modely v Kaldi a Pytorch sa líšia trénovacím cyklom. Pri nástroji Kaldi sa používajú rôzne optimalizácie, ako napríklad priemerovanie výsledných modelov, ktoré sa v Pytorch

[5]Google Cloud Compute - https://cloud.google.com/compute/
[6]Metacentrum - https://metavo.metacentrum.cz/
[7]mini librispeech dataset - https://www.openslr.org/31//

nástroj	% WER	dĺžka trénovania (1 epocha)
KALDI	19.12	~2.5 min
Pytorch	19.81	~25 min

Tabuľka 5.2: Porovnanie úspešnosti 5-vrstvových sietí typu TDNN na testovacích dátach v nástroji Kaldi a v jazyku Pytorch.

typ a poradie vrstiev	indexy kontextu	veľkosť bottlenecku	veľkosť výstupnej vrstvy
1. tdnn	-2,-1,0,1,-2	0	520
2. tdnnf	-1,0,1	96	520
3. tdnnf	-3,0,3	96	520
4. tdnnf	-3,0,3	96	520
5. tdnnf	-3,0,3	96	520
6. lineárna	0	0	2032

Tabuľka 5.3: Nastavenie neurónovej siete a hodnoty argumentov jej jednotlivých vrstiev.

neimplementovali. Spôsob úpravy učiaceho faktoru počas učenia je taktiež odlišný. Jedna s odlišností sa prejavuje ako pomalšie konvergovanie ku správnemu riešeniu. Celkovo nebol pozorovaný vplyv na finálnu úspešnosť modelu.

K porovnaniu modelov používame hodnotenie chybovosti WER, ktoré vychádza z totožného jazykového modelu. Výsledky porovnania sú zobrazené v tabuľke 5.2. Vidíme, že rozdiel chybovosti je menší než 1%, čo je z nášho pohľadu vcelku úspešné. V príslušnej tabuľke je zjavný rozdiel v dĺžke trénovania jednej epochy. Túto dlhú výpočetnú dobu jednej epochy v prípade Pytorch implementácie sme sa snažili napraviť, no žiaľ neúspešne. Zefektívnenie trénovania sa nám podarilo uskutočniť nahradením funkcie `unfold`, spomínanej na začiatku sekcie 5.2, konvolučnými vrstvami. Funkcia `unfold` trénovanie jednej epochy predĺžila takmer 2-násobne oproti terajšej implementácii.

Následne sme porovnali sieť vyskladanú z vrstiev TDNN-F. Jednotlivé argumenty sú podobné vrstve TDNN, pribudol nám však parameter pre veľkosť *bottlenecku*, ako môžeme vidieť v tabuľke 5.3. Výsledok porovnania WER aj času medzi jednotlivými implementáciami je zobrazený v tabuľke 5.4. Vidíme, že chybovosť je medzi rozdielnymi nástrojmi opäť veľmi podobná a líši sa len o desatinné hodnoty. Sieť v Pytorch je ale opäť výrazne pomalšia. Rýchlejšia je len v porovnaní so sieťou TDNN, a navyše dosahujeme oproti nej aj lepšie výsledky. Zvýšenie rýchlosti je spôsobené najmä vďaka výraznému zníženiu celkového počtu učiacich parametrov v rámci siete, kde v porovnaní so sieťou TDNN je počet parametrov o viac ako polovicu menší, ako vidíme v tabuľke 5.5.

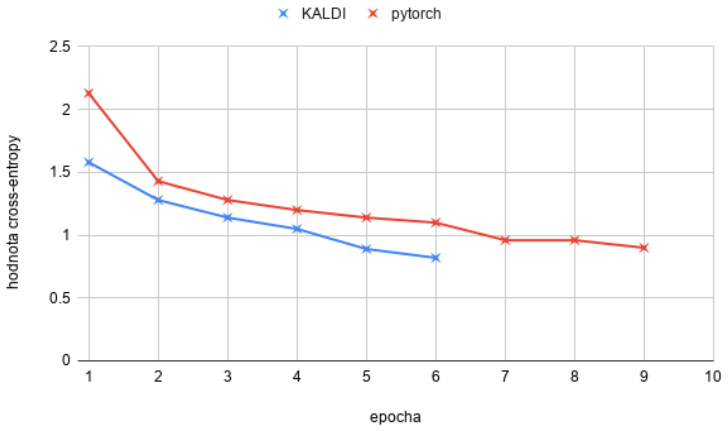
Napriek tomu, že sme dosiahli v implementácii sietí takmer totožné výsledky, nepodarilo sa nám napodobniť rýchlosť výpočtu. Z tabuliek (5.2, 5.4) pre porovnanie Kaldi a Pytorch

nástroj	% WER	dĺžka trénovania (1 epocha)
KALDI	18.9	~1.5 min
Pytorch	18.66	~15 min

Tabuľka 5.4: Porovnanie úspešnosti 5-vrstvových sietí typu TDNN-F na testovacích dátach v nástroji KALDI a v jazyku Pytorch.

typ siete	počet parametrov
TDNN	4 410 072
TDNN-F	1 926 392

Tabuľka 5.5: Porovnanie medzi počtom učiacich parametrov v sieti TDNN a TDNN-F.



Obr. 5.2: Porovnanie priemernej hodnoty objektívnej funkcie pre jednotlivé epochy.

môžeme vidieť, že dĺžka výpočtu jednej epochy v jazyku Pytorch bola v priemere takmer 10-násobná. Porovnanie počtu epoch potrebných k dosiahnutiu rovnakého výsledku ako v prípade trénovania v nástroji Kaldi je znázornené v grafe 5.2. Tento počet musel byť väčší v Pytorch ako v prípade nástroja Kaldi. Tieto odlišnosti sú spôsobené optimalizačnými technikami ako *limited-memory Broyden Fletcher Goldfarb Shannon* (L-BFGS) [24], *iVectors* [44], *backstitch* [49] a pod., ktoré sme nepreniesli z nástroja Kaldi do nami implementovanej siete.

typ siete	% WER
TDNN-F - 5 vrstiev (Pytorch)	18.66
TDNN-F - 7 vrstiev (Pytorch)	19.12
TDNN-F - 7 vrstiev (Kaldi)	18.83

Tabuľka 5.6: Tabuľka nám zobrazuje porovnanie úspešnosti 5 a 7-vrstvových sietí typu TDNN-F na testovacích dátach.

V rámci experimentov sme skúšali aj inicializáciu príslušných matíc v jednotlivých vrstvách pomocou funkcie `init.orthogonal_` v inicializácii siete v rámci jazyka Pytorch. Táto funkcia nám neposkytla prostriedky k zrýchleniu konvergovania k želanému výsledku.

V ďalšom kroku sme natrénovali hlbšiu sieť s počtom vrstiev 7. Každá nová vrstva mala indexy kontextu: -3,0,3, tak ako je popísané v tabuľke 5.3, a veľkosti výstupných vrstiev a *bottlenecku* sú podobne 520, respektíve 96. Z tabuľky 5.6 pozorujeme dosiahnutie horších výsledkov oproti 5 vrstvám. Zhoršenie je v rámci desatín percenta, no očakávali sme skôr nepatrné zlepšenie. V tomto prípade môže pomôcť implementácia *skip-connections*, ako je popísané v práci [36], ktorá ale absentuje v nástroji Kaldi. Naše pokusy o implementáciu neviedli k zlepšeniu hodnoty WER.

Kapitola 6

End-to-End rečové systémy s nízko-dimenzionálnou faktorizáciou

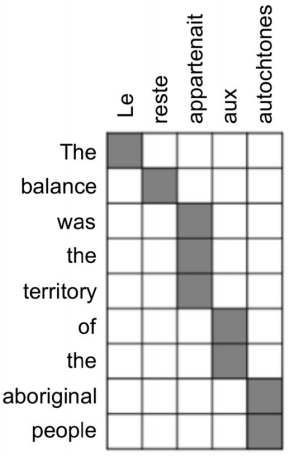
Typický ASR systém je rozdelený do niekoľkých komponentov pozostávajúcich z akustického, lexikologického a jazykového modelu. Prvotne boli modely založené na štatistických modeloch. V súčastnosti sa presúvajú jednotlivé implementácie modelov na neurónové siete, avšak každý model má vlastnú architektúru. S príchodom end-to-end systémov pre ASR sa architektúra prepájania viacerých modelov zjednodušuje na jeden model neurónovej siete. Veľkou výhodou tohto prístupu je zjednodušenie vývoja nových aplikácii ASR a zároveň ich použitie.

End-to-end ASR modely už priamo predpovedajú pravdepodobnosti slovných sekvencií W na základe sekvencie vstupných akustických rámcov X, teda hľadáme najlepšie W v $p(W|X)$, podobne ako v rovnici 2.1. Zatiaľ čo klasické ASR metódy rozkladajú tento problém na jazykový model a akustický model jednotlivo trénovaných na rozdielnych typoch dát, modely end-to-end používajú na učenie prevažne dvojicu, sekvenciu akustických rámcov a sekvenciu príslušných znakov. Zriedkavejšie je použitie priameho predpovedania sekvencie slov. Pri priamom predpovedaní slov by sme potrebovali veľké množstvo dát aby bolo každé slovo v slovníku dostatočne reprezentované, inak by to viedlo k zlej úspešnosti a zároveň by sme nedokázali predpovedať slová mimo slovníka.

End-to-end architektúry sa delia na dva hlavné typy. *Attention-based* modely sú založené na *Attention* mechanizme [7]. Ten predpovedá zarovnanie medzi sekvenciou rámcov a znakov. Tento typ mechanizmu sa úspešne používa pri problémoch spracovania prirodzeného jazyka, ako je napríklad preklad viet na obrázku 6.1, kde sa hľadajú zarovnania slov medzi dvojicou jazykov.

Ďalším typom architektúry používanej v ASR je CTC. Ten nám pomáha s problémom zarovnania sekvencií pri učení. Vďaka CTC nepotrebujeme presne dané časové rámce pre jednotlivé znaky, postačí nám sekvencia, narozdiel od *cross-entropy* (2.2).

Attention mechanizmus je založený na enkodér-dekodér architektúre, ktorú by sme mohli popísať aj ako mapovanie sekvencie akustických rámcov na textovú sekvenciu. Enkodér načítava vstupnú sekvenciu a dekodér predikuje výstupnú textovú sekvenciu. Vytvoríme závislosti medzi výstupnou skrytou vrstvou enkodéru a vstupom do dekodéru. Nevýhodou tejto mechaniky je produkcia nesekvenčných zarovnaní, čo spôsobuje problémy v rozpoznávaní reči, kde je postupnosť rámcov dôležitá. Problém môže taktiež spôsobovať veľký



Obr. 6.1: Zarovnanie slov medzi anglickým a francúzskym jazykom pomocou *Attention-based* modelu. Môžeme vidieť nesekvenčné zarovnanie napríklad pri francúzskom slove 'appartenait', ktorému prislúchajú tri slová v anglickom jazyku 'was the territory'. Prevzaté z [2].

rozdiel v dĺžkach vstupnej a výstupnej sekvencie. V [52] je preto navrhnutá architektúra, ktorá rieši tieto problémy a to prepojením dvoch typov end-to-end architektúr *Attention* a CTC.

6.1 Výpočet rozpoznávania reči v End-to-End rečových systémoch

V nasledujúcej časti si predstavíme princíp hodnotiacej funkcie pre určenie správnej sekvencie znakov s použitím architektúry E2E, v ktorej používame dve techniky: CTC a *Attention* mechanizmus.

CTC

CTC predpovedá sekvenciu C vzájomne odlišných písmen z abecedy $\mathcal{U}$ veľkosti L, $C = c_l \in \mathcal{U} | l = 1, \ldots, L$. Na vstupe je sekvencia rámcov $X = x_t \in \mathbb{R}^D | t = 1, \ldots, T$, a D je veľkosť vektoru akustického rámca. CTC navyše používa prídavný symbol *blank*, ktorý slúži ako neurčitý symbol, ktorý je predpovedaný ak si metóda pri vstupnom rámci, nie je istá aký iný symbol abecedy $\mathcal{U}$, by to mohol byť. Abecedu $\mathcal{U}$ teda rozšírime o znak *blank* $(-)$, $\mathcal{U}' = \mathcal{U} \cup < - >$. V CTC použijeme sekvencie $Z = \{z_1, \ldots, z_T\}, z_t = \{z_t^1, \ldots, z_t^{|\mathcal{U}+1|}\}$, kde z_t^i určuje pravdepodobnosť že výstupom bude symbol i z abecedy $\mathcal{U}'$ v čase t.

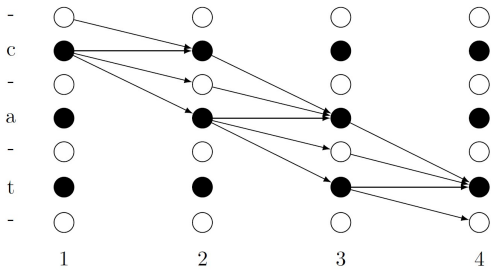
K prevodu sekvencií Z na finálnu sekvenciu C použijeme mapovaciu funkciu $\mathcal{R}()$. Táto funkcia pozostáva z dvoch operácií:

- Spojenie totožných susedných znakov. Ako ukážku môžeme použiť sekvencie '`c-aa-t`' a '`ccc-a-t`', ktoré sa rovnako prevedú na redukovanú sekvenciu '`c-a-t`'.
- Odstránenie symbolu *blank* $(-)$, keďže indikuje neurčitosť. Napríklad sekvencia '`c-a-t`' sa redukuje na finálnu sekvenciu '`cat`'.

Ak by sme mali na vstupe 4 rámce a očakávali by sme sekvenciu '`cat`', počet možných sekvencií Z by sa rovnal 7:

$$\mathcal{R}(\{\texttt{ (-cat),(c-at),(ccat),(ca-t),(caat),(cat-),(catt) }\})=\texttt{'cat'}$$

Možné sekvencie vieme zobraziť aj v sieti na obrázku 6.2, založenej na algoritme *dynamic time warping* (DTW) [41]. Povolené operácie v sieti sú posun na nasledujúci znak alebo zotrvanie na aktuálnom znaku. Nutnosťou je posun v čase, nie je možné vrátiť sa späť.



Obr. 6.2: Hodnoty 1 až 4 predstavujú krok v čase. 'c','a','t' a '-' označujú možné výstupné znaky. Prevzaté z [48].

Výpočet pravdepodobnosti výstupnej sekvencie abecedy $\mathcal{U}'$ k vstupným rámcom je v nasledujúcej rovnici (6.1):

$$p(\pi|X) = \prod_{t=1}^{T} z_t^{\pi_t}, \forall \pi \in \mathcal{V}^T, \tag{6.1}$$

kde π_t reprezentuje znak sekvencie π v čase t. $\mathcal{V}^T$ je kolekcia možných sekvencií π o veľkosti T. Keďže jednotlivé znaky vo výstupnej sekvencii Z sú na sebe nezávislé, môžeme ich medzi sebou vynásobiť, čím dostaneme pravdepodobnosť pre všetky sekvencie π k vstupným rámcom X. Nakoniec si definujeme funkciu CTC určujúcu pravdepodobnosť sekvencie C z rámcov X:

$$p_{ctc}(C|X) = \sum_{\pi \in \mathcal{R}^{-1}(C)} p(\pi|X), \tag{6.2}$$

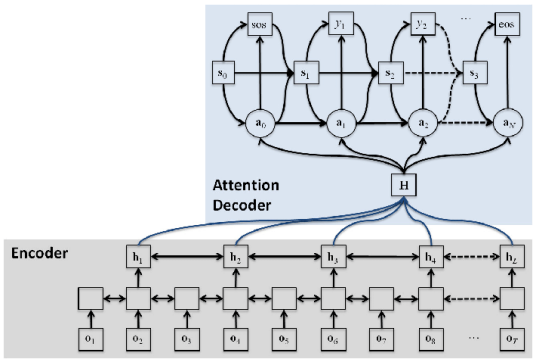
kde rovnica $p(\pi|X)$ je definovaná v 6.1. $\mathcal{R}^{-1}(C)$ je inverzná operácia generujúca všetky sekvencie z kolekcie $\mathcal{V}^T$ pre príslušnú sekvenciu znakov C zo vstupných rámcov X. V E2E systéme sa namiesto vstupných rámcov, používajú hodnoty z enkodéru, kde sú časové rámce x_t, reprezentované vektorom h_t (viď. obrázok 6.4).

Attention mechanizmus

Na rozdiel od CTC prístupu, *Attention* mechanizmus nevytvára žiadne nezávislé predpoklady na jednotlivé časové rámce h_t, ale priamo predpovedá celú výstupnú sekvenciu zo zakódovanej vstupnej sekvencie H a hľadá medzi nimi závislosti, ako je zobrazené na ilustračnom obrázku 6.1. Závislosti nemusia byť sekvenčné. *Attention* model (zobrazený na obrázku 6.3) pozostáva z dvoch častí, a to enkodéru a dekodéru.

V rovnici 6.3 definujeme enkodér, ktorý je prevažne implementovaný nejakým typom rekurentnej neurónovej siete. Enkodér konvertuje celú vstupnú sekvenciu rámcov X na skrytý vektor h_t.

Ďalej si v rovnici 6.5 definujeme funkciu *Attention*(), bližšie popísanú v [8]. Z danej funkcie dostaneme *Attention* váhu, ktorá reprezentuje zarovnanie medzi vektorom h_t a každým znakom vo výstupnej sekvencii c_l



Obr. 6.3: *Attention* model zložený z enkodéru a dekodéru. Prevzaté z [31].

Dekodér v rovnici 6.6 je taktiež implementovaný rekurentnou sieťou. Určuje nám pravdepodobnosť nasledujúceho znaku v sekvencii C z predchádzajúcich znakov c_{l-1} a vstupných akustických rámcov. Tieto rámce sú na vstupe do dekodéru vo forme vektorov získaných zo skrytej vrstvy predchádzajúceho stavu dekodéru q_{l-1} a váženého súčtu r_l medzi súčtom *Attention* hodnôt a_{lt} a zakódovaným výstupom z enkodéru h_t.

$$h_t = Encoder(X) \tag{6.3}$$

$$a_{lt} = Attention(\{a_{l-1}\}_t, q_{l-1}, h_t) \tag{6.4}$$

$$r_l = \sum_{t}^{T} a_{lt} h_t \tag{6.5}$$

$$p(c_l|c_1, \ldots, c_{l-1}, X) = Decoder(r_l, q_{l-1}, c_{l-1}) \tag{6.6}$$

V [52] je detailnejší popis odvodenia pravdepodobnostnej rovnice pre attention:

$$p_{att}(C|X) = \prod_{l=1}^{L} p(c_l|c_1, \ldots, c_{l-1}, X) \tag{6.7}$$

Výhodou tohto modelu je jeho komplexnosť a možnosť trénovania aj ASR problémov len pomocou jednej hĺbkovej neurónovej siete. Nevýhodu, ktorú má oproti bežným metódam a metóde CTC, je vytváranie nepravidelných zarovnaní, ako už bolo skôr poukázané.

Pri rozpoznávaní znakov z E2E systému sa vo všeobecnosti hľadá najpravdepodobnejšia sekvencia znakov $\hat{C}$ zo vstupných rámcov X:

$$\hat{C} = \underset{C \epsilon \mathcal{U}^*}{argmax}\{log p_{att}(C|X)\} \tag{6.8}$$

Dekódovaním postupne vytvárame skupinu hypotéz Ω_l dĺžky l. Počiatočná hypotéza Ω_0 obsahuje jeden štartovací symbol $< sos >$. Skóre jednotlivých hypotéz sa určuje pomocou rovnice 6.9, kde pre hypotézu $\alpha(< sos >, X)$ je skóre rovné 0. Každá ďalšia hypotéza pre dĺžky od $l = 1$ do L bude expandovaná na základe Ω_{l-1} a uložená do skupiny hypotéz Ω_l.

$$\alpha(h, X) = \alpha(g, X) + log\alpha(c|g, X), \tag{6.9}$$

kde g označuje jednu z čiastočných hypotéz zo skupiny Ω_l, c je nasledujúci znak pridaný ku g a h je novo vytvorená hypotéza $h = g \cdot c$. Ak má predpovedaný znak c hodnotu $< eos >$ pridá sa do skupiny dokončených hypotéz . Z nej sa nakoniec vyberie najlepšia hypotéza a teda najlepšia sekvencia znakov $\hat{C}$ podobne ako v rovnici 6.8. Z dôvodu zefektívnenia algoritmu sa určité množstvo hypotéz s najlepším skóre ukladá do premennej Ω_l. Problémy, na ktoré môžeme naraziť, sú predčasné generovanie finálneho znaku, čo nám generuje krátke sekvencie alebo, naopak, generovanie príliš dlhých sekvencií kvôli opätovnému generovaniu znakov z už videných vstupných rámcov.

Spojenie CTC a Attention

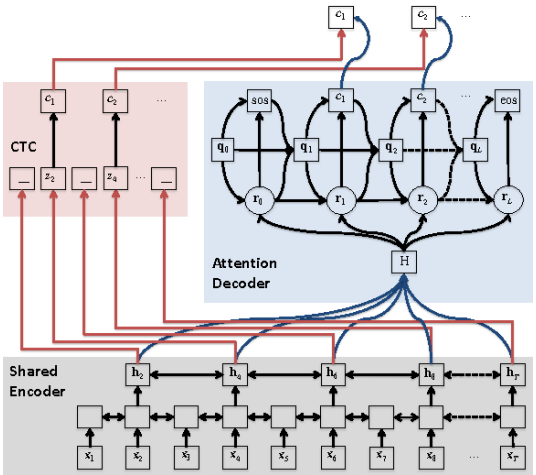
Spojením spomínaných dvoch algoritmov dostaneme hybridnú CTC/*Attention* sieť. Navrhnutá architektúra využíva výhody CTC trénovania a *Attention* mechanizmu.

V [52] je navrhnutá architektúra, ktorá k trénovaniu E2E architektúry používa ako výpomocnú objektívnu funkciu CTC . Vďaka nej vieme vynútiť sekvenčné zarovnanie medzi sekvenciou akustických rámcov a sekvenciou výstupných znakov, čím dôjde k zabráneniu veľkých skokov a opakovaniu rámcov.

V tejto spojenej architektúre budeme najlepšiu sekvenciu $\hat{C}$ určovať pomocou rovnice 6.10. To bude mať za následok vyriešenie spomínaného problému s repetíciou a s predčasným ukončením sekvencií.

$$\hat{C} = \underset{C \in \mathcal{U}^*}{argmax}\{\lambda log p_{ctc}(C|X) + (1-\lambda) log p_{att}(C|X)\} \tag{6.10}$$

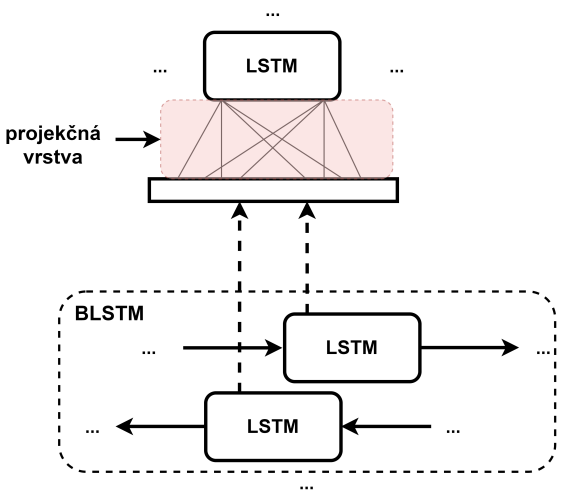
Hodnota λ je konštanta z intervalu: $0 \leq \lambda \leq 1$. V trénovacom cykle E2E systému λ určuje mieru použitia CTC a *Attention* metódy. V prípade $\lambda = 1$ sa dekódovanie sekvencie uskutoční len pomocou CTC. Dekódovanie len metódou *Attention* prebehne ak bude $\lambda = 0$. Ideálna hodnota $\lambda = 0.2$ bola hlásená v [21]. Obrázok 6.4 zobrazuje celkovú architektúru modelu.



Obr. 6.4: Hybridná CTC/*Attention* E2E architektúra. Zdieľaný enkodér je trénovaný oboma CTC a *Attention* modelmi súčasne. Enkodér transformuje sekvenciu $\{x_1, \ldots, x_T\}$ na zakódovanú postupnosť znakov $H = \{h_1, \ldots, h_T\}$. Dekodér spolu s CTC generuje finálnu sekvenciu znakov $\{c_1, \ldots, c_L\}$. Prevzaté z [21].

6.2 Architektúra s nízko-dimenzionálnou faktorizáciou

V pôvodnej architektúre je enkodér a dekodér implementovaný pomocou niekoľkých obojsmerných LSTM (BLSTM) vrstiev spojených lineárnymi vrstvami (BLSTMP), ukážka časti siete je na obrázku 6.5. V prípade dekodéru sa väčšinou používa len jedna BLSTM vrstva. Faktorizáciu budeme aplikovať na lineárne vrstvy enkodéru. Očakávame zlepšenie úspešnosti siete najmä v prípade menšieho množstva dát. V našej architektúre budeme využívať aj skôr prezentované modely TDNN a TDNN-F z kapitoly 3.1.2. Týmito modelmi naimplementujeme zdieľaný enkodér a nahradíme tak pôvodné BLSTMP vrstvy. Taktiež vyskúšame TDNN(-F) model spojiť s vrstvou BLSTMP.



Obr. 6.5: Časť BLSTMP enkóderu. Faktorizácia je aplikovaná na projekčnú vrstvu, ktorá je na obrázku zvýraznená červenou farbou.

Faktorizácia v enkodéry s BLSTMP

Enkodér v tejto architektúre funguje v zdieľanom režime, kde sú výstupné dáta poskytnuté pre *Attention* ale aj CTC model.

Jednotlivé vrstvy sú zostavené z vrstiev BLSTMP, ktoré obsahujú lineárne vrstvy. Tie nám vytvárajú projekciu výstupu jednej vrstvy BLSTM na vstup druhej vrstvy BLSTM. Výstupná dimenzia z BLSTM vrstvy má veľkosti N a jeho výstup kvôli obojsmernosti veľkosť $N * 2$. Lineárna vrstva má teda rozmery vstupu $N * 2$ a výstupu N. V tejto časti nám vzniká *bottleneck* podobne ako v prípade TDNN-F modelu. Je tu preto vhodné aplikovať faktorizáciu. Enkodér obsahuje niekoľko vrstiev (typicky tri až viac) a vďaka tomu máme k dispozícii niekoľko projekčných vrstiev, ktorých váhy možno upravovať a tým zároveň docieliť splnenie podmienky semi-ortogonality. Tento postup je podrobnejšie popísaný v časti 4.

Faktorizácia v enkodéri s TDNN(-F) vrstvami

Nad rámec toho sme vytvorili zdieľaný enkodér aj z modelu TDNN(-F), čím zmeníme pôvodnú architektúru E2E enkodéru. Použijeme rôzny počet vrstiev v závislosti na veľkosti vstupného kontextu a budeme vychádzať aj z architektúr použitých v časti 5.3. Vyskúšame

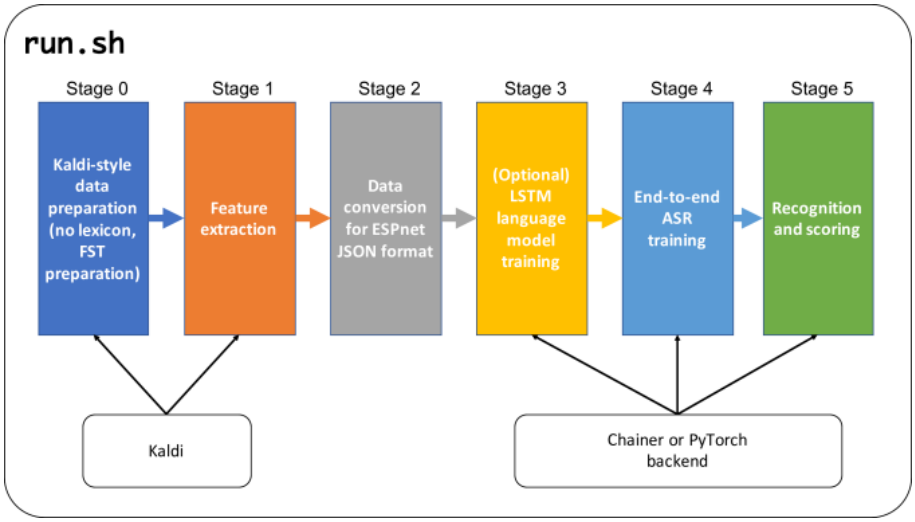
aj spojenie TDNN(-F) vrstiev s jednou BLSTMP vrstvou. Tú zapojíme na koniec, teda ako poslednú vrstvu na výstup z TDNN(-F) modelu, podobne ako v [30].

6.3 Implementácia v nástroji ESPnet

Hlavnou úlohou v tejto časti je v existujúcej E2E architektúre pre spracovanie reči implementovať faktorizáciu na určité vrstvy a zároveň implementovať enkodér pomocou vrstiev TDNN(-F). K tejto úlohe využijeme existujúci nástroj ESPnet [51] implementovaný v jazyku Pytorch. K tomu, aby vytvorené TDNN(-F) modely boli otestované mimo komplexného E2E systému, sme už skôr v kapitole 5.2 popísali ich aplikáciu pre rozpoznávanie reči.

ESPnet

ESPnet je voľne prístupný nástroj s E2E architektúrou na rozpoznávanie reči [51]. Implementovaný je v dvoch jazykoch pre neurónové siete: Pytorch a Chainer [46]. K príprave vstupných dát a najmä k extrakcii príznakov sa plne využíva nástroj Kaldi. My budeme využívať a upravovať najmä jeho implementáciu v jazyku Pytorch. K učeniu CTC používa ESPnet efektívnu knižnicu *warp CTC* [4], ktorá zrýchľuje učenie o 5-10%. V porovnaní s inými existujúcimi nástrojmi na ASR, akými sú Kaldi alebo Julius, má ESPnet počet riadkov kódu rádovo nižší, čím sa stáva prehľadnejším pre rôzne úpravy. Repozitár nástroja ESPnet už obsahuje vypracované návody pre rôzne známe datasety.



Obr. 6.6: Popis šiestich krokov v štandardnom návode v nástroji ESPnet pre jednotlivé experimenty. Prvé dve fázy používajú nástroj Kaldi. Prevzaté z [51].

Na obrázku 6.6 je zobrazený priebeh jednotlivých krokov v štandardnom návode v nástroji ESPnet. Skript pre ESPnet má niekoľko výhod, respektíve zjednodušení v porovnaní s návodmi pre Kaldi, ako napríklad: nie je potrebné pripravovať lexikón, trénovať zarovnania rámcov pomocou rôznych techník alebo kompilovať pravdepodobnostné rozhodovacie stromy.

Každý návod pozostáva z 6 fáz implementovaných v skripte *run.sh*:

- **Fáza 0** - Príprava dát: používa sa Kaldi súborová štruktúra [35].
- **Fáza 1** - Extrakcia príznakov: v tejto fáze sa vytvárajú FBank príznaky z akustických rámcov za pomoci existujúcich Kaldi nástrojov.
- **Fáza 2** - Konverzia dát: v tomto kroku sa z trénovacích a testovacích dát získaných z Kaldi vytvorí súbor vo formáte *json*, ktorý obsahuje všetky potrebné informácie pre nástroj ESPnet.
- **Fáza 3** - Trénovanie jazykového modelu: tento krok v našom prípade nepoužívame, pretože používame metriku CER. Zavedenie tohto kroku je výhodné u metriky WER k dosiahnutiu lepších výsledkov.
- **Fáza 4** - E2E trénovanie: pomocou dát vo formáte *json* natrénujeme nastavenú konfiguráciu enkodéru a dekodéru v systéme s hybridnou metódou CTC a *Attention*.
- **Fáza 5** - Rozpoznávanie: natrénované modely vo fáze 3 a 4 použijeme na vygenerovanie výsledkov z testovacích dát a určíme CER, respektíve WER.

V tejto práci sa budeme venovať len úpravám 4. fázy, v ktorej doplním priebeh trénovania modelu o faktorizáciu a taktiež budem adaptovať model podľa potrieb.

Generovanie príznakov

Na generovanie príznakov sa používajú utility z nástroja Kaldi, skripty sa nachádzajú v zložke `steps/`. Pre MFCC príznaky sa používa utilita `compute-mfcc-feats` pre FBank `compute-fbank-feats`. Obom utilitám sa pomocou parametrov v konfiguračnom súbore upresňujú nastavenia pre extrakciu príznakov. Väčšina návodov v nástroji ESPnet používa práve príznaky FBank.

6.3.1 Implementácia enkodéru

V tejto časti popíšeme implementáciu dvoch typov enkodérov. Jedným z nich je už existujúci enkodér vystavaný z BLSTM vrstiev, ktoré sú prepojené projekčnými vrstvami. Druhým je enkodér vystavaný z blokov TDNN(-F). Následne sa budeme zaoberať aj formou vstupu a výstupu pre E2E systém v nástroji ESPnet.

Enkodér s BLSTMP

ESPnet ponúka niekoľko typov enkodérov pre vytvorenie E2E modelu. Enkodér môže byť vyskladaný z vrstiev LSTM blokov alebo jednoduchších blokov *gated recurrent units* (GRU) [38]. K ich prepojeniu sa môžu použiť aj projekčné lineárne vrstvy alebo aj inicializačné bloky typu VGG, pomenované po akademickej skupine *Visual Geometry Group* [43]. Typ enkodéru určíme pomocou argumentu `etype`. Ten nastavíme podľa názvov vrstiev. V našom prípade sme používali argument s hodnotou `'blstmp'`, lebo potrebujeme model s obojsmernou LSTM vrstvou prepojenou lineárnymi vrstvami.

Ďalšie nastaviteľné argumenty pre enkodér v konfiguračnom súbore sú:

- `eunits` - Nastavuje počet pamäťových jednotiek v jednej vrstve LSTM, teda veľkosť výstupu.
- `eprojs` - Veľkosť vstupného vektoru do vrstvy LSTM.
- `elayers` - Počet vrstiev enkodéru.
- `subsample` - Zapisuje sa ako postupnosť čísel oddelená znakom '_'. Každé číslo určuje koľko rámcov z predchádzajúcej vrstvy preskočíme.

Na ukážku sme hodnoty nastavili na `eunits=320, eprojs=320, elayers=4, etype=blstmp` a enkodér je pomocou objektovej štruktúry v jazyku Pytorch popísaný nasledovne:

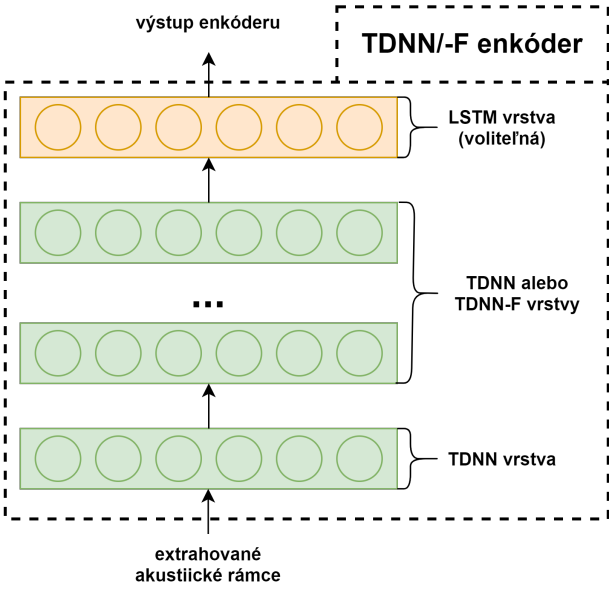
```
(enc): Encoder(
    (enc): ModuleList(
      (0): RNNP(
        (birnn0): LSTM(83, 320, batch_first=True, bidirectional=True)
        (bt0): Linear(in_features=640, out_features=320, bias=True)
        (birnn1): LSTM(320, 320, batch_first=True, bidirectional=True)
        (bt1): Linear(in_features=640, out_features=320, bias=True)
        (birnn2): LSTM(320, 320, batch_first=True, bidirectional=True)
        (bt2): Linear(in_features=640, out_features=320, bias=True)
        (birnn3): LSTM(320, 320, batch_first=True, bidirectional=True)
        (bt3): Linear(in_features=640, out_features=320, bias=True)
      )
     )
    )
```

Názvy BLSTM blokov vznikajú spojením reťazca 'birnn' a poradia vrstvy. Lineárne vrstvy vznikajú rovnako a to s použitím reťazca 'bt' a poradia vrstvy. Ďalšie parametre sme doplnili pre potreby výpočtu faktorizácie v učiacom cykle. Tieto parametre sa využívajú len v prípade trénovania modelu:

- `ortho-alpha` - Nastavuje rýchlosť konvergovania matice k jej semi-ortogonálnemu tvaru. Implicitná hodnota je 0,125.
- `layer` - Zapisuje sa ako postupnosť čísel oddelená znakom '_'. Každé číslo určuje poradie vrstvy a spojením s reťazcom 'bt' nám presne určí váhovú maticu pre výpočet faktorizácie. V prípade enkodéru s TDNN-F sa tento parameter používa len v prípade použitia vrstvy LSTM a jeho hodnota sa nastaví na 'last'. Na výpočet faktorizácie ale inak nie je potrebný z dôvodu fixného výpočtu faktorizácie v pevne určených váhových maticiach s *bottleneckom*.
- `ortho` - Zapína výpočet faktorizácie.
- `ortho-iteration` - Určuje frekvenciu výpočtu faktorizácie v závislosti na počte iterácií
- `ortho-epoch` - Určuje frekvenciu výpočtu faktorizácie v závislosti na počte epoch

Enkodér s TDNN(-F)

Pre zistenie vplyvu siete TDNN(-F) na E2E systém sme pôvodný enkodér vyskladaný z blokov BLSTM zamenili sieťou TDNN(-F). Jeho architektúra je zobrazená na obrázku 6.7.



Obr. 6.7: Architektúra enkodéru zostaveného z TDNN alebo TDNN-F vrstiev a voliteľnej vrstvy LSTM. Vstup predstavujú akustické rámce extrahované pomocou metódy FBank alebo MFCC. Výstup enkodéru sa používa ako vstup pre *Attention* mechanizmus a CTC.

Pre aplikáciu tohto enkodéru je potrebné nastaviť hodnotu `etype=tdnn`. Enkodér je vyskladaný z už naimplementovaných blokov z časti 5.2, kde prvé dva bloky sú fixné. Prvý blok je TDNN a berie vstup s kontextom [-2,2], druhý v poradí je už nami vybraný typ, ale s fixnou veľkosťou kontextu [-1,1]. Nemenné hodnoty sú v tabuľke 6.1 zvýraznené hrubým písmom. Ďalšie argumenty ovplyvňujúce enkodér sú:

- `etdnns` - Hodnota nastavuje veľkosti vstupného a výstupného vektoru pre všetky vrstvy v enkodéri okrem prvej vrstvy, ktorá berie na vstup parametrizované akustické rámce.
- `tdnn` - Základná hodnota je `True` a v tom prípade sú bloky typu TDNN. Ak bude hodnota `False`, použijú sa bloky TDNN-F.
- `bndim` - Tento parameter určuje veľkosť *bottlenecku* v prípade použitia TDNN-F blokov.
- `lstm` - Ak bude hodnota `True`, použitá výstupná vrstva bude typu BLSTMP.
- `eprojs` - Nastavuje veľkosti vstupného a výstupného vektoru pre blok BLSTM, ak sa používa.
- `strides` - Jednotlivé kontexty vrstiev vieme definovať postupnosťou čísel oddelených znakom '_'. Na ukážku, sieť z tabuľky 6.1 dostaneme ak bude `strides='1_3_3'` a `tdnn=True`.

typ a poradie vrstiev	indexy kontextu
1. tdnn	**-2,-1,0,1,-2**
2. tdnn	**-1,0,1**
3. tdnn (`stride=1`)	-1,0,1
4. tdnn (`stride=3`)	-3,0,3
5. tdnn (`stride=3`)	-3,0,3

Tabuľka 6.1: Tabuľka nám zobrazuje nastavenie vrstiev enkodéru v programe ESPnet pri nastavení parametra `strides='1_3_3'` a `tdnn=True`. Hrubým písmom sú zvýraznené nemenné hodnoty.

V prípade použitia TDNN-F blokov je potrebné zapnúť výpočet faktorizácie pomocou argumentu `ortho` a taktiež je možné nastaviť `ortho-alpha`, `ortho-iteration` a `ortho-epoch`.

K modifikovaniu siete v existujúcom kóde v ESPnet potrebujeme upraviť súbor `e2e_asr.py` v časti `pytorch-backend`. Na základe vstupného parametra `etype` určíme typ modulu, v ktorom sa nachádza objekt definujúci sieť v enkodéri. V kóde nižšie môžeme vidieť, z ktorého modulu sa používa funkcia `encoder_for` definujúca enkodér.

```
if args.etype == "tdnn":
    from espnet.nets.pytorch_backend.tdnn.encoders import encoder_for
else:
    from espnet.nets.pytorch_backend.rnn.encoders import encoder_for

self.enc = encoder_for(args, idim, self.subsample)
```

V module `encoders.py` je definovaná sieť TDNN(-F). Jednotlivé typy blokov sú definované v module `layers.py`, a sú totožné s implementáciou použitou pri porovnávaní TDNN(-F) sietí s nástrojom Kaldi v časti 5.2.

Formát dát pre E2E systém

Enkodér TDNN(-F) aj BLSTM dostávajú rovnaký formát vstupu. Ten pozostáva zo sekvencie vstupných akustických rámcov, ktorých počet sa odvíja od dĺžky danej sekvencie. Parametrom `maxlen-in` vieme obmedziť najdlhšiu vstupnú sekvenciu. Pomocou `maxlen-out` obmedzíme najdlhšiu výstupnú sekvenciu.

Každá trénovacia veta je po spracovaní v nástroji Kaldi, kde prebehne extrakcia príznakov, prevedená do formátu *json*. Na každú vetu môžeme pozerať aj ako na dávku o veľkosti 1. Jeden príklad z listu trénovacích viet je zobrazený nižšie. Je definovaný identifikátorom (*faem0_si1392*) a obsahuje objekty `input` a `output`.

```
"faem0_si1392": {
    "input": [
        {
        "feat": "feats.1.ark:13",
        "name": "input1",
        "shape": [ 474, 26 ]
        }
    ],
```

```
    "output": [
        {
        "name": "target1",
        "shape": [ 39, 29 ],
        "tokenid": "18 22 4 14 11 ...",
        ...
        }
    ]
}
```

Objekt `input` obsahuje premenné:

- `feat` - Odkaz na súbor typu `ark`, ktorý obsahuje vektorovú reprezentáciu akustických rámcov
- `name` - Názov vety.
- `shape` - Určuje počet rámcov X vo vete a každý je reprezentovaný vektorom dĺžky Y. $[X, Y]$

Objekt `output` obsahuje viacero premenných, pričom najpodstatnejšie sú:

- `name` - Názov vety.
- `shape` - Určuje počet výstupných tokenov X, ktoré môžu nadobúdať hodnoty od 0 po Y. $[X, Y]$
- `tokenid` - Sekvencia výstupných tokenov, ktoré reprezentujú písmená a medzery.

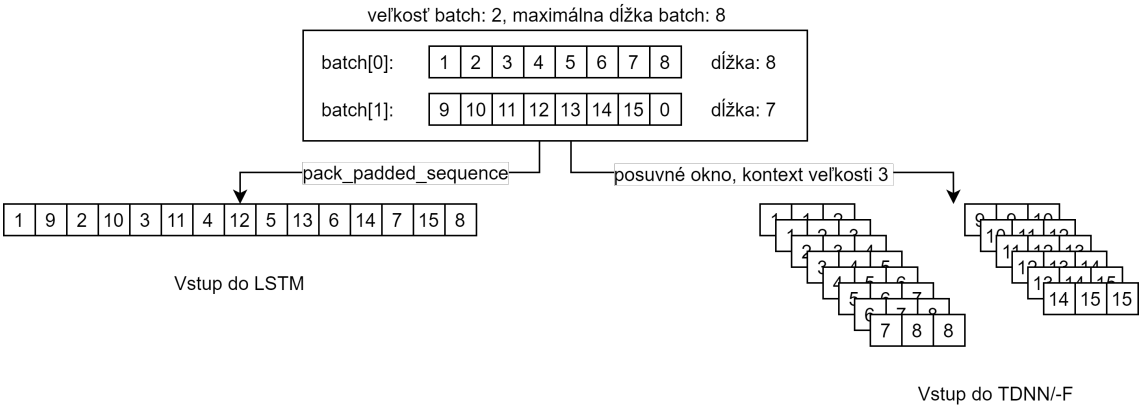
Každá vstupná veta má inú dĺžku sekvencie akustických rámcov. Pre dosiahnutie najefektívnejšieho trénovania je nutné mať dávku veľkosti väčšej ako 1, preto musíme sekvencie v jednej trénovacej dávke zarovnať nulami. K tomu nám stačí nájsť najdlhšiu sekvenciu v dávke a ostatné doplniť nulami. Pre efektívnosť je vytvorená premenná `ilen`, ktorá obsahuje sekvenciu dĺžok jednotlivých viet v dávke bez núl.

Na vstupe enkodéru je teda dávka tvaru $[B_{size}, X_{max}, D_{input}]$ a list `ilen` dĺžky B_{size}.

- B_{size} - veľkosť dávky.
- X_{max} - najdlhšia sekvencia vstupných rámcov z dávky
- D_{input} - veľkosť vektoru reprezentujúca jeden rámec.

V prípade enkodéru typu BLSTMP sa celá dávka prevedie na objekt `PackedSequence`, ktorý je v jazyku Pytorch používaný ako vstup pre všetky modely rekurentných neurónových sietí.

V prípade enkodéru typu TDNN(-F) musíme spracovávať každú sekvenciu rámcov v jednej dávke samostatne. Jednotlivé sekvencie prevedieme posuvným oknom na skupinu menších sekvencií, ktoré sa zhodujú s veľkosťou vstupu pre TDNN(-F) sieť. Na tento účel použijeme funkciu `unfold` spomínanú už v časti 5.2. Pre správne zarovnanie sekvencií musíme doplniť začiatok a koniec sekvencií kópiou prvého rámca, respektíve kópiou posledného rámca. Počet rámcov v okne určíme podľa veľkosti vstupného kontextu. Rozdiel vstupov medzi LSTM a TDNN(-F) je zobrazený na obrázku 6.8.



Obr. 6.8: Rozdiel medzi vstupom do bloku LSTM a do bloku TDNN(-F).

Vstupom do LSTM je jeden vektor reprezentujúci celú dávku ľubovoľnej veľkosti. Vrstva LSTM je preto efektívnejšia. Medzi ďalšie výhody patrí implementácia modulu Pytorch v jazyku C++, čo ešte viac zefektívňuje model oproti implementácií TDNN(-F) sietí. Tie sme si v prípade úpravy vstupu museli prispôsobiť tak, aby správne fungovali v E2E systéme rozpoznávania reči. Jednotlivé dávky sú preto upravené v cykle po jednom a sú transformované z tvaru $[X_{curr}, D_{input}]$, kde X_{curr} je dĺžka aktuálnej sekvencie vstupných rámcov, do tvaru $[B_{size}, D_{input}, C_{size}]$, kde C_{size} je veľkosť kontextu.

Ukážku konkrétneho príkladu vidíme na obrázku 6.8, kde sa vstup do TDNN(-F) tvaru [2, 8, 26] delí na dva cykly. V prvom cykle sa spracuje dávka tvaru [8, 26, 3] a v druhom cykle [6, 26, 3].

Následne výstupy enkodérov transformujeme rovnako ako v prípade výstupu z enkodéru LSTM do tvaru $[B_{size}, O_{size}]$, kde O_{size} je veľkosť výstupnej dimenzie z enkodéru.

Problémy implementácie

Pri testovaní E2E systému so zdieľaným enkodérom TDNN-F sietí sa v prípade implementácie podľa 5.2 používa funkcia pre dávkovú normalizáciu nasledovaná aktivačnou funkciou ReLU. Pri tomto nastavení sa na výstupe z daného TDNN-F bloku generovali nulové vektory, čo spôsobilo problémy pri učení celkového systému a spočiatku sme dosahovali pri použití TDNN-F enkodéru zlé výsledky.

Problém sme vyriešili úpravou existujúcej implementácie a nahradili sme aktivačnú funkciu ReLU s funkciou Tanh. Dávkovú normalizáciu sme odstránili úplne. Po tejto úprave už vektory na výstupoch neboli nulové.

Pokus zmeniť aj typ extrakcie príznakov z pôvodnej FBank metódy používanej v nástroji ESPnet na metódu MFCC, ktorú sme používali v implementácii TDNN-F sietí, skončil neúspechom. V oboch prípadoch sa pri pôvodnej architektúre s dávkovou normalizáciou generovali nulové vektory.

6.3.2 Aplikovanie nízko-dimenzionálnej faktorizácie

V časti 4 sme bližšie vysvetlili princíp faktorizácie, ktorý by nám mal pomôcť pri trénovaní modelu, a to odstránením redundancie medzi jednotlivými vektormi vo váhovej matici. Popísali sme aj funkciu *orthonormal_matrix_conv* implementovanú pomocou jazyka Pytorch v časti 5.2. Totožnú funkciu budeme používať aj v prípade trénovacieho cyklu v ESPnet.

V rámci nástroja ESPnet sa používa klasický cyklus učenia implementovaný pomocou knižnice Chainer[1]. Našim cieľom bolo modifikovať učenie pri aktualizovaní váh siete. Tento krok prebieha v objekte *Updater*. Ten je redefinovaný v jazyku Pytorch v súbore *asr.py* v časti *pytorch-backend*. Objekt obsahuje funkciu `update_core`, kde prebieha výpočet gradientov a spätnej propagácie. Podobne ako v prípade trénovania TDNN(-F) sietí doplníme tieto výpočty o úpravu váh pomocou faktorizácie, ktoré budú aplikované až po spätnej propagácii. Pre tieto potreby sme doimplementovali funkciu `update_ortho` a na základe hodnoty v parametri `etype` sa určia typy vrstiev pre úpravu váh. Za podmnienky, že bude hodnota parametra rovná `tdnn`, bude použitá funkcia `update_ortho_tdnn` upravujúca váhy v enkodéri s TDNN-F. V prípade nastavenia parametra na hodnotu `blstmp` sa zavolá funkcia `update_ortho_proj` pre upravenie projekčných lineárnych vrstiev v enkodéri s BLSTM vrstvami. Do utilít programu ESPnet sme preto doplnili modul `ortho.py`, v ktorom sú definované spomínané funkcie, a aj funkcia *orthonormal_matrix_conv*, ktorá je doplnená o vstupný parameter `ortho-alpha` na nastavovanie rýchlosti aktualizácie váhovej matice.

6.4 Experimenty

Experimenty v nástroji ESPnet sme vykonávali v Metacentre. K jeho inštalácii je možné dohľadať prehľadný návod na repozitári GitHub [50]. Problémy môžu nastať v prípade kompatibility *CUDA*[2], pretože nie všetky klastre v Metacentre majú rovnakú verziu. V našom prípade sme preto k výpočtom využívali len klaster s názvom *adan*[3].

Dáta

V našich experimentoch sme použili tri typy datových sád:

- *AN4* [3] - Databázu sme použili ako úplne základnú testovaciu sadu, ktorá obsahuje len oddelene vyslovované alfanumerické znaky a celkovo má niečo menej ako hodinu hovorenej reči. Pomocou nej sme testovali správnosť výpočtov v jednotlivých komponentoch systému.

- *TIMIT* [11] - Databáza obsahuje spolu 6300 viet. Je zložená zo 630 rečníkov, ktorí nahrali 10 viet v 8 dialektoch zo Spojených štátov amerických. Na trénovanie sa používa len okolo 5 hodín hovorenej reči, čím sa radí medzi najkratšie dátové sady. Výhodou je rýchlosť výpočtov jednotlivých experimentov a možnosť otestovať viacero variácií.

- *LibriSpeech* [32] - Databáza obsahuje celkovo 960 hodín hovorenej reči extrahovanej z audio kníh, ktoré boli zvolené pre otestovanie aj na väčších dátových sadách. Táto sada je dostupná zdarma na rozdiel od veľkej časti väčších dátových sád, za ktoré je nutné platiť. Dáta sú v tejto databáze delené na tri sady dĺžky 100, 360 a 500 hodín. V pôvodnom návode v nástroji ESPnet sa používajú všetky tri sady a prevedú sa do jednej. V našej problematike sa ale snažíme dostať k lepším výsledkom v prípade menších dátových sád, a preto sme používali len prvú sadu s dĺžkou 100 hodín, k čomu sme vhodne upravili pôvodný návod.

[1] Cyklus učenia v Chainer - https://docs.chainer.org/en/stable/guides/trainer.html
[2] Compute Unified Device Architecture
[3] klaster adan - https://wiki.metacentrum.cz/wiki/Cluster_Adan

Pomocou programu `spm_train`[4], ktorý je súčasťou nástroja ESPnet, sa trénovací text v datových sadách tokenizuje na značky s určitým počtom. Ten je zadaný ako argument na vstupe. Používa sa aj podobná utilita nástroja ESPnet: `text2token.py`. Vygenerované značky nám poslúžia na následné porovnanie vypočítanej sekvencie značiek systému s očakávanou sekvenciou značiek a tým určíme chybovosť celkového systému metrikou CER.

Počet značiek dátovej sady AN4 a TIMIT vygenerovaných utilitou `text2token.py` je 29. Pozostávajú z písmen abecedy, okrem toho aj zo špeciálnych značiek *<blank>*, *<unk>*, *<space>* a *<eos>*. Pri dátovej sade LibriSpeech sa používa program `spm_train`, ktorému sme určili vstupným argumentom počet značiek na 32. Spomínaných 29 značiek je rozšírených ešte o značky: *_THE*, *_W*, *ZZ*.

6.4.1 Enkodér typu BLSTMP

Na začiatok sme museli nájsť najlepšiu architektúru pre určenie referenčnej hodnoty s metrikou CER pre následné porovnanie s architektúrami s nízko-dimenzionálnou faktorizáciou. Toto hľadanie sme vykonávali na základe porovnávania rôzneho počtu vrstiev obojsmernej LSTM s projekčnou lineárnou vrstvou (BLSTMP) v enkodéri. Ich počet sa nastavuje pomocou parametra `elayers`. Ostatné parametre určujúce hodnoty pre enkodér, dekodér a *Attention* sú fixné a boli nastavené podľa tabuľky 6.2.

parameter	hodnota
`eprojs`	320
`eunits`	320
`dlayers`	1
`dunits`	300
`adim`	320

Tabuľka 6.2: Nastavenie základných parametrov architektúry pre porovnanie rôzneho počtu vrstiev BLSTMP. Parametre začínajúce na písmeno `e` označujú hodnoty enkodéru. Písmeno `d` označuje parametre dekodéru a písmeno `a` parametre pre *Attention*. Parametre dekodéru a *Attention* sú počas celého experimentovania nemenné.

V experimente sme použili dátovú sadu TIMIT. Jednotlivé vstupné rámce boli extrahované pomocou metódy FBank, pričom veľkosť dimenzie jedného rámca bola nastavená na hodnotu 26.

`elayers`	% CER
2	37.7
3	34.9
4	36.6
5	36.3

Tabuľka 6.3: Porovnanie počtu vrstiev BLSTMP v enkodéry a vplyv tohto parametra na úspešnosť celkovej architektúry.

Všetky architektúry boli pustené s počtom epoch 20. Parameter λ bol nastavený na hodnotu 0.5. Ako optimalizačná metóda bola použitá *adadelta*. Pokus použiť optimalizačnú

[4]program obsahuje tokenizér SentencePiece: https://github.com/google/sentencepiece

metódu *adam* skončil neúspechom, pretože jednotlivé výpočty boli menej stabilné, čo viedlo k horšiemu celkovému výsledku.

S takto nastavenou počiatočnou konfiguráciou sme už len menili parameter `elayers` a dosiahnuté výsledky sú zhrnuté v tabuľke 6.3. Najlepšie výsledky sme dostali z konfigurácie s 3 vrstvami BLSTMP v enkodéri, ktorá dosiahla najnižšiu hodnotu CER rovnú 34.9 %. Podrobnejšia architektúra enkodéru je popísaná v tabuľke 6.4.

poradie vrstvy	**typ vrstvy**		**veľkosť vstupu**	**veľkosť výstupu**
1.	BLSTMP	BLSTM	26	320
2.		lineárna	640	320
3.	BLSTMP	BLSTM	320	320
4.		lineárna	640	320
5.	BLSTMP	BLSTM	320	320
6.		lineárna	640	320

Tabuľka 6.4: Popis jednotlivých parametrov a ich hodnôt pre vrstvy v enkodéri s 3 vrstvami BLSTM, ktoré sú prepojené projekčnými vrstvami a výtvárajú tak vrstvu BLSTMP.

Pre enkodér s tromi vrstvami sme skúmali vplyv veľkosti vstupno-výstupných dimenzií a počet pamäťových jednotiek vo vrstve BLSTMP. Tie nastavujeme pomocou parametrov `eprojs` a `eunits`. Z tabuľky 6.5 môžeme vidieť, že ak zvýšime tieto hodnoty, dostaneme síce lepšie výsledky, ale za cenu spomalenia výpočtu spôsobeného najmä veľkým počtom učiacich parametrov. Po znížení týchto hodnôt dostávame horšie výsledky, pričom rýchlosť výpočtu sa už mení len mierne. Architektúra s hodnotami 1024 presahovala dostupnú pamäť.

`eprojs`	`eunits`	% CER	čas 1 epochy	počet parametrov
160	160	42.2	43 s	1 219 040
160	320	35.8	37 s	3 666 400
320	320	34.9	48 s	4 793 280
640	320	34.4	50 s	7 047 040
640	640	32.9	63 s	19 007 360

Tabuľka 6.5: Porovnanie rýchlosti, počtu učiacich parametrov enkodéru a úspešnosti celkovej architektúry v závislosti na veľkosti jednotlivých vrstiev a počtu pamäťových jednotiek v enkodéry.

Pre porovnanie vplyvu samostatných dekódovacích metód *Attention* a CTC na E2E systém sme nastavili hodnotu λ do dvoch extrémov, a to hodnotu 0 pre samostatný *Attention* a hodnotu 1 pre samostatnú CTC. Hybridná metóda odpovedá nastaveniu v intervale 0 až 1. Podľa očakávaní najlepšie dopadla hybridná metóda, viď. tabuľka 6.6. Najhoršiu úspešnosť sme dosiahli v prípade použitia CTC metódy. Samostatne sa takmer vôbec nedokázala naučiť správne predpovedať sekvencie, keďže nemá znalosť závislostí medzi vstupnou a výstupnou sekvenciou.

Vplyv faktorizácie enkodéru na systém

Z predošlých experimentov sme si ako vhodnú referenčnú architektúru enkodéru pre následné testovanie vplyvu faktorizácie na E2E systém vybrali enkodér s 3 vrstvami BLSTMP, podrobnejší rozpis vrstiev je v tabuľke 6.4. Tabuľka 6.7 zobrazuje nastavenia základných

hodnota λ	**% CER**
0.0	42.1
1.0	85.4
0.5	36.6

Tabuľka 6.6: Porovnanie extrémov hodnoty λ. Ak je $\lambda = 0$, používa sa len *Attention* metóda. Naopak len CTC metóda sa používa, ak je $\lambda = 1$.

hodnôt architektúry enkodéru. Takto nastavený enkodér dosiahol v rámci E2E systému úspešnosť 34.9%.

parameter	λ	**eprojs**	**eunits**	**subsample**
hodnota	0.5	320	320	1_2_1

Tabuľka 6.7: Nastavenie parametrov referenčnej architektúry enkodéru pre porovnávanie s enkodérmi s nízko-dimenzionálnou faktorizáciou.

Faktorizáciu budeme aplikovať na lineárne projekčné vrstvy, ktoré sú v základnej architektúre tri, vždy po každej BLSTM vrstve. Výpočet faktorizácie sme podobne ako v implementácii TDNN-F sietí aplikovali každú 4. iteráciu. Hodnota `ortho-alpha` bola nastavená na 0.125. Parameter `layer`, ktorý sme menili, môže nadobúdať sekvenciu hôdnot s číslami 0, 1, 2. Výsledky percent CER sú pre rôzne nastavenia parametra uvedené v tabuľke 6.8.

layer	**% CER**
0	36.4
1	35
2	34.8
0_1	37.5
1_2	35.5
0_1_2	36.2

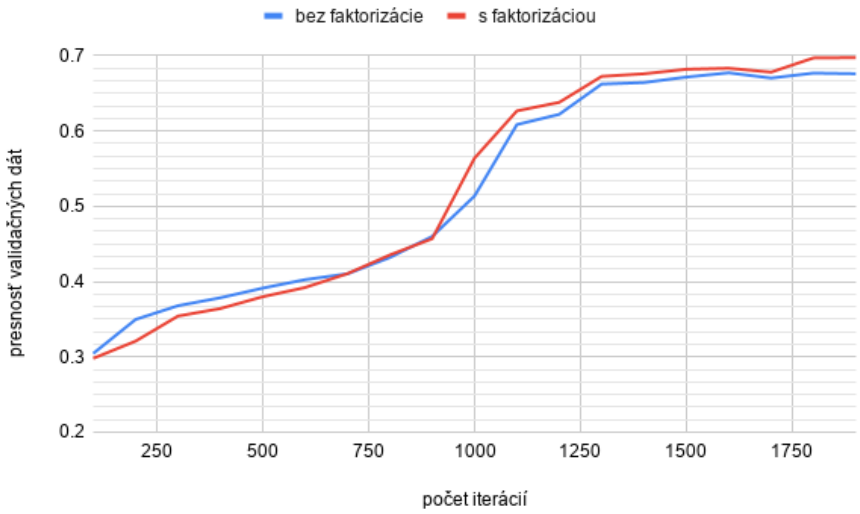
Tabuľka 6.8: Porovnanie aplikácie faktorizácie na rôzne vrstvy v 3 vrstvom enkodéri BLSTM pri veľkosti 320.

Výsledkom experimentov bolo zlepšenie úspešnosti len v prípade faktorizácie poslednej projekčnej vrstvy. Vývoj presnosti systému s 3 vrstvami BLSTMP v enkodéri a s faktorizáciou poslednej lineárnej vrstvy je zobrazený v grafe 6.9. Výpočet presnosti na validačných dátach prebieha každú stú iteráciu. Presnosť systému s faktorizáciou približne do polovice učenia je lepší, ale v ďalších iteráciách sa už rozdiel vyrovnáva.

Následne sme skúsili podobné experimenty zopakovať aj na enkodéri s parametrami `eprojs` a `eunits` s hodnotou 640, kde je úspešnosť modelu bez faktorizácie 32,9 %. Porovnanie enkodérov s faktorizáciou je v tabuľke 6.9. Experimenty potvrdili zlepšenie výsledku len pri faktorizácii poslednej projekčnej vrstvy.

Testovanie faktorizácie váhových matíc v enkodéri

K otestovaniu správneho procesu pri výpočte nízko-dimenzionálnej faktorizácie na váhové matice sme si vizualizovali váhové matice poslednej projekčnej lineárnej vrstvy pred výpočtom a po výpočte faktorizácie takmer v polovici procesu učenia v grafe 6.10. Tento graf



Obr. 6.9: Porovnanie priebehu učenia medzi systémom s 3 vrstvým enkodérom s faktorizáciou aplikovanou na poslednú projekčnú vrstvu a enkodérom bez faktorizácie. Červená čiara ukazuje priebeh zlepšovania presnosti systému s faktorizáciou každú stú iteráciu na validačných dátach. Modrá naopak priebeh bez faktorizácie.

`layer`	% CER
0	33.7
2	32.7
0_1_2	33.6

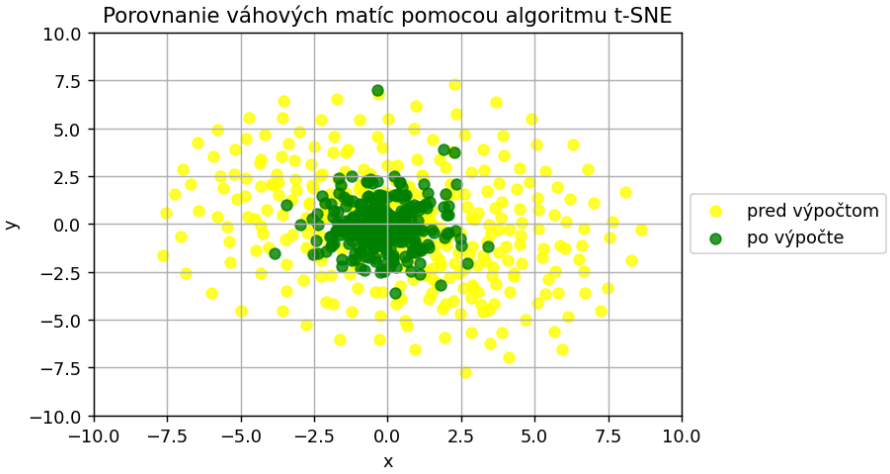
Tabuľka 6.9: Porovnanie aplikácie faktorizácie na rôzne vrstvy v 3 vrstvom enkodéri BLSTM pri veľkosti 640.

má s grafom 4.1 podobné črty, čo naznačuje správnosť výpočtov. Správnosť výpočtu nám navyše potvrdzuje aj hodnota *frobenius norm*, ktorá nadobúda hodnoty blízke nule.

Neočakávané výsledky sme dostali v prípade nastavenia nulovej hodnoty pre rýchlosť konvergovania matice k jej semi-ortogonálnemu tvaru. Z rovnice 4.2 je definovaný učiaci faktor ako konštanta rovná hodnote 0.125, rovnako ako v implementácii premenná `update_speed` v časti 5.2. Túto hodnotu v nástroji ESPnet nastavíme pomocou `ortho-alpha`. Pri hodnote 0 sa úspešnosť celkového systému v priemere zhoršila o desatiny percenta.

Pri debugovaní procesu sme zistili, že sa matica pred výpočtom a po výpočte nemení. Nedospeli sme preto k záveru, prečo sa menia výsledky, aj keď sa váhová matica nemení. Pre porovnanie vplyvu konštanty na úspešnosť systému sme urobili experimenty s rôznymi hodnotami konštanty zobrazenými v tabuľke 6.10. Podľa očakávaní sme najlepšiu úspešnosť dostali s hodnotou 0.125.

Podobne sme otestovali aj sumu S_{all} kosínusových vzdialeností všetkých vektorov po učení vo váhových maticiach. Pri učení bez faktorizácie mali váhové matice priemerne hodnotu S_{all} v desiatkach, s faktorizáciou je hodnota S_{all} priemerne v jednotkách tisícin, takmer 10000-krát menšia.



Obr. 6.10: Porovnanie redukcie váhových matíc poslednej projekčnej vrstvy pomocou algoritmu t-SNE. Žltou farbou je označená váhová matica pred výpočtom faktorizácie. Zelenou farbou je označená matica po výpočte.

`ortho-alpha`	% CER
0.0	35.2
0.0625	35.3
0.125	34.9
0.25	35.1

Tabuľka 6.10: Porovnanie konštanty `ortho-alpha`, ktorá má vplyv na rýchlosť konvergovania matice k jej semi-ortogonálnemu tvaru.

Extrakcia príznakov

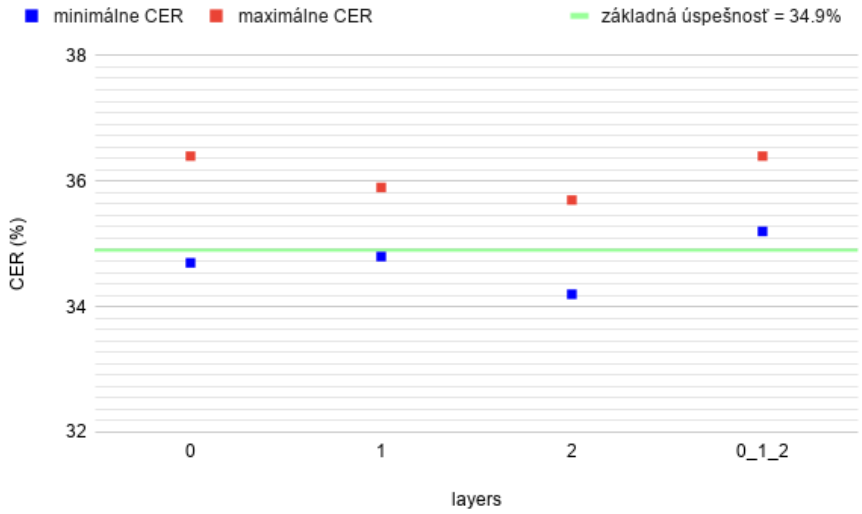
Vyskúšali sme aj ďalšiu metódu extrakcie príznakov MFCC. Veľkosť rámcov sme nastavili na 40. Pri natrénovaní systému referenčnej architektúry s príznakmi MFCC sme dosiahli výrazné zhoršenie o viac ako 5%, viď. tabuľka 6.11. Rozhodli sme sa preto v ďalších experimentoch používať len metódu FBank.

extrakcia príznakov	veľkosť príznakov	% CER
MFCC	40	40.4
FBank	26	34.9

Tabuľka 6.11: Porovnanie úspešnosti systému pri rôznom type extrakcie príznakov v referenčnej architektúre systému s 3-vrstvým BLSTMP enkodérom.

Vylepšenie faktorizácie

Následné skúmanie bolo zamerané na zlepšenie výsledku systému. Nastavením rôznych hodnôt parametrov `ortho-iteration` a `ortho-epoch` sme dosiahli rozdielne úspešnosti, v niektorých prípadoch až v rozmedzí 1,7%. Z toho môžeme konštatovať, že na úspešnosť systému vplýva počet výpočtov faktorizácie a pravdepodobne aj frekvencia výpočtov.



Obr. 6.11: Porovnanie minimálnych (modré štvorce) a maximálnych (červené štvorce) hodnôt CER dosiahnutých pri faktorizácii jednotlivých vrstiev pri rôznych nastaveniach `ortho-iteration` a `ortho-epoch`. Základná hodnota bez faktorizácie je označená zelenou čiarou.

Vhodným nastavením spomínaných parametrov sa dá dosiahnuť lepšia úspešnosť ako je vyobrazené na grafe 6.11. Najlepšiu úspešnosť sme dosiahli opäť v prípade poslednej projekčnej vrstvy, kde sa CER rovná 34.2%. Parametre pre dosiahnutie tohto najlepšieho výsledku pre rôzne hodnoty `layer` sú zhrnuté v tabuľke 6.12. Mierne zlepšenie úspešnosti sme dosiahli aj v prípade prvej a druhej projekčnej vrstvy oproti pôvodnému testovaniu po každých 4 iteráciách. Najlepšie výsledky pri aplikácií faktorizácie na dané vrstvy dosiahneme pri rôznych hodnotách parametrov `ortho-iteration` a `ortho-epoch`, to zároveň predstavuje rôzny počet výpočtov faktorizácie, nedá sa preto určiť univerzálne najlepšie nastavenie.

`layer`	`ortho-iteration`	`ortho-epoch`	% CER	počet výpočtov faktorizácie
0	3	2	34.7	95
1	2	2	34.8	158
2	3	1	34.2	193
0_1_2	2	2	35.2	158

Tabuľka 6.12: Optimálne nastavenie parametrov `ortho-iteration` a `ortho-epoch` pre aplikáciu faktorizácie na jednotlivé vrstvy, nastavené pomocou hodnoty `layer`. Zároveň sú zobrazené počty výpočtov faktorizácie počas učenia pri 20 epochách.

V ďalšom kroku sme otestovali vplyv faktorizácie na úspešnosť systému s 4 a 5 vrstvým BLSTMP enkodérom, ktoré majú rovnaké parametre aj pre pridané vrstvy. Podobne ako v predchádzajúcich prípadoch sme spustili učenie s rôznymi parametrami `layer`, `ortho-iteration` a `ortho-epoch`. Najlepšie výsledky sme opäť dosiahli pri aplikácii faktorizácii posledných vrstiev. Oproti základnému modelu bez faktorizácie sme vždy dosiahli mierne zlepšenia. Rozdiely sú zobrazené v tabuľke 6.13.

počet vrstiev	referečný % CER	% CER po faktorizácií	rozdiel
3	34.9	34.2	-0.7
4	36.6	35.9	-0.7
5	36.3	35.7	-0.4

Tabuľka 6.13: Porovnanie CER po aplikovaní faktorizácie v enkodéri BLSTMP s dátami TIMIT. Referenčný CER určuje úspešnosť systému bez faktorizácie. Rozdiel popisuje, o koľko percent sa zlepšil systém pri vhodnom nastavení po aplikácií faktorizácie.

LibriSpeech

Aplikáciu faktorizácie sme otestovali aj na väčšej dátovej sade LibriSpeech. Oproti dátam s TIMIT, ktorý má 5 hodín, obsahuje táto sada takmer 100 hodín trénovacích dát. Pri extrakcii príznakov sme na rozdiel od TIMIT použili väčšiu dimenziu veľkosti 83 a totožnú architektúru systému so 4 vrstvami BLSTMP v enkodéri. S týmito nastaveniami sme dosiahli referenčnú hodnotu systému CER = 6.1%.

Po testovaní aplikácie faktorizácie na rôzne vrstvy (nastavené pomocou `layer`) sme aj v tomto prípade dosiahli najlepšie výsledky pri aplikácií faktorizácie na poslednú vrstvu, a to zlepšením o 0.5%. Najhoršie výsledky sme dosiahli v prípade faktorizácie všetkých vrstiev systému, kde sa v niektorých prípadoch úspešnosť zhoršila až takmer o 2%.

S predpokladom, že najlepšie výsledky poskytuje faktorizácia posledných projekčných vrstiev sme vyskúšali zvýšiť a znížiť počet vrstiev enkodéru. V oboch prípadoch sme dosiahli zlepšenie úspešnosti. Porovnanie v tabuľke 6.14.

počet vrstiev	referečný % CER	% CER po faktorizácií	rozdiel
3	5.7	5.5	-0,2
4	6.1	5.6	-0,5
5	5.8	5.6	-0,2

Tabuľka 6.14: Porovnanie CER s dátami Librispeech po aplikovaní faktorizácie na posledné vrstvy projekčných vrstiev v enkodéroch BLSTMP s rôznym počtom vrstiev. Referenčný CER určuje úspešnosť systému bez faktorizácie. Rozdiel popisuje zlepšenie systému po aplikácií faktorizácie pri vhodnom nastavení.

6.4.2 Enkodér typu TDNN

V tejto časti práce je našim cieľom nájsť čo najlepšie nastavenie architektúry typu TDNN, ktoré by dosahovalo podobné alebo lepšie výsledky ako enkodér typu BLSTMP. Na testovanie funkčnosti systému sme používali minimálnu datovú sadu *AN4*. Vypísali sme si jednotlivé váhové matice a zistili sme, že ,s výnimkou prvej vrstvy, sú všetky hodnoty matice nulové. Sieť sa nebola schopná úspešne učiť a dosahovala CER viac než 70%, a to aj pri rôznych nastaveniach architektúry enkodéru s TDNN(-F). Chybu sme napravili úpravou blokov, popísanou v časti 6.3.1. Po tejto úprave už matice neboli nulové a hodnota CER klesla k 10%.

Počet vrstiev TDNN(-F)

Správne nastavenie počtu vrstiev TDNN(-F) sme hľadali experimentovaním s dátovou sadou TIMIT. Počet vrstiev TDNN(-F) v enkodéri musí byť minimálne 2. Prvá vrstva má fixný typ TDNN a kontext, druhá má fixný len kontext a môžeme nastaviť jej typ. Fixná časť je vyznačená aj v tabuľke 6.1. Sieť má rovnaké nastavenie vstupných hodnôt, a to príznaky FBank veľkosti 26. Veľkosť vstupu a výstupu jednotlivých blokov je nastavená parametrom `etdnns` na hodnotu 320 a hodnota $\lambda = 0.5$. Počet epoch je nastavený na 20.

veľkosť kontextového okna	`strides`	**% CER**	
		TDNN	TDNN-F
13	3	49.1	50.9
19	3_3	50.3	46.6
25	3_3_3	43.1	46.4
31	3_3_3_3	45.7	46.3
37	3_3_3_3_3	52.6	45.1
43	3_3_3_3_3_3	50.2	46.4

Tabuľka 6.15: Porovnanie úspešnosti systému s enkodérom TDNN a TDNN-F na testovacích dátach pri rôznych nastaveniach parametra `strides`, ktorý zároveň určuje veľkosť kontextu pre vstup do enkodéru.

Nastavením rôznych hodnôt `strides`, ktoré určujú počet vrstiev a zároveň symetrický kontext, sme pri použití vrstiev TDNN dosiahli najlepšie výsledky pri `strides=3_3_3` s CER = 43.1% na testovacích dátach. U vrstiev TDNN-F s rovnakými hodnotami `strides` a s veľkosťou *bottlenecku* nastavenou parametrom `bndim=96` sme najlepší výsledok CER=45.1% dosiahli s parametrom `strides=3_3_3_3_3`. Porovnania sú zobrazené v tabuľke 6.15. Opäť, ako v prípade faktorizácie BLSTMP vrstiev, sme najlepší výsledok dosiahli rôznym nastavením parametrov `ortho-iteration` a `ortho-epoch`.

Testovanie faktorizácie váhových matíc v enkodéri

Pre otestovanie správneho procesu pri výpočte nízko-dimenzionálnej faktorizácie na váhové matice sme postupovali rovnako ako pri enkodéri typu BLSTMP. Porovnali sme ale testovaním sumu S_{all} kosínusových vzdialeností všetkých vektorov vo váhovej matici s *bottleneckom*. Matica mala po procese učenia v jednotlivých vrstvách TDNN-F hodnotu S_{all} blízku nule, čo naznačuje správnosť výpočtu. Porovnanie stavu matíc pred a po výpočte faktorizácie sme neotestovali, pretože faktorizácia sa aplikuje na vrstvy TDNN-F už od začiatku učenia.

Porovnanie počtu učiacich parametrov

Ďalej sme najlepšiemu modelu TDNN (viď tabuľka 6.4) rozšírili veľkosť vstupu a výstupu nastavením `etdnns`. Po každom zväčšení sme dosiahli lepšiu úspešnosť CER s tým, že je dokonca porovnateľná s referenčným 3-vrstvovým BLSTMP enkodérom bez faktorizácie. TDNN model má nevýhody vo forme veľkého počtu učiacich parametrov, a to takmer 4-krát väčší počet ako referečný model, čo spôsobuje väčšie pamäťové nároky na grafickú kartu. Údaje sú zobrazené v tabuľke 6.16. Čas výpočtov sme v porovnaní nezahrnuli, a to z dôvodu rozdielnej implementácie medzi nami vytvorenými vrstvami TDNN(-F) a LSTM, ktoré sú optimalizované a implemetované v jazyku C++, a tým pádom sú výrazne efektívnejšie.

typ vrstiev	etdnns	bndim	% CER	počet parametrov
TDNN	640	-	39	5 008 000
TDNN	1280	-	34.6	19 846 400
TDNN-F	640	96	42.8	1 076 224
TDNN-F	640	256	40.7	2 715 264
TDNN-F	1280	96	42.7	2 152 064
TDNN-F	1280	256	42	5 429 504
BLSTMP	320	-	34.9	4 793 280

Tabuľka 6.16: Porovnanie úspešnosti enkodérov typu TDNN a TDNN-F z tabuľky 6.15 s parametrom `strides=3_3_3` rozšíreným o veľkosť vstupu a výstupu nastavením `etdnns` a v prípade TDNN-F aj nastavením `bndim`. Zároveň je zobrazený počet učiacich parametrov jednotlivých enkodérov v porovnaní s referečným 3-vrstvovým BLSTM enkodérom bez faktorizácie (BLSTMP).

Následne sme porovnali aj vrstvy TDNN-F, ktorým sme rozšírili veľkosť `etdnns` a zväčšil aj veľkosť *bottlenecku* o viac ako dvojnásobok a to na hodnotu 256. Výsledky sú v tabuľke 6.16. Z týchto hodnôt je zrejmá výrazná redukcia počtu parametrov pri použití vrstiev TDNN-F. Nedosahujeme ale také hodnoty CER ako v prípade vrstiev TDNN alebo BLSTMP. Dokonca ani enkodér s vrstvami TDNN-F s počtom parametrov väčším ako enkodér BLSTMP nedosahuje podobnú úspešnosť.

Hybridný enkodér

Po analýze hodnotiacich funkcií v systémoch, ktoré dosiahli podobnú úspešnosť, sme zistili, že najlepší enkodér s vrstvami BLSTMP má hodnotu hodnotiacej funkcie *Attention* menšiu ako optimálny enkodér TDNN s `etdnns=320`. Tento rozdiel môže byť spôsobený najmä tým, že *Attention* mechanizmus bol navrhnutý ako zlepšenie pre systémy s rekurentnými neurónovými sieťami.

typ vrstiev enkodéru	hodnota *Attention*	% CER
TDNN	57	43.1
TDNN + BLSTMP	22	33
BLSTMP	26	34.9

Tabuľka 6.17: Porovnanie hodnotiacej funkcie *Attention* troch typov enkodérov: TDNN, BLSTMP a TDNN s poslednou vrstvou BLSTMP.

Vytvorili sme preto hybridný enkodér rozšírením TDNN(-F) vrstiev o jednu BLSTMP vrstvu (viď obrázok 6.7) a nastavením parametra `lstm=true`. Táto vrstva je umiestnená ako posledná za vrstvami TDNN(-F). Rozšírenie sme aplikovali na enkodér typu TDNN(-F) s hodnotami `etdnns=320`, `eprojs=320`, `strides=3_3_3` a v prípade TDNN-F aj `bndim=96` a nastavením parametrov `ortho-iteration` a `ortho-epoch` v rozmedzí od 1 do 5. Týmto prístupom sa nám podarilo znížiť hodnotu hodnotiacej funkcie *Attention* (viď. tabuľka 6.17). Počet učiacich parametrov je oproti enkodéru len s BLSTMP vrstvami, ktorý má 4 793 280 parametrov, výrazne menší (viď. tabuľka 6.18) a v prípade TDNN vrstiev sme dosiahli lepšie percentuálne hodnoty CER o takmer 2%. Použitím hybridného enkodéru sme v oboch

typoch TDNN a TDNN-F dosiahli výrazné zlepšenie percentuálnej hodnoty CER o viac ako 10%. Výsledky sú v tabuľke 6.18.

typ vrstiev	**% CER**		**počet parametrov**
	bez BLSTMP	s BLSTMP	s BLSTMP
TDNN	43.1	33	3 123 840
TDNN-F	46.4	35.8	2 386 944

Tabuľka 6.18: Porovnanie úspešnosti TDNN(-F) vrstiev na testovacích dátach s BLSTMP vrstvou a bez nej. Zobrazený je aj počet parametrov enkodéru TDNN(-F) s vrstvou BLSTMP.

Pri použití vrstvy typu BLSTMP došlo k vzniku projekčnej vrstvy aj v hybridnom enkodéri. Nastavením parametra `layer=last` aplikujeme výpočet faktorizácie na túto vrstvu. Rôznym nastavením parametrov `ortho-iteration` a `ortho-epoch` sme v prípade TDNN vrstiev a ani TDNN-F vrstiev nedosiahli zlepšenie v porovnaní s výsledkami z tabuľky 6.18.

Našou ďalšou snahou bolo nájsť čo najlepšie nastavenia enkodéru za podmienok zlepšenia oproti referenčnému enkodéru, zlepšenia efektívnosti modelu a zníženia množstva učiacich parametrov. Toto sme dosiahli rozšírením vrstiev TDNN(-F) pomocou parametra `etdnns` o dvojnásobok a v prípade TDNN-F aj rozšírením `bndim`. Ponechali sme pôvodnú hodnotu `eprojs=320`, pretože má výrazný vplyv na počet učiacich parametrov. Lepšie hodnoty CER sme dosiahli s rozšíreným hybridným enkodérom TDNN-F. Oproti referenčnému enkodéru vystavanému len z vrstiev BLSTMP sme dosiahli 3.2% zlepšenie. TDNN-F enkodér má dokonca menší počet učiacich parametrov. Pri hodnote `bndim=96` je ich počet takmer dvakrát menší a zároveň je zlepšená úspešnosť o 2.4%. S TDNN vrstvami sme nedosiahli žiadne významné zlepšenie pri rozšírení. Porovnanie je v tabuľke 6.19.

typ vrstiev	**eprojs**	**etdnns**	**bndim**	**% CER**	**počet parametrov**
TDNN+BLSTMP	320	640	-	35	6 241 280
TDNN-F+BLSTMP	320	640	96	32.5	2 893 184
TDNN-F+BLSTMP	320	640	256	31.7	4 481 024
BLSTMP	320	-	-	34.9	4 793 280

Tabuľka 6.19: Porovnanie úspešnosti systému a počtu učiacich parametrov v enkodéri: medzi enkodérom typu TDNN(-F) s poslednou vrstvou BLSTMP, pri zväčšení veľkosti vrstiev `etdnns` a pri TDNN-F aj `bndim` s enkodérom typu BLSTMP.

LibriSpeech

Aplikáciu enkodéru typu TDNN(-F) sme otestovali aj na väčšej dátovej sade LibriSpeech. Dáta sa oproti databáze TIMIT líšia väčším počtom trénovacích dát. Tie sú pred trénovaním extrahované rovnakou metódou FBank, ale generujeme vektory o veľkosti 83, nie 26 ako v prípade TIMIT. Ako referenčný model na porovnanie sme použili 3-vrstvový enkodér typu BLSTMP (architektúra je popísaná v tabuľke 6.4), ktorý dosiahol výsledok CER=5.7%. Ten budeme porovnávať s hybridným enkodérom typu TDNN(-F). Využili sme tabuľku 6.19, kde majú hybridné enkodéry TDNN(-F) s poslednou vrstvou BLSTMP hodnoty parametrov `eprojs=320`, `etdnns=640` a pri TDNN-F aj `bndim=96`. Takto nastavené enkodéry dosahujú lepšie alebo podobné hodnoty CER oproti enkodéru typu BLSTMP na databáze TIMIT.

Otestovali sme preto vplyv väčšej dátovej sady na identickú architektúru E2E systému z predošlých experimentov. V oboch prípadoch hybridného enkodéru sme dosiahli takmer dvojnásobné zhoršenie hodnoty CER. V tabuľke 6.20 sú výsledky daných experimentov na dátovej sade LibriSpeech.

typ vrstiev	`eprojs`	`etdnns`	`bndim`	**% CER**
TDNN+BLSTMP	320	640	-	7.8
TDNN-F+BLSTMP	320	640	96	10.2
BLSTMP	320	-	-	5.7

Tabuľka 6.20: Porovnanie úspešnosti systému natrénovanej na dátovej sade LibriSpeech s enkodérom typu TDNN(-F) s poslednou vrstvou BLSTMP a enkodéru len s vrstvami BLSTMP.

Kapitola 7

Záver

Úlohou tejto práce bolo implementovať a otestovať nízko-dimenzionálnu faktorizáciu na hybridných i E2E sieťach, kde sa jedná o stále neotestovanú kombináciu.

V prvej časti práce sme sa dozvedeli detaily ohľadom trénovania neurónových sietí pomocou konvolučných neurónových sietí a jej konkrétnej architektúry v podobe TDNN, respektíve TDNN-F sietí, ktoré sme implementovali v jazyku Pytorch. Cieľom bolo dosiahnuť, čo najlepšie výsledky, aby sa úspešnosť vyrovnala už implementovanej sieti v nástroji Kaldi. Táto úloha sa podarila a výsledky sme dosiahli takmer identické. Nepostačujúca je ale rýchlosť trénovania v jazyku Pytorch, ktorá je v porovnaní s nástrojom Kaldi priemerne 10-krát pomalšia. V rámci rozdielu sietí TDNN a TDNN-F v Pytorch sme v druhej menovanej dosiahli mierne zlepšenie úspešnosti, ale zároveň výraznú redukciu učiacich parametrov, vďaka čomu sa zmenšili pamäťové nároky a zrýchlilo sa trénovanie.

V ďalšej časti sme predstavili problematiku E2E systémov a s ňou spojenú hybridnú metódu dekódovania, ktorá vznikne spojením metódy *Attention* a CTC. V tomto systéme sme pozorovali vplyv aplikácie nízko-dimenzionálnej faktorizácie na projekčné vrstvy v enkodéri. Experimentmi s databázou TIMIT sme dosiahli najlepšie hodnoty CER pri aplikácii faktorizácie na poslednú projekčnú vrstvu v enkodéri. Dosiahli sme zlepšenie hodnoty CER o 0.7%. Na väčšej dátovej sade LibriSpeech(100h) sme dosiahli zlepšenie len o 0.2%. V niektorých experimentoch na rovnakej architektúre sme dosiahli zhoršenie hodnôt CER. Za následok to má frekvencia výpočtov faktorizácie, ktorá výrazne ovplyvňuje výsledný stav systému. Rovnako nestabilný je aj enkodér vytvorený z vrstiev TDNN-F zakončený jednou vrstvou typu BLSTMP. Vďaka tomuto typu enkodéru sme experimentmi na dátovej sade TIMIT dosiahli pri správnom nastavení frekvencie výpočtov faktorizácie zlepšenie hodnôt CER o 3.2%. Tento model mal ale podobný počet učiacich parametrov ako referenčný 3-vrstvový BLSTMP enkodér. Redukovaním počtu učiacich parametrov takmer o polovicu pomocou vrstiev TDNN-F v architektúre enkodéru sme dosiahli zlepšenie CER o 2.4%. Na väčšej dátovej sade LibriSpeech(100h) sme však na podobných architektúrach nedosiahli zlepšenie hodnoty CER. Použitie tohto typu enkodéru v E2E systéme je preto vhodné pri trénovaní systému s menším množstvom dát.

Literatúra

[1] *Convolution arithmetic tutorial* [online], 17. novembra 2017 [cit. 2020-03-26]. Dostupné z: http://deeplearning.net/software/theano/tutorial/conv_arithmetic.html.

[2] Abigail See, M. L. *Machine Translation, Sequence-to-sequence and Attention* [online]. 2019 [cit. 2020-03-30]. Dostupné z: http://web.stanford.edu/class/cs224n/slides/cs224n-2020-lecture08-nmt.pdf.

[3] Acero, A. *Acoustical and environmental robustness in automatic speech recognition*. Springer Science & Business Media, 2012.

[4] Amodei, D., Anubhai, R., Battenberg, E., Case, C., Casper, J. et al. Deep Speech 2: End-to-End Speech Recognition in English and Mandarin. *CoRR*. 2015, abs/1512.02595. Dostupné z: http://arxiv.org/abs/1512.02595.

[5] Anusuya, M. A. a Katti, S. K. Speech Recognition by Machine, A Review. *CoRR*. 2010, abs/1001.2267. Dostupné z: http://arxiv.org/abs/1001.2267.

[6] Chau. *TDNN* [https://github.com/cvqluu/TDNN]. GitHub, 2019.

[7] Chorowski, J., Bahdanau, D., Cho, K. a Bengio, Y. End-to-end Continuous Speech Recognition using Attention-based Recurrent NN: First Results. *CoRR*. 2014, abs/1412.1602. Dostupné z: http://arxiv.org/abs/1412.1602.

[8] Chorowski, J., Bahdanau, D., Serdyuk, D., Cho, K. a Bengio, Y. Attention-Based Models for Speech Recognition. *CoRR*. 2015, abs/1506.07503. Dostupné z: http://arxiv.org/abs/1506.07503.

[9] Donckt, J. V. D. *TDNN* [https://github.com/jonasvdd/TDNN]. GitHub, 2019.

[10] Gajdár, M. *Nové techniky v oblasti trénování neuronových sítí - Connectionist temporal classification*. Brno, CZ, 2017. Bakalárska práca. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.vutbr.cz/studenti/zav-prace/detail/106129.

[11] Garofolo, J., Lamel, L., Fisher, W., Fiscus, J., Pallett, D. et al. TIMIT Acoustic-phonetic Continuous Speech Corpus. *Linguistic Data Consortium*. November 1992.

[12] Gers, F. A., Schmidhuber, J. a Cummins, F. Learning to forget: Continual prediction with LSTM. IET. 1999.

[13] Goodfellow, I., Bengio, Y. a Courville, A. *Deep Learning*. MIT Press, 2016. 200-220 s. ISBN 9780262035613. http://www.deeplearningbook.org.

[14] Graves, A. a Jaitly, N. Towards end-to-end speech recognition with recurrent neural networks. In: *International conference on machine learning*. 2014, s. 1764–1772.

[15] Graves, A., Jaitly, N. a Mohamed, A.-r. Hybrid speech recognition with deep bidirectional LSTM. In: IEEE. *2013 IEEE workshop on automatic speech recognition and understanding*. 2013, s. 273–278.

[16] Gupta, V., Kenny, P., Ouellet, P. a Stafylakis, T. I-vector-based speaker adaptation of deep neural networks for french broadcast audio transcription. In: IEEE. *2014 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. 2014, s. 6334–6338.

[17] He, K., Zhang, X., Ren, S. a Sun, J. Deep residual learning for image recognition. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, s. 770–778.

[18] Hochreiter, S. a Schmidhuber, J. Long short-term memory. *Neural computation*. MIT Press. 1997, zv. 9, č. 8, s. 1735–1780.

[19] Huang, X., Acero, A. a Hon, H. *Spoken Language Processing: A Guide to Theory, Algorithm, and System Development*. 1. vyd. USA: Prentice Hall PTR, 2001. ISBN 9780130226167. Dostupné z: https://books.google.cz/books?id=reZQAAAAMAAJ.

[20] Ioffe, S. a Szegedy, C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR*. 2015, abs/1502.03167. Dostupné z: http://arxiv.org/abs/1502.03167.

[21] Kim, S., Hori, T. a Watanabe, S. Joint CTC-Attention based End-to-End Speech Recognition using Multi-task Learning. *CoRR*. 2016, abs/1609.06773. Dostupné z: http://arxiv.org/abs/1609.06773.

[22] Kingma, D. P. a Ba, J. Adam: A method for stochastic optimization. *ArXiv preprint arXiv:1412.6980*. 2014.

[23] Lawrence, S., Giles, C. L., Ah Chung Tsoi a Back, A. D. Face recognition: a convolutional neural-network approach. *IEEE Transactions on Neural Networks*. 1997, zv. 8, č. 1, s. 98–113.

[24] Liu, X., Liu, S., Sha, J., Yu, J., Xu, Z. et al. Limited-memory bfgs optimization of recurrent neural network language models for speech recognition. In: IEEE. *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2018, s. 6114–6118.

[25] Maaten, L. v. d. a Hinton, G. Visualizing data using t-SNE. *Journal of machine learning research*. 2008, zv. 9, Nov, s. 2579–2605.

[26] Mikolov, T., Karafiát, M., Burget, L., Černockỳ, J. a Khudanpur, S. Recurrent neural network based language model. In: *Eleventh annual conference of the international speech communication association*. 2010.

[27] Mohri, M., Pereira, F. a Riley, M. Weighted Finite-State Transducers in Speech Recognition. *Comput. Speech Lang.* GBR: Academic Press Ltd. január 2002, zv. 16, č. 1, s. 69–88. DOI: 10.1006/csla.2001.0184. ISSN 0885-2308. Dostupné z: https://doi.org/10.1006/csla.2001.0184.

[28] Nasr, G. E., Badr, E. A. a Joun, C. Cross Entropy Error Function in Neural Networks: Forecasting Gasoline Demand. In: Haller, S. M. a Simmons, G., ed. *FLAIRS Conference*. AAAI Press, 2002, s. 381–384. ISBN 1-57735-141-X. Dostupné z: http://dblp.uni-trier.de/db/conf/flairs/flairs2002.html#NasrBJ02.

[29] Navarro, G. A guided tour to approximate string matching. *ACM computing surveys (CSUR)*. ACM New York, NY, USA. 2001, zv. 33, č. 1, s. 31–88.

[30] Nguyen, V. H. An End-to-End Model for Vietnamese Speech Recognition. *2019 IEEE-RIVF International Conference on Computing and Communication Technologies (RIVF)*. 2019, s. 1–6.

[31] Ochiai, T., Watanabe, S., Hori, T. a Hershey, J. Multichannel End-to-end Speech Recognition. Marec 2017.

[32] Panayotov, V., Chen, G., Povey, D. a Khudanpur, S. Librispeech: an asr corpus based on public domain audio books. In: IEEE. *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2015, s. 5206–5210.

[33] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E. et al. Automatic differentiation in pytorch. 2017.

[34] Peddinti, V., Povey, D. a Khudanpur, S. A time delay neural network architecture for efficient modeling of long temporal contexts. In: *Sixteenth Annual Conference of the International Speech Communication Association*. 2015.

[35] Povey, D. *Data preparation* [online]. [cit. 2020-03-26]. Dostupné z: http://kaldi-asr.org/doc/data_prep.html.

[36] Povey, D., Cheng, G., Wang, Y., Li, K., Xu, H. et al. Semi-Orthogonal Low-Rank Matrix Factorization for Deep Neural Networks. In:.

[37] Povey, D., Ghoshal, A., Boulianne, G., Burget, L., Glembek, O. et al. The Kaldi Speech Recognition Toolkit. In: *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding*. IEEE Signal Processing Society, December 2011. ISBN 978-1-4673-0366-8. IEEE Catalog No.: CFP11SRW-USB.

[38] Ravanelli, M., Brakel, P., Omologo, M. a Bengio, Y. Light Gated Recurrent Units for Speech Recognition. *IEEE Transactions on Emerging Topics in Computational Intelligence*. 2018, zv. 2, s. 92–102.

[39] Robbins, H. E. A Stochastic Approximation Method. *Annals of Mathematical Statistics*. 2007, zv. 22, s. 400–407.

[40] Robinson, T., Hochberg, M. a Renals, S. IPA: improved phone modelling with recurrent neural networks. In: *Proceedings of ICASSP '94. IEEE International Conference on Acoustics, Speech and Signal Processing*. 1994, i, s. I/37–I/40 vol.1.

[41] Salvador, S. a Chan, P. Toward accurate dynamic time warping in linear time and space. *Intelligent Data Analysis*. IOS Press. 2007, zv. 11, č. 5, s. 561–580.

[42] Shaikh, F. *Introduction to Gradient Descent Algorithm (along with variants) in Machine Learning* [online]. 2017 [cit. 2020-03-26]. Dostupné z: https://www.analyticsvidhya.com/blog/2017/03/introduction-to-gradient-descent-algorithm-along-its-variants/.

[43] Simonyan, K. a Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR*. 2015, abs/1409.1556.

[44] Soufifar, M., Kockmann, M., Burget, L., Plchot, O., Glembek, O. et al. IVector Approach to Phonotactic Language Recognition. In:. Január 2011, s. 2913–2916.

[45] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. a Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*. JMLR. org. 2014, zv. 15, č. 1, s. 1929–1958.

[46] Tokui, S., Oono, K., Hido, S. a Clayton, J. Chainer: a next-generation open source framework for deep learning. In: *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*. 2015, sv. 5, s. 1–6.

[47] Waibel, A., Hanazawa, T., Hinton, G., Shikano, K. a Lang, K. J. Phoneme recognition using time-delay neural networks. *IEEE transactions on acoustics, speech, and signal processing*. IEEE. 1989, zv. 37, č. 3, s. 328–339.

[48] Wang, D., Wang, X. a Lv, S. An Overview of End-to-End Automatic Speech Recognition. *Symmetry*. Multidisciplinary Digital Publishing Institute. 2019, zv. 11, č. 8, s. 1018.

[49] Wang, Y., Peddinti, V., Xu, H., Zhang, X., Povey, D. et al. Backstitch: Counteracting Finite-Sample Bias via Negative Steps. In: *Interspeech*. 2017, s. 1631–1635.

[50] Watanabe, S. *Installation* [online]. [cit. 2020-04-06]. Dostupné z: https://espnet.github.io/espnet/installation.htm.

[51] Watanabe, S., Hori, T., Karita, S., Hayashi, T., Nishitoba, J. et al. ESPnet: End-to-End Speech Processing Toolkit. *CoRR*. 2018, abs/1804.00015. Dostupné z: http://arxiv.org/abs/1804.00015.

[52] Watanabe, S., Hori, T., Kim, S., Hershey, J. R. a Hayashi, T. Hybrid CTC/attention architecture for end-to-end speech recognition. *IEEE Journal of Selected Topics in Signal Processing*. IEEE. 2017, zv. 11, č. 8, s. 1240–1253.

[53] Wikipedia. *Cosine similarity* [online]. [cit. 2020-04-09]. Dostupné z: https://en.wikipedia.org/wiki/Cosine_similarity.

[54] Zeiler, M. D. Adadelta: an adaptive learning rate method. *ArXiv preprint arXiv:1212.5701*. 2012.

[55] Černocký, J. *Zpracování řečových signálů — studijní opora* [online]. 2006 [cit. 2020-05-26]. Dostupné z: https://www.fit.vutbr.cz/study/courses/ZRE/public/opora/zre_opora.pdf.

Príloha A

Obsah priloženého DVD

Priložené DVD obsahuje:

- návod na inštaláciu potrebných utilít a programov
- skripty Pytorch pre trénovanie sietí typu TDNN a TDNN-F
- dekódovacie skripty pomocou nástroja Kaldi
- skripty na trénovanie v Metacentre

Obsah zhodný s obsahom na repozitáry:
https://git.fit.vutbr.cz/xgajda03/pytorch_tdnn

A.1 Repozitár rozšíreného nástroja ESPnet

Nástroj ESPnet rozšírený o utility s nízko-dimenzionálnou faktorizáciou:
https://git.fit.vutbr.cz/xgajda03/espnet.ortho