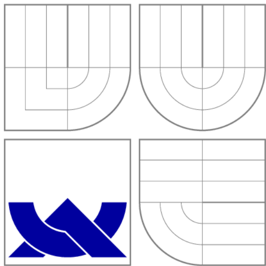
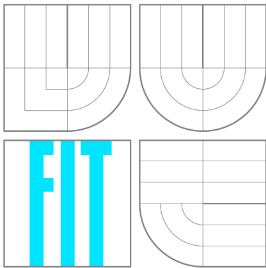


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

REKURENTNÍ NEURONOVÉ SÍTĚ PRO ROZPOZNÁVÁNÍ ŘEČI

RECURRENT NEURAL NETWORKS FOR SPEECH RECOGNITION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. TOMÁŠ NOVÁČIK
AUTHOR

VEDOUCÍ PRÁCE — Ing. KAREL VESELÝ
SUPERVISOR

BRNO 2016

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2015/2016

Zadání diplomové práce

Řešitel: **Nováčik Tomáš, Bc.**
Obor: Inteligentní systémy
Téma: **Rekurentní neuronové sítě pro rozpoznávání řeči**
Recurrent Neural Networks for Speech Recognition
Kategorie: Zpracování řeči a přirozeného jazyka

Pokyny:

1. Seznamte se s problematikou neuronových sítí (NN) a s jejich použitím v oblasti automatického přepisu řeči.
2. Seznamte se s teorií rekurentních sítí LSTM a trénovacího kriteria Connectionist Temporal Classification (CTC).
3. Vyberte vhodný toolkit a řečovou databázi pro trénování rozpoznávače řeči.
4. Natrénujte rozpoznávač s LSTM akustickým modelem, porovnejte s dopřednou neuronovou sítí.
5. Experimentujte s různými konfiguracemi systému.

Literatura:

- A. Graves, S. Fernández, et al.: Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks, ICML, 2006
- C. M. Bishop: Pattern recognition and machine learning, Springer, 2007
- D. Yu, L. Deng: Automatic Speech Recognition, a deep learning approach, Springer, 2015

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Veselý Karel, Ing.**, UPGM FIT VUT
Konzultant: Karafiát Martin, Ing., Ph.D., UPGM FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 25. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

// Abstrakt
Tato diplomová práce se zabývá implementací rekurentních neuronových sítí v prostředí jazyka lua za pomocí knihovny torch. Řeší problematiku trénování rekurentních neuronových sítí a to jak z hlediska optimální trénovací strategie, tak z hlediska urychlení trénovacího procesu. Zkoumá zakomponování technik batch normalizace a dropout do architektur rekurentních neuronových sítí. Jednotlivé typy rekurentních sítí jsou následně porovnány na úkolu rozpoznávání řeči prostřednictvým datové sady AMI, kde slouží pro modelování akustického modelu, a dochází ke srovnání s klasickou dopřednou neuronovou sítí. Nejlepší výsledek je dosažen prostřednictvým rekurentní neuronové sítě BLSTM. Následně dojde k natrénování rekurentních neuronových sítí prostřednictvím objektivní funkce CTC na databázi TIMIT, kde nejlepšího výsledku opět dosáhne BLSTM.

Abstract
This master thesis deals with the implementation of various types of recurrent neural networks via programming language lua using torch library. It focuses on finding optimal strategy for training recurrent neural networks and also tries to minimize the duration of the training. Furthermore various types of regularization techniques are investigated and implemented into the recurrent neural network architecture. Implemented recurrent neural networks are compared on the speech recognition task using AMI dataset, where they model the acustic information. Their performance is also compared to standard feedforward neural network. Best results are achieved using BLSTM architecture. The recurrent neural network are also trained via CTC objective function on the TIMIT dataset. Best result is again achieved using BLSTM architecture.

Klíčová slova
ctc, irnn, rnn, lstm, gru, kaldi, eesen, rekurentní neuronové sítě, rozpoznávání řeči, torch, paralelní zpracování sekvencí, akustický model, batch normalizace, ami, timit

Keywords
ctc, irnn, rnn, lstm, gru, kaldi, eesen, recurrent neural networks, speech recognition, torch, parallel sequence processing, acustic model, batch normalization, ami, timit

Citace
NOVÁČIK, Tomáš. *Rekurentní neuronové sítě pro rozpoznávání řeči*. Brno, 2016. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Veselý Karel.

Rekurentní neuronové sítě pro rozpoznávání řeči

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Karla Veselého. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Tomáš Nováčik
31. května 2016

Poděkování

Chtěl bych poděkovat mému vedoucímu Ing. Karlu Veselému za věnovaný čas, ochotu, udělené rady a připomínky. Také bych rád poděkoval Ing. Martinu Karafiátovi, Ph.D. za udělené rady. Můj velký dík také patří mým rodičům, kteří mě vždy podporovali během studijí.

© Tomáš Nováčik, 2016.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**

2 Úvod do rozponávání řeči **5**
2.1 Zpracování signálu a extrakce příznaků . . . 7
2.2 Akustický model . . . 7
2.3 Jazykový model . . . 10
2.4 Hledání hypotéz . . . 10

3 Neuronové sítě **14**
3.1 Aktivační funkce . . . 15
3.2 Trénování . . . 16
3.2.1 Algoritmus zpětného šíření chyby . . . 16
3.3 Inicializace . . . 21

4 Rekurentní neuronové sítě **22**
4.1 Trénování . . . 23
4.2 LSTM . . . 24
4.3 GRU . . . 27
4.4 IRNN . . . 28
4.5 Obousměrné rekurentní sítě . . . 28
4.6 Connectionist temporal classification . . . 29

5 Implementace **33**
5.1 Torch . . . 33
5.2 Implementace rekurence . . . 35
5.3 Moduly . . . 36
5.3.1 RNN . . . 40
5.3.2 IRNN . . . 41
5.3.3 Optimalizace . . . 41
5.4 Ovládání . . . 45
5.5 Testování rychlosti a paměťové náročnosti . . . 48

6 Datové sady **51**
6.1 AMI . . . 51
6.2 TIMIT . . . 51

7 Experimenty **52**
7.1 HMM 52
7.1.1 CD-DNN-HMM 53
7.1.2 CD-RNN-HMM 56
7.2 CTC 61
7.3 Vyhodnocení 61

8 Závěr **63**

Literatura **64**

Přílohy **67**
Seznam příloh 68

A Obsah CD **69**

Kapitola 1

Úvod

Rozpoznávání řeči je jedním z mnoha odvětví, ve kterém se v dnešní době široce uplatňují poznatky a techniky z oblasti strojového učení. Rozpoznávání řeči prošlo dlouhým historickým vývojem, od jehož začátku ušlo dalekou cestu. V počátcích rozpoznávání řeči bylo možné rozeznat jen malé množství slov. Jednalo se hlavně o vědeckou oblast, jež měla pramálo praktického uplatnění, celý výzkum se soustředil hlavně na jeden klíčový jazyk - angličtinu. V dnešní době se s rozeznáváním řečí setkáváme takřka denodenně v podobě různých agentů, jež s námi komunikují skrze mobilní telefony. Analýza řeči čím dál víc proniká i do dalších oblastí lidského poznání, jako je například medicína nebo biometrie. Nutno ovšem dodat, že tyto pokroky mají i svá úskalí, tkvící především v omezování lidské svobody a neustálého dohledu tajných nebo represivních složek různých států nad občany, které s vývojem těchto technologií ukrajují z lidského soukromí čím dál více.

Obrovský pokrok v rozpoznávání řeči byl způsoben hlavně výrazným pokrokem v oblasti strojového učení. Renesanci strojovému učení přinesl hlavně vývoj rychlejšího hardware a provádění výpočtů na grafické kartě, které umožnily zpracovávat daleko větší množství dat a vytvářet komplexnější modely. Jednou z klíčových technologií dnešní doby pro rozpoznávání řeči představují neuronové sítě, jež byly použity i v minulosti, ale neměly výraznější úspěch právě kvůli nízkému výpočetnímu výkonu počítačů tehdejší doby. [32]

Přes veškerý pokrok v oblasti strojového učení stále nejsme schopni dostatečně rychle a přesně indexovat většinu řečových dat, jež my lidé produkujeme. V oblasti rozpoznávání řeči tuto situaci komplikuje velké množství existujících jazyků, dialektů, šumu, různých fyzických vlastností lidí a velké množství různých situací, ve kterých se projevují emoce člověka. Je nutné si uvědomit, že i když budeme uvažovat pouze jednoho jedince a necháme ho vyslovit jedno slovo dvakrát, tak nikdy nebudou tato slova znít úplně totožně a nebudou stejně dlouhá.

Motivace

Schopnost převést řeč na psaný text nám umožňuje v datech vyhledávat užitečné informace, které zefektivní naši schopnost učení se novým dovednostem, protože v knihách a textových dokumentech se často nesetkáváme s tím samým podáním informace, jaké nám může poskytnout například spolužák při vysvětlování látky. Psaný text lze přeložit do jiného jazyka a zefektivnit tak spolupráci v mezinárodním prostředí.

Očekávaným přínosem této práce je získání experimentálních poznatků souvisejících s problematikou zpracování řeči a strojového učení a jejich následného využití k dosažení

lepších výsledků při rozpoznávání řeči. Zejména pak aplikací rekurentních neuronových sítí a chybové funkce Connectionist Temporal Classification (CTC). Rekurentní sítě se pro tuto činnost obzvláště hodí, protože jsou schopny efektivně kódovat předešlou historii vstupů a nespoléhat se při klasifikaci pouze na aktuální vstup, jako tomu je u dopředné neuronové sítě. CTC narozdíl od klasické klasifikace nepoužívá pevné zarovnání jednotlivých tříd v čase a zaměřuje se pouze na pořadí výskytu těchto tříd.

Organizace

Kapitola 2 obsahuje úvod do problematiky rozpoznávání řeči, kapitola 3 a 4 pak slouží jako úvod do problematiky dopředných a rekurentních neuronových sítí. V kapitole 5 je pak popsána implementace modulu pro rekurentní neuronové sítě a v kapitole 6 je obeznámení s datovými sadami použitými při experimentech. Kapitola 7 obsahuje provedené experimenty a kapitola 8 představuje závěr celé práce.

Kapitola 2

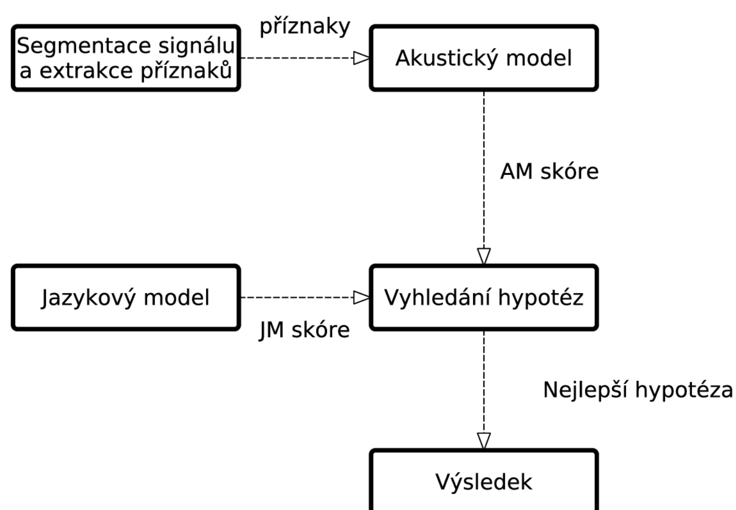
Úvod do rozponávání řeči

Tato kapitola čerpá z knihy [32] a částečně z dalších pramenů.

Rozpoznávání řeči je proces, při kterém se převádí řeč v podobě signálu do psaného textu. Systémy pro převod řeči do psaného textu sestávají ze čtyř následujících modulů:

1. zpracování signálu a extrakce rysů (sekce 2.1)
2. akustický model (sekce 2.2)
3. jazykový model (sekce 2.3)
4. hledání hypotéz (sekce 2.4)

V úvodní části zpracování řeči musí dojít k detekci řečových segmentů a převodu digitálních dat do frekvenčního spektra, které reflektuje řeč. Následně se pracuje s frekvenčním spektrem, které se většinou dále upravuje, aby ve výsledku příznaky co nejlépe odrážely obsaženou řečovou informaci. Příznaky se následně propagují do akustického modelu, kde jim je přiřazeno skóre, reflektující věrohodnost výskytu nějakého typu řečové jednotky, nejčastěji fonému. Jazykový model následně pracuje s tímto akustickým skóre a snaží se vytvořit nejpravděpodobnější posloupnost slov, kdy dochází k zohlednění vlastností jazyka nebo specifické oblasti, které se rozpoznávání týká. Výstupem jsou ohodnocené sekvence slov. Z této posloupnosti je potřeba vybrat co nejpravděpodobnější interpretaci vstupu, která představuje výsledek. Akustický a jazykový model se získávají z trénovacích dat, což jsou data, která máme k dispozici pro danou úlohu a je u nich znám přepis. Na obrázku 2.1 je zobrazeno propojení jednotlivých modulů pro rozpoznávání řeči. V této práci se budeme výrazně zaměřovat na akustický model, protože se v něm dají využít rekurentní neuronové sítě.



Obrázek 2.1: Interakce mezi jednotlivými moduly při automatickém přepisu řeči.

2.1 Zpracování signálu a extrakce příznaků

Vstupní signál se většinou rozdělí na rámce, které se jistým způsobem překrývají, a následně se provede *rychlá fourierova transformace*, jejíž výsledek se následně zpracovává do finální podoby příznaků. Mezi nejpopulárnější typy příznaků patří:

- MFCC
- PLP
- FBank

Na takto získané příznaky se následně aplikují techniky pro redukci dimenzionality (LDA), přídávají se k nim delta a delta-delta koeficienty, normalizují se, adaptují se na řečníka (fMLLR) nebo dochází k jejich konkatenaci.

2.2 Akustický model

Akustický model zachycuje pravděpodobnost výskytu $P(x|s)$ příznaku na stavu s. Trénovací data většinou obsahují textový přepis řeči, kdy je nutno jednotlivá slova přemapovat na fonémy nebo kontextově závislé fonémy.

Akustický model se z historického hlediska nejčastěji reprezentoval kombinací směsi gaussovských modelů (neboli GMM z anglického Gaussian mixture models) a Skrytých Markovových modelů (neboli HMM z anglického Hidden markov models), kdy se GMM a HMM používaly pro modelování časové dynamiky akustických jednotek. A celý takto vzniklý model se označoval GMM-HMM. Jednalo se o generativní model, kdy jednotlivé komponenty GMM modelovaly jednotlivé fonémy. V dnešní době se místo GMM používají buď dopředné neuronové sítě (kapitola 3) nebo rekurentní neuronové sítě (kapitola 4), protože jsou schopny díky diskriminativnímu trénování lépe klasifikovat jednotlivé stavy HMM. Mohlo by se zdát, že v tuto chvíli již nepotřebujeme GMM, ale není tomu tak, protože stále potřebujeme vygenerovat nějaké vstupní zarovnání na jednotlivé fonémy (nebo kontextově závislé fonémy) pro trénování neuronových sítí.

Směsi gaussovských rozložení

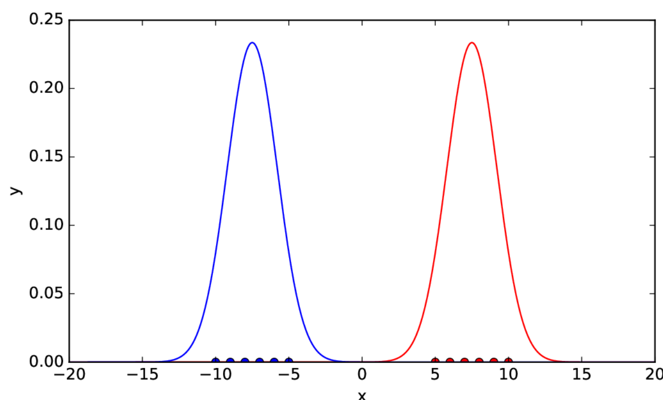
Směsi gaussovských rozložení se často využívají pro klasifikaci, jednorozměrná Gaussova funkce je definovaná jako:

$$g(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}, \tag{2.1}$$

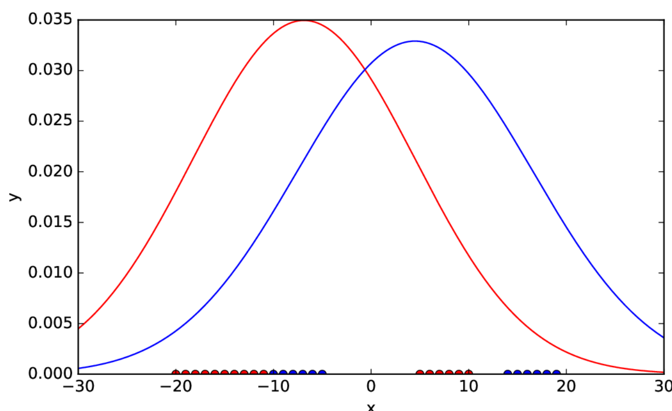
kde μ je střední hodnota a σ je směrodatná odchylka.

Uvažujme dvě jednoduché třídy, které obsahují pouze jednorozměrné příznaky, jež se budeme snažit modelovat dvěma gaussovými funkcemi. Pokud distribuce dat v těchto třídách bude stejná, jako je Gaussovo rozložení pravděpodobnosti, pak pro modelování těchto tříd bude ideální gaussovské rozložení pravděpodobnosti, jehož parametry lze odhadnout na základě střední hodnoty a standardní odchylky v datech těchto tříd. Tato situace je vybrazena na obrázku 2.2.

Jiná situace nastane, pokud se data těchto tříd prolínají (obrázek 2.3). V tomto případě už nedojde ke správnému rozlišení jednotlivých bodů.



Obrázek 2.2: Jednoduchá ukázka klasifikace dvou tříd pomocí gaussova rozložení pravděpodobnosti.



Obrázek 2.3: Ukázka chybné klasifikace při prolínání dat.

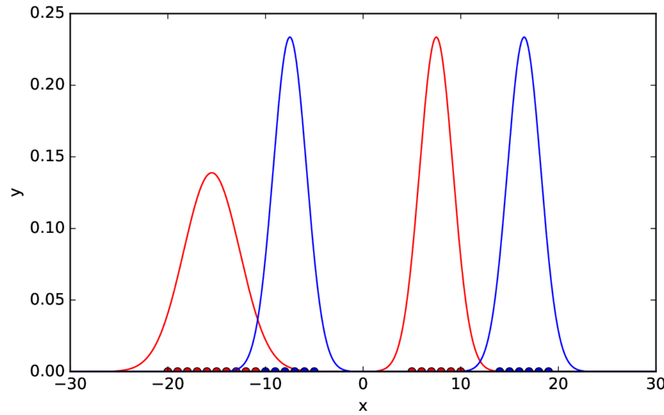
Řešením této situace je přidání vícero gassovských rozložení pro každou třídu, kdy je každé gaussovské funkci přiřazena váha. Tyto modely se nazývaji směsi gaussovských rozložení a obecně jsou definovány vzorcem 2.2.

$$g(x) = \sum_{m=1}^{M} \frac{c_m e^{-\frac{1}{2}(x-\mu_m)^T \Sigma_m^{-1}(x-\mu_m)}}{\sqrt[2]{(2\pi)^D |\Sigma_m|}}, \tag{2.2}$$

kde Σ je kovariační matice, D odpovídá rozměru vektoru x, c_m jsou váhy jednotlivých gaussovských rozložení, M je celkový počet komponent ve směsi gaussovských rozložení pro každou třídu a platí $\sum_{m=1}^{M} c_m = 1$. Na obrázku 2.4 je zobrazeno ilustrační řešení předchozího problému.

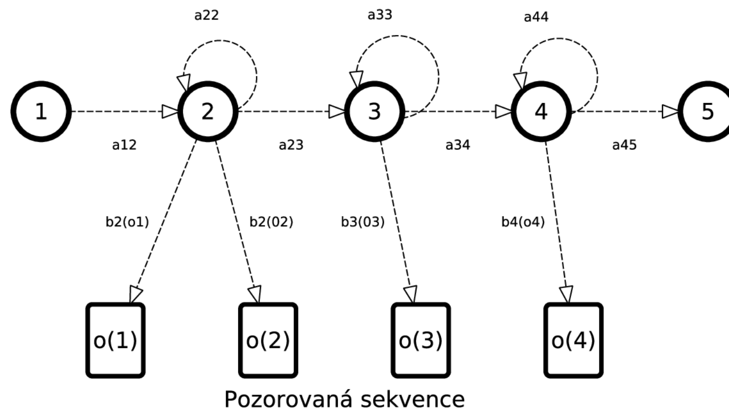
Skryté Markovovy modely

Markovův model je konečný automat, jehož hrany jsou ohodnoceny pravděpodobnostmi $a_{i,j}$, které udávají pravděpodobnost přechodu do následujícího stavu v každém jednom časovém okamžiku t, přičemž v každém okamžiku t je při vstupu do stavu s vygenerovaná



Obrázek 2.4: Ukázka správné klasifikace při prolínání dat, při použití vícero gaussovek.

věrohodnost odpovídající funkci hustoty pravděpodobnosti b_s asociované se stavem a aktuálním vstupem. Funkce hustoty pravděpodobnosti může být modelovaná pomocí GMM, DNN (kapitola 3) nebo i RNN (kapitola 4). Typická grafická reprezentace je vyobrazena na obrázku 2.5. První a poslední stav představují vstupní a výstupní místa modelu pro příznaky o, které neemitují žádné hodnoty.



Obrázek 2.5: Markovův model.

Uvažujme, že je pozorováním O vygenerována sekvence stavů $S = \{1, 2, 2, 3, 4, 5\}$ modelu M, pak podmíněnou věrohodnost $P(O, S|M)$ můžeme vypočíst:

$$P(O, S|M) = a_{1,2}b_2(o_1)a_{2,2}b_2(o_2)a_{2,3}b_3(o_3)a_{3,4}b_4(o_4)a_{4,5} \tag{2.3}$$

V praxi je ovšem sekvence stavu S neznámá - odtud skryté Markovovy modely. V tomto případě je nutné provést ohodnocení přes všechny možné kombinace stavů $S = \{s_1, \dots, s_T\}$:

$$P(O|M) = \sum_S a_{s_1,s_2} \prod_{t=1}^{T} b_{s_t}(o_t)a_{s_t,s_{t+1}}, \tag{2.4}$$

všechny sekvence musí končit ve stavu s_{T+1} a začínat ve stavu s_0. Tento proces je výpočetně náročný. Proto se využívá dynamického programování a algoritmu backward/forward.

[31]

2.3 Jazykový model

Jazykový model produkuje pravděpodobnosti výskytu sekvencí slov, tyto pravděpodobnosti jsou vypočítány většinou pomocí *n-gramového modelu*, kdy je pravděpodobnost výskytu slova ovlivněna $n-1$ slovy předcházejícími jeho výskytu. Nejjednodušší modely pro $n=1$ se nazývají unigramové modely, pro $n=2$ bigramové a tak dále. Obecně platí, že čím větší n tím lepší výsledky může jazykový model poskytovat, za předpokladu, že je trénován na dostatečně velkém vzorku dat.

Jazykové modely bývají v praktických rozpoznávačích trénovány na velkých textových korpusech, které jsou obrovské v porovnání s množstvím textové informace prezentované ve zvukových záznamech. Pokud se rozpoznávač zaměřuje na nějakou typickou oblast, pak by měl jazykový model obsahovat hlavně slova z dané oblasti.

Klíčovým úkolem jazykového modelu je upřednostnění těch sekvencí slov, jež se často vyskytují ve své blízkosti. Toto pomáhá hlavně ve chvílích, kdy jsou dvě slova jenom velice těžko rozlišitelná na základě akustické informace.

Alternativou k *n-gramovému modelu* jsou rekurentní neuronové sítě [23].

2.4 Hledání hypotéz

Následující sekce čerpá z [24]. Pro hledání hypotéz se využívají vážené konečné stavové automaty (WFSA z anglického weighted finite state acceptor) nebo konečné stavové převodníky (WFST z anglického Weighted finite state transducer), kterými je reprezentován akustický a jazykový model. Nalezení hypotézy je formulováno jako:

$$\hat{W} = \frac{\text{argmax}(P(W)P(X|W))}{P(X)}, \tag{2.5}$$

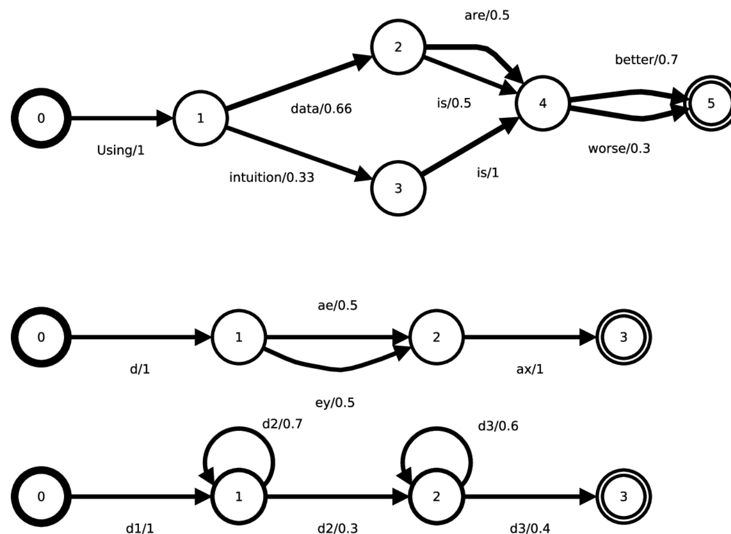
kde $\hat{W}$ představuje takovou sekvenci slov W, která maximalizuje věrohodnost jazykového modelu $P(W)$ a věrohodnost $P(X|W)$ pozorování rámce X vztaženou na pozorování slova W.

WFSA

Vážený konečný automat je tvořen sadou stavů, jež jsou propojeny hranami. Každá hrana obsahuje znak z přijímané abecedy a je ohodnocena jistou váhou. Kromě klasických stavů existuje počáteční stav, ze kterého startuje proces přijímání řetězce, a konečná množina koncových stavů, která symbolizuje ukončení přijímání řetězce. Každému přijatému řetězci přísluší finální ohodnocení, které je tvořeno váhami, které se nacházely na hranách při přijímaní daného řetězce. V tomto případě jsou jednotlivé WFSA zpracovávány samostatně a výsledek je vždy předán následujícímu WFSA. Na obrázku 2.6 jsou zobrazeny WFSA pro jazykový, fonetický a HMM model.

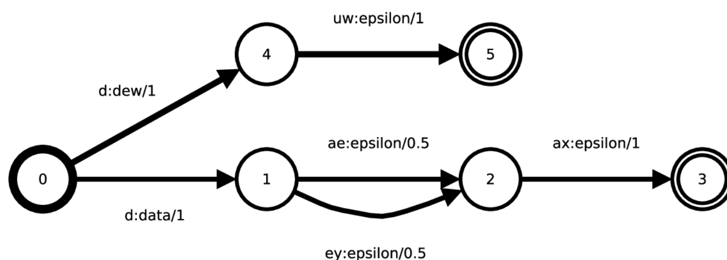
WFST

Vážený konečný převodník je podobný WFSA, jeho hlavní výhodou je zachovávání kontextové informace a možnost kombinovat více modelů do jednoho společného modelu pomocí operace kompozice. Kontextová informace se v tomto případě zachovává na hranách mezi



Obrázek 2.6: WFSA pro jazykový model, fonetický model a HMM.

jednotlivými stavy, kdy každá hrana má jednak svoji váhu, přijímaný znak z abecedy, ale také výstupní znak. Ukázka použití je na obrázku 2.7.



Obrázek 2.7: WFST představující lexikon.

Při průchodu do konečného stavu se opět akumulují váhy na hranách mezi stavy pomocí zvoleného operátoru a hned v prvním stavu se emituje symbol odpovídající stavu z jazykového modelu. Všechny následující stavy emitují *epsilon*, tedy prázdný znak. V konečném stavu se tedy ocitáme se symbolem slova a jeho skóre.

V klasické úloze rozpoznávání řeči se setkáváme s následujícími typy WFST:

- H - reprezentující HMM, jeho vstupem je identifikace funkce hustoty pravděpodobnosti a výstupem je kontextově závislý foném.
- C - reprezentující kontextovou závislost jednotlivých fonémů, na vstupu je kontextově závislý foném a na výstupu je foném
- L - reprezentující výslovnostní model pro jednotlivá slova, na vstupu jsou fonémy a na výstupu potom jednotlivá slova
- G - reprezentující n-gramový jazykový model, v tomto případě se jedná o klasický automat, protože jeho vstupní i výstupní symboly jsou identické, avšak pro zjednodušení se doplňuje o výstupní symboly, aby zapadl do kontextu převodníku

Výsledný převodník je charakterizován kombinací těchto převodníků:

$$HCLG = H \circ C \circ L \circ G. \tag{2.6}$$

Postupy, které vedou k dosažení tohoto výsledného převodníku se v praxi liší v závislosti na pořadí a druhu jednotlivých optimalizačních algoritmů. Pro toolkit kaldi [26] je typický následující postup:

$$HCLG = asl(min(rds(det(H_a \circ min(det(C \circ min(det(L \circ G)))))), \tag{2.7}$$

kde význam jednotlivých operací je následující:

- det - determinizace
- min - minimalizace
- rds (z anglického remove disambiguation symbols) - odstranění pomocných symbolů, kdy pomocné symboly slouží k rozlišení některých sekvencí symbolů (typicky u slov se stejnou výslovností nebo pří využívání back-off modelu)
- asl (z anglického add self loop) - přidání jednoduchých cyklů

Přidáme-li ještě další komponentu U, která určuje akustické skóre jednotlivých rámců, dostáváme výsledný graf S, jehož prohledáním se snažíme dosáhnout co největšího skóre.

$$S = U \circ (HCLG) \tag{2.8}$$

Tento prohledávací graf bývá často obrovský, někdy dochází ke konstrukci pouze částečného grafu a zbytek se sestavuje dynamicky, pokud je to potřeba. Často se taky přistupuje k jeho prořezávání a během vyhledávání se aplikuje tzv. *beam search*. Výstupem je sada hypotéz, která se označuje jako *lattice*. Jedná se o kompaktní strukturu, ve které jsou sdílené společné části jednotlivých nalezených cest spolu s dalšími informacemi, jako je třeba zarovnání jednotlivých slov.

Hodnocení kvality

Pro určení úspěšnosti převodu mluvené řeči na text se využívá různých typů metrik. Jeden z hlavních rozdílů napříč těmito metrikami tkví ve velikosti základní jednotky, nad kterou se evaluuje úspěšnost, typicky věta, foném nebo slovo. V této práci budeme pracovat s metrikou *Word Error Rate (WER)*, jejíž základní jednotkou je slovo, nebo *Phoneme Error Rate (PER)*, u níž je základní jednotkou foném. Metrike *WER* je definovaná následovně:

$$WER = 100\frac{I + D + S}{WORD_COUNT}, \tag{2.9}$$

kde význam jednotlivých proměnných je následující:

- I - počet nesprávně vložených slov
- D - počet chybějících slov
- S - počet slov, která byla zaměněna za jiná slova
- $WORD_COUNT$ - počet slov v referenčním přepisu

Tato metrika může přesáhnout hodnoty 100%. Pokud zaměníme slova za fonémy, dostáváme PER. [17]

Kaldi a Eesen

Kaldi je open source software určený nejen pro rozpoznávání řeči. Jeho implementační jazyk je *C++*. Obsahuje velké množství předpřipravených receptů nad známými datasety, které většinou dosahují state of the art výsledků. Jeho součástí je řada bashových a perlových skriptů, které slouží pro připravení dat od extrakce příznaku, trénování akustických modelů až po dekódování. Umožňuje distribuované zpracování velkého množství dat a také paralelní trénování akustických modelů jak na CPU tak i GPU. V současné době je podporována celá řada neuronových sítí, ale také GMM modely. Pro dékodování využívá WFST (sekce 2.4).[26]

Eesen je odvozený od toolkitu kaldi, který rozšiřuje kaldi o podporu pro CTC (sekce 4.6) a integruje tuto techniku do dekódovacího procesu. [22]

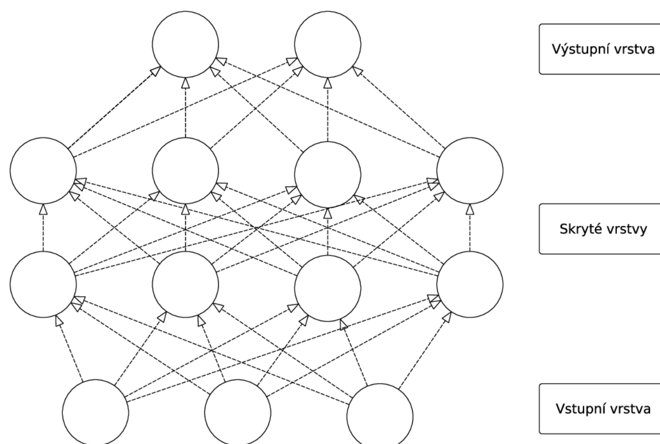
Kapitola 3

Neuronové sítě

Neuronové sítě jsou matematický model vzdáleně inspirovaný hustě propojenou sítí neuronů v lidském mozku. Nejpopulárnějším modelem je dopředná neuronová síť, jejíž topologie je zobrazena na obrázku 3.1. Tato neuronová síť má celkově 4 vrstvy, 1 výstupní, 2 skryté a 1 vstupní vrstvu. V dopředné neuronové síti je vstupní signál propagován každou vrstvou do následující vrstvy, tedy každý neuron propaguje svůj výstup všem ostatním neuronům v další vrstvě, neuron představuje základní složku každé vrstvy. Uvažujme, že neuron má I vstupů, pak je jeho funkce popsána v následujícím vzorci:

$$neuron = f(\sum_{i=1}^{I} w_i x_i + b), \tag{3.1}$$

kde f je aktivační funkce, b je bias, w jsou váhy přiřazené vstupu a x představuje vstup samotný.



Obrázek 3.1: Topologie dopředné neuronové sítě.

3.1 Aktivační funkce

Existují různé typy aktivačních funkcí, mezi nejpopulárnější patří logistická sigmoida:

$$\sigma(z) = \frac{1}{1+e^{-z}}, \tag{3.2}$$

s derivací:

$$\sigma'(z) = (1-\sigma(z))\sigma(z), \tag{3.3}$$

hyperbolický tangens:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}, \tag{3.4}$$

s derivací:

$$\tanh'(z) = 1 - \tanh(z)^2, \tag{3.5}$$

rectified linear unit(ReLU):

$$ReLU(z) = max(0, z), \tag{3.6}$$

s derivací:

$$ReLU'(z) = max(0, sgn(z)), \tag{3.7}$$

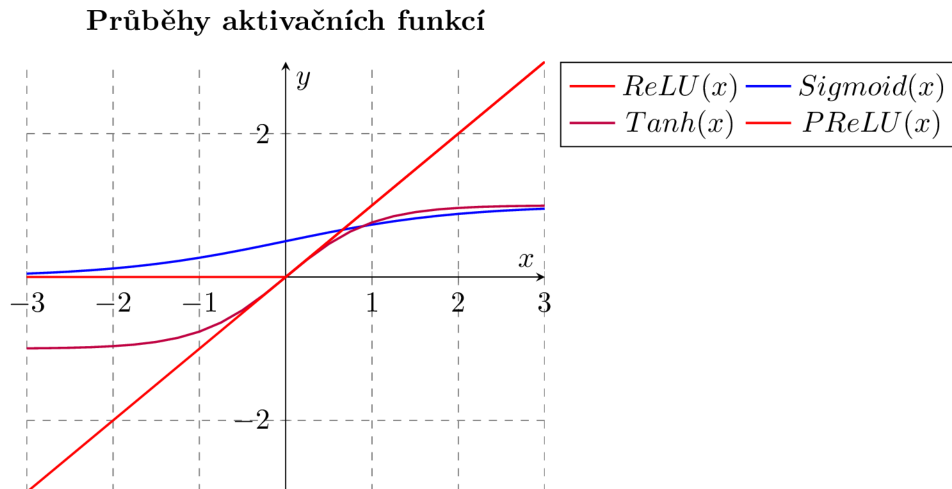
a parametrizovaná ReLU, která byla představena v [13] a zaznamenala úspěch na datové sadě imageNet při použití s konvolučními neuronovými sítěmi. Jako jediná z aktivačních funkcí obsahuje parametr, který je adaptován během trénování, za zavedením tohoto parametru je snaha o odstranění nulového gradientu:

$$PReLU(z) = \begin{cases} z & \text{if } z > 0 \\ az & \text{if } z <= 0 \end{cases} \tag{3.8}$$

$$PReLU'(z) = \begin{cases} 1 & \text{if } z > 0 \\ a & \text{if } z <= 0 \end{cases} \tag{3.9}$$

Na následujícím grafu jsou vyobrazeny průběhy jednotlivých aktivačních funkcí. U aktivační funkce $PReLU$ byl zvolen koeficient $a = 1$.

Průběhy aktivačních funkcí



Neuronové sítě mohou být použity jak pro klasifikaci tak i pro regresi, v této práci se ovšem budeme zabývat klasifikací. V případě klasifikace každý výstupní neuron představuje jednu klasifikační třídu. Mějme ve výstupní vrstvě I neuronů, pokud chceme, aby výstup sítě šel interpretovat jako posteriorní pravděpodobnosti, jejichž suma musí být 1, použijeme v poslední vrstvě nelinearitu softmax:

$$softmax_i(x) = \frac{e^{x_i}}{\sum\limits_{c=1}^{C} e^{x_i}} \tag{3.10}$$

Práce s neuronovou sítí je obecně charakterizována dvěma etapami:

- trénování modelu na známých datech, kde je k dispozici mapování mezi třídami a daty
- používání modelu na neznámých datech a výpočet posteriorní pravděpodobnosti, že daná data náleží dané třídě

3.2 Trénování

Trénování je proces, při kterém dochází k takové úpravě parametrů modelu, aby se co nejvíce minimalizovala chyba na výstupní vrstvě. Velikost chyby na výstupu neuronové sítě se charakterizuje pomocí objektivní funkce. Uvažujme následující proměnné W, b, y, o, které mají po řadě následující význam: všechny váhy modelu, všechny biasy modelu, aktuální výstup na výstupní vrstvě, aktuální vstup, pak můžeme definovat obecnou objektivní funci jako O a chybu ϵ:

$$\epsilon = O(W, b; o, y) \tag{3.11}$$

Tato chyba by měla co nejvíce korelovat se schopností neuronové sítě správně klasifikovat neznámá data. Nejpopulárnější funkcí pro klasifikaci představuje cross entropie (CE) (vzorec 3.12).

$$O_{CE}(W, b; o, y) = -\sum_{i=1}^{C} y_i \log v_i, \tag{3.12}$$

kde C představuje celkové množství tříd. y_i představuje pravděpodobnost, že dané pozorování o náleží s pravděpodobností $P_{emp}(i|o)$ třídě i získanou z trénovacích dat, a v_i je ta samá pravděpodobnost $P_n(i|o)$ odhadnutá neuronovou sítí. Ve většině případů se v trénovacích datech vyskytují jenom dvě hodnoty pravděpodobnosti P_{emp}, a to 1 pro třídu c, do které je dané pozorování o zahrnuto, a 0 pro všechny ostatní třídy.

Nejpoužívanějším algoritmem pro trénování neuronových sítí je algoritmus zpětného šíření chyby (kapitola 3.2.1).

3.2.1 Algoritmus zpětného šíření chyby

Nejznámějším algoritmem pro trénování neuronových sítí je algoritmus zpětného šíření chyby, též backpropagation, který je odvozen za použití řetězového pravidla pro výpočet derivací. Parametry neuronové sítě můžou být zlepšovány během trénování pomocí objektivní funkce a derivací prvního řádu následujícím způsobem:

$$W_{i+1}^l \leftarrow W_i^l - \varepsilon \Delta W_i^l, \tag{3.13}$$

$$b_{i+1}^l \leftarrow b_i^l - \varepsilon \Delta b_i^l, \tag{3.14}$$

kde W_i^l, b_i^l, ΔW_i^l, Δb_i^l, ε představují matici vah, matici biasů, korekci vah, korekci biasů ve vrstvě l a rychlost učení, při i-té iteraci algoritmu. Při zpracování M pozorování o se změna váhy vypočte následujícím způsobem:

$$\Delta W_i^l = \frac{1}{M} \sum_{m=1}^{M} \nabla_{W_i^l} O(W, b; o^m, y^m), \tag{3.15}$$

změna biasu pak:

$$\Delta b_i^l = \frac{1}{M} \sum_{m=1}^{M} \nabla_{b_i^l} O(W, b; o^m, y^m), \tag{3.16}$$

$\nabla_\Theta O$ je gradient chybové funkce O vzhledem k parametrům modelu Θ. Pro klasifikaci pomocí CE (vzorec 3.12) a výstupní vrstvy s nelinearitou softmax (vzorec 3.10) dostáváme gradient výstupní vrstvy vzhledem k matici vah výstupní vrstvy, kde z_i^L představuje hodnotu neuronu i v poslední vrstvě před aplikací nelinearity softmax a v_i^l představuje potenciál neuronu i respektive hodnotu neuronu i před aplikací nelinearity ve vrstvě l:

$$\begin{aligned} \nabla_{W_i^L} O_{CE}(W, b; o, y) &= \nabla_{z_i^L} O_{CE}(W, b; o, y) \frac{\partial z_i^L}{\partial W_i^L} \\ &= \epsilon_i^L \frac{\partial (W_i^L v_i^{L-1} + b_i^L)}{\partial W_i^L} \\ &= \epsilon_i^L (v_i^{L-1})^T \\ &= (v_i^L - y)(v_i^{L-1})^T, \end{aligned} \tag{3.17}$$

chyba pak bude definována následovně:

$$\begin{aligned} \epsilon_i^L &\triangleq \nabla_{z_i^L} O_{CE}(W, b; o, y) \\ &= -\frac{\partial \sum_{k=1}^{C} y_k \log softmax_k(z_i^L)}{\partial z_i^L} \\ &= \frac{\partial \sum_{k=1}^{C} y_k \log \sum_{j=1}^{C} e^{z_j^L}}{\partial z_i^L} - \frac{\partial \sum_{k=1}^{C} y_k \log e^{z_k^L}}{\partial z_i^L} \\ &= \frac{\partial \log \sum_{j=1}^{C} e^{z_j^L}}{\partial z_i^L} - \frac{\partial \sum_{k=1}^{C} y_k z_k^L}{\partial z_i^L} \\ &= (v_i^L - y), \end{aligned} \tag{3.18}$$

pro bias podobně:

$$\nabla_{b_i^L} O_{CE}(W, b; o, y) = (v_i^L - y), \tag{3.19}$$

uvažujme za skryté vrstvy všechny vrstvy s indexem l, kde platí $0 < l < L$, úprava vah pak bude mít následující charakter:

$$
\begin{aligned}
\nabla_{W_i^l} O_{CE}(W,b;o,y) &= \nabla_{v_i^l} O_{CE}(W,b;o,y)\frac{\partial v_i^l}{\partial W_i^l} \\
&= diag(f'(z_i^l))\epsilon_i^l \frac{\partial (W_i^l v_i^{l-1} + b_i^l)}{\partial W_i^l} \\
&= diag(f'(z_i^l))\epsilon_i^l (v_i^{l-1})^T \\
&= [f'(z_i^l) \cdot \epsilon_i^l](v_i^{l-1})^T,
\end{aligned} \tag{3.20}
$$

úprava biasů pak:

$$
\begin{aligned}
\nabla_{b_i^l} O_{CE}(W,b;o,y) &= \nabla_{v_i^l} O_{CE}(W,b;o,y)\frac{\partial v_i^l}{\partial b_i^l} \\
&= diag(f'(z_i^l))\epsilon_i^l \frac{\partial (W_i^l v_i^{l-1} + b_i^l)}{\partial b_i^l} \\
&= diag(f'(z_i^l))\epsilon_i^l (v_i^{l-1})^T \\
&= f'(z_i^l) \cdot \epsilon_i^l,
\end{aligned} \tag{3.21}
$$

ϵ_i^l představuje chybu ve vrstvě l, $\cdot$ je meziprvkové násobení, $diag(x)$ je čtvercová matice, jejíž hodnoty v diagonále odpovídají hodnotě x, $f'(z_i^l)$ představuje vektor derivací aktivačních funkcí ve vrstvě, derivace jednotlivých funkcí jsou zobrazeny v následujících vzorcích 3.3, 3.7 a 3.5.

Propagace chyby do nižších vrstev je charakterizována následovně:

$$
\begin{aligned}
\epsilon_i^{l-1} &= \nabla_{v_i^{l-1}} O_{CE}(W,b;o,y) \\
&= \frac{\partial v_i^l}{\partial v_i^{l-1}} \nabla_{v_i^l} O_{CE}(W,b;o,y) \\
&= \frac{\partial (W_i^l v_i^{l-1} + b_i^l)}{\partial v_i^{l-1}} diag(f'(z_i^l))\epsilon_i^l \\
&= (W_i^l)^T [f'(z_i^l) \cdot \epsilon_i^l]
\end{aligned}. \tag{3.22}
$$

Pro zlepšení průběhu trénování se často používá momentum (kapitola 3.2.1). Jeden z výrazných problémů při trénování neuronové sítě představuje tzv. přetrénování (kapitola 3.2.1), kdy dochází k přílišné adaptaci modelu neuronové sítě na trénovací data a ten pak špatně generalizuje na nová data, která nebyla součástí trénování.

Momentum

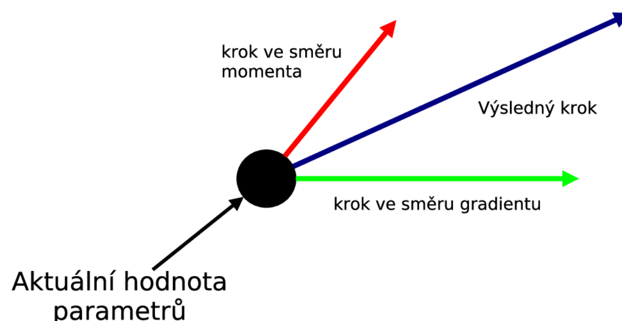
Momentum představuje techniku, díky níž je možné zabránit oscilacím vah při trénování technikou zpětného šíření chyby pomocí rozložení úprav vah δW_i^l a biasů δb_i^l do více iterací, konkrétně aplikováním momenta na rovnice 3.15 a 3.16 dostáváme následující vzorce:

$$
\Delta W_i^l = \rho \Delta W_{i-1}^l + (1-\rho)\frac{1}{M}\sum_{m=1}^{M} \nabla_{W_i^l} O(W,b;o^m,y^m), \tag{3.23}
$$

$$\Delta b_i^l = \rho \Delta b_{i-1}^l + (1-\rho)\frac{1}{M}\sum_{m=1}^{M} \nabla_{b_i^l} O(W, b; o^m, y^m), \tag{3.24}$$

kde se momentum značí ρ a má většinou hodnoty v rozsahu 0.9 - 0.99. Výsledný efekt představuje vyhlazení korekcí parametrů a tedy zmenšení oscilace během trénování.[32]

Efekt aplikace momenta je vyobrazen na obrázku 3.2.

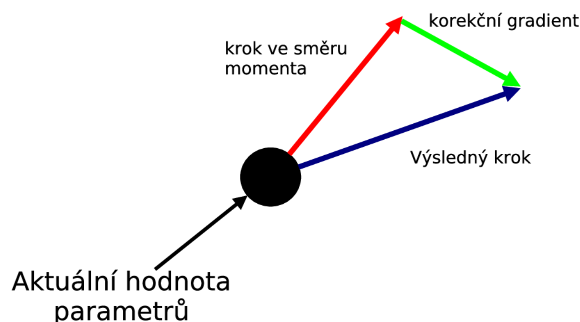


Obrázek 3.2: Efekt klasického momenta.

Nesterovo momentum

U Nesterova momenta se podobně jako u klasického momenta akumuluje hodnota převažujícího směru, avšak liší se v aplikaci. V tomto případě se momentum aplikuje před výpočtem aktuálního gradientu. Následně se vypočte gradient, jenž koriguje směr pohybu, vypočte se z něj nová korekční hodnota momenta a ta se následně aplikuje jako ΔW.[28]

Efekt aplikace Nesterova momenta je vyobrazen na obrázku 3.3.



Obrázek 3.3: Efekt Nesterova momenta.

Dropout

Dropout je regularizační technika, která se aplikuje pouze během trénování neuronové sítě. Podstatou této techniky je náhodné vynechání jisté podmnožiny neuronů v každé vrstvě. Tímto způsobem dochází k vložení šumu do reprezentace trénovacích dat ve skrytých vrstvách. Každý neuron se tak stává méně závislý na ostatních neuronech a měl by produkovat robustnější výstup. Míra vynechávání je ovládána parametrem, který udává, jak moc je pravděpodobné, že výstup neuronu v dané vrstvě bude vynechán. Během evaluace modelu se tato technika vypne.

Batch normalizace

Batch normalizace představuje jednu z nejnovějších technik, která by měla mít vliv hlavně na rychlost trénování, nicméně některé články uvádějí, že by měla u velkých sítí sloužit také jako regularizační prostředek [12].

Základní myšlenka této metody tkví v aplikování normalizace na výstup vrstev neuronové sítě. Není ovšem potřeba vypočíst hodnoty pro normalizaci nad celou datovou sadou, ale pouze aproximovat prostřednictvím jednotlivých datových dávek, jejichž částečné výpočty slouží k online výpočtu celkové střední hodnoty a rozptylu. Konkrétně pak [18] zavádí samostatnou vrstvu, pro pozici i jednoho parametru všech vstupních prvků v dávce by tato transformace měla následující tvar:

$$\mu = \frac{1}{M}\sum_{i=1}^{M} x_i \tag{3.25}$$

$$\sigma^2 = \frac{1}{M}\sum_{i=1}^{M}(x_i - \mu)^2 \tag{3.26}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu}{\sigma^2 + \epsilon} \tag{3.27}$$

$$y_i \leftarrow \gamma\hat{x}_i + \beta \tag{3.28}$$

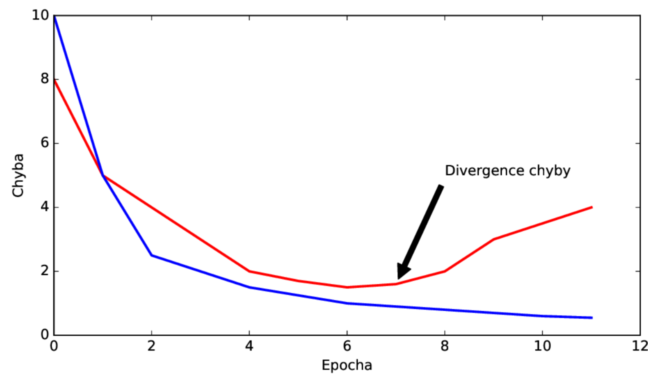
,

kde μ představuje střední hodnotu, σ^2 rozptyl, $\hat{x}_i$ představuje normalizovanou hodnotu a následují parametry γ a β, které jsou určeny pro posun hodnot a změnu jejich měřítka. V tomto případě jde o snahu zachovat vždy stejný dynamický rozsah hodnot, jedná se o klasické parametry, které se mění při trénování neuronové sítě. Tato vrstva je zjednodušena a slouží pouze pro odhad střední hodnoty a rozptylu na základě aktuální dávky.

V případě evaluačního módu jsou místo odhadů v rámci dávky použity statistiky získané z trénovacích dat.

Přetrénování

Přetrénování (z anglického overfiting) je stav, kdy dochází s každým dalším trénovacím cyklem ke zlepšení výsledků klasifikace na trénovací sadě, avšak na testovacích sadě se naopak chyba zvětšuje. Tento negativní jev se nejčastěji řeší pomocí techniky *early stopping*, kdy dochází k vyčlenění části trénovacích dat nazvané *held-out* sada, na které neprobíhá trénování, ale pouze se testuje úspěšnost klasifikátoru. Pokud dojde ke zhoršení klasifikace na *held-out* sadě, je trénování nejprve zpomaleno snížením rychlosti učení a později zastaveno. Tento fenomén je graficky zobrazen na obrázku 3.4.



Obrázek 3.4: Vývoj chyby během trénování na trénovací a *held-out* sadě.

3.3 Inicializace

Ukazuje se, že počáteční inicializace parametrů neuronové sítě je kritická pro úspěšné trénování, zvláště pak v případě hlubokých sítí. Existuje vícero způsobů jak provést dobrou inicializaci.

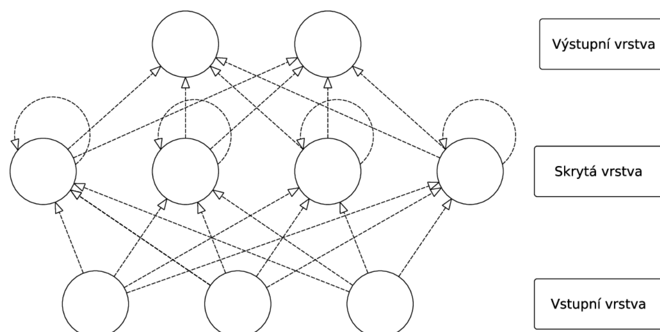
Nejefektivnější způsob je takzvané předtrénování, při kterém dochází před samotnou optimalizací neuronové sítě pomocí algoritmu zpětného šíření chyby k využití jiného optimalizačního algoritmu, po této prvotní fázi se pak pokračuje klasickým gradientním přístupem. Tento přístup byl poprvé aplikován v Deep Belief sítích, které odstartovaly zájem o hluboké neuronové sítě [15]. Nevýhodou tohoto přístupu je čas, který je nutný věnovat předtrénování.

Další možností je použít inicializaci generátorem náhodných čísel s vhodně zvolenou hustotou pravděpodobností pro váhy [20], [14], [8].

Kapitola 4

Rekurentní neuronové sítě

Rekurentní neuronové sítě (RNN) se významně liší od klasických dopředných sítí, jejich výstup není dán pouze aktuálním vstupem ale i vnitřním stavem sítě. Jejich vnitřní stav kóduje historii vstupů, jež už byla zpracována, což jim umožňuje se učit různé reprezentace a závislosti mezi vstupy napříč časem. Topologie jednoduché RNN s jednou skrytou vrstvou je na obrázku 4.1. Je patrné, že spojení ve skryté vrstvě vedou jak k výstupní vrstvě tak i ke skryté vrstvě samotné, vstupem těmto rekurentním spojením je výstup skryté vrstvy v předešlém časovém kroku.



Obrázek 4.1: Topologie rekurentní neuronové sítě.

Nechť je v každém časovém kroku t specifikován vstup jako x_t, stavy skryté vrstvy jako h_t a aktivace skryté vrstvy jako z_t, přičemž jsou všechny tyto proměnné reprezentovány vektory o velikosti $K \times 1$, $N \times 1$ a $L \times 1$, pak jednoduchá RNN z obrázku 4.1 může být reprezentována následujícími vzorci:

$$h_t = f(W_{xh}x_t + W_{hh}h_{t-1}), \tag{4.1}$$

$$z_t = g(W_{hy}h_t), \tag{4.2}$$

kde W_{hy} je matice vah o velikosti $L \times N$ spojující N neuronů ze skryté vrstvy s L výstupními neurony, W_{xh} je matice vah o velikosti $N \times K$ spojující K vstupních neuronů s N neurony ve

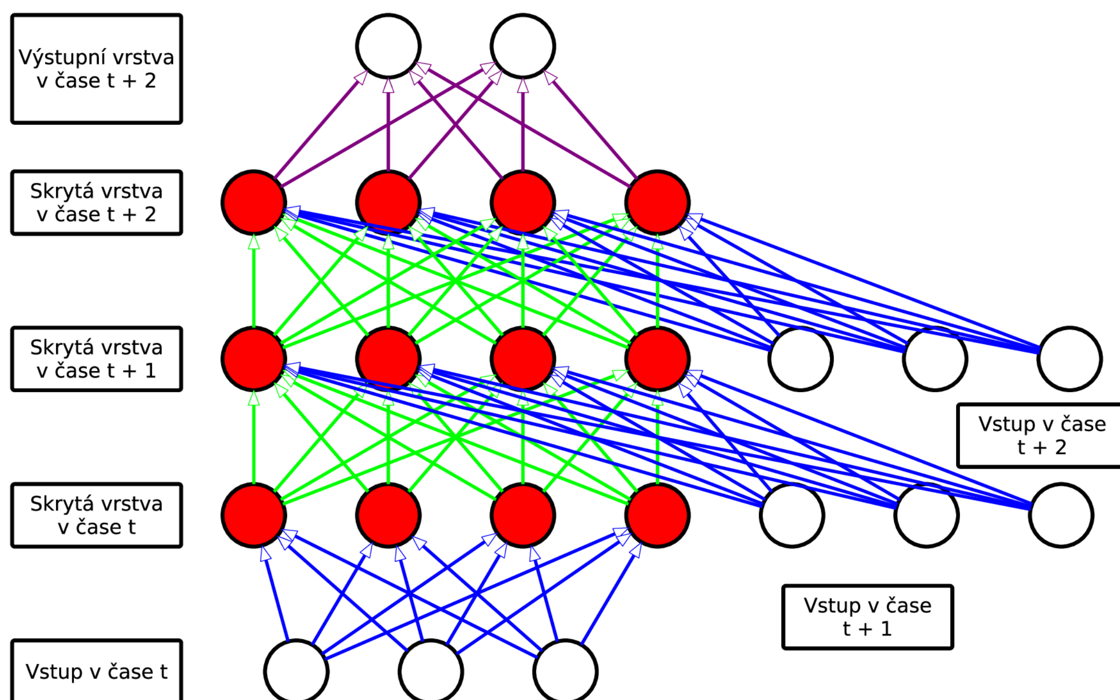
skryté vrstvě a W_{hh} je matice vah realizující propagaci signálu z předešlého časového kroku, $u_t = W_{xh}x_t + W_{hh}h_{t-1}$ je vektor o velikosti $N \times 1$ a obsahuje potenciály skrytých neuronů, $v_t = W_{hy}h_t$ je vektor o velikosti $L \times 1$ a obsahuje potenciály neuronů výstupní vrstvy, f_{u_t} je aktivační funkce skrytých neuronů, $g(v_t)$ je aktivační funkce neuronů výstupní vrstvy. Jako aktivační funkce se používají u RNN ty samé aktivační funkce jako u dopředných neuronových sítí, tedy: $tanh$ (vzorec 3.4), $sigmoida$ (vzorec 3.2), $ReLU$ (vzorec 3.6) a v případě klasifikace má výstupní vrstva nelinearitu $softmax$ (vzorec 3.10). [32]

4.1 Trénování

Trénování RNN je výpočetně náročnější než trénování dopředných sítí, a to jednak z důvodu časových závislostí mezi vstupy, které zabraňuje efektivnímu výpočtu na grafické kartě, ale také kvůli explodujícímu a mizejícímu gradientu [25]. K trénování se používá modifikovaný algoritmus zpětného šíření chyby, jenž je schopný zohledňovat chyby z předchozích časových úseků. Označuje se potom jako algoritmus zpětného šíření chyby v čase nebo také anglicky backpropagation through time (BPTT).

Algoritmus zpětného šíření chyby v čase

Algoritmus zpětného šíření chyby v čase je založen na převedení rekurentní struktury neuronové sítě na dopřednou strukturu. Pokud budeme uvažovat RNN z obrázku 4.1, tak rozvinutí pro 3 časové úseky bude mít podobu zobrazenou na obrázku 4.2. V čase t nemá skrytá vrstva žádný vstup z toho důvodu, že se jedná o první prvek sekvence, v takovémto případě se většinou nahrazuje vstup vektorem nul. Všechny vyobrazené skryté vrstvy (červené neurony) sdílí svoje matice vah, modré spojení představují váhy směřující ze vstupní vrstvy do skryté vrstvy, fialové spojení pak spojení skryté vrstvy s výstupní vrstvou, v každém časovém kroku se jedná o ty samé váhy. V takto rozvinuté síti již jde jednoduše aplikovat klasický algoritmus pro zpětné šíření chyby (sekce 3.2.1) s tím rozdílem, že finální úpravy vah se zprůměrují přes všechny dílčí úpravy vah v jednotlivých časových úsecích.



Obrázek 4.2: Rozvinutá RNN pro 3 časové kroky, modrou barvou jsou reprezentovány váhy W_{xh}, zelenou potom W_{hh} a fialovou W_{hy}.

4.2 LSTM

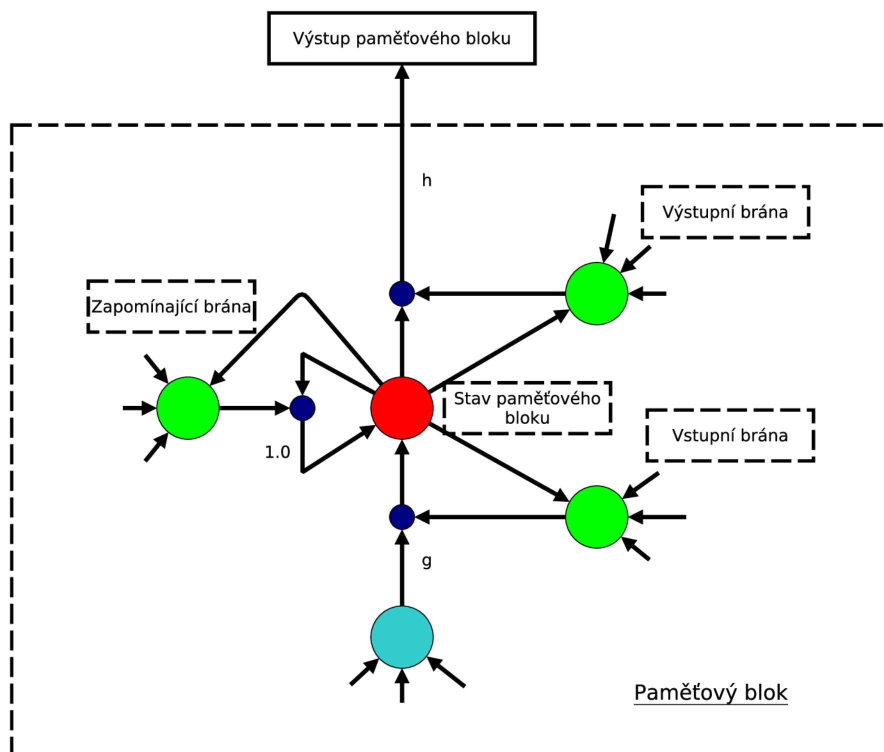
Long short-term memory sítě patří do skupiny rekurentních neuronových sítí, avšak mají daleko složitější strukturu jednotlivých neuronů. Představují způsob jak se vypořádat s problémem mizejícího gradientu, jejich struktura obsahuje prvky určené k zachování dlouhodobé informace. Byly poprvé představeny v [16], následně byla jejich struktura vylepšena o mechanizmus zapomínajících bran, které umožnily neuronové síti efektivně zapomínat nepotřebné informace [6]. Další vylepšení představoval mechanizmus peepholes, který umožňoval neuronové síti zpřesnit predikci jednotlivých událostí[7]. Tato sekce čerpá informace z [9].

Struktura

LSTM neuronová síť se skládá ze sady dopředných vrstev, které propagují svůj výstup jak následující vrstvě tak i do budoucnosti, jednotlivé vrstvy se skládají z tzv. paměťových bloků, struktura paměťového bloku je zobrazena na obrázku 4.3. Jednotlivé prvky paměťového bloku plní následující funkci:

- vstupní brána - kontroluje průchod informace do paměťového bloku
- stav paměťového bloku - uchovává dlouhodobou informaci
- zapomínající brána - určuje množství uchovávaného signálu
- výstupní brána - určuje množství propagovaného signálu

Paměťový blok může obsahovat více stavových buněk, které uchovávají stav paměťového bloku.



Obrázek 4.3: **LSTM paměťový blok.** Skládá se z vnitřního stavu, který je udržován pomocí fixní váhy 1.0 a ze tří bran, které jsou vyobrazeny zelenou barvou, dále obsahuje multiplikativní jednotky (zobrazeny modrou barvou), které násobí mezi sebou svůj vstup. Písmena g a h reprezentují aplikaci nelineární funkce.

Dopředná propagace signálu

Následující rovnice představují propagaci signálu z jednoho paměťového bloku, jejich pořadí je důležité. Jako u ostatních typů rekurentních neuronových sítí se informace z předešlého časového kroku, jenž není k dispozici, nahrazuje vektorem nul. Horní index jednotlivých proměnných představuje časový úsek, dolní indexy ι, ϕ, ω, c po řadě představují příslušnost vah spojení nebo aktivací k vstupní bráně, zapomínající bráně, výstupní bráně a ke stavům paměťového bloku. Proměnné I, H, C představují po řadě celkové množství vstupů dané vrstvy, výstupů dané vrstvy a množství stavových buněk v popisovaném paměťovém bloku. w_{ij} představuje váhu spojení jednotky i s jednotkou j, x_i^t je vstup paměťového bloku na pozici i v čase t, vstupní suma modulu j paměťového bloku v čase t se označuje jako a_j^t, někdy se taky označuje jako preaktivace, po aktivaci pak b_j^t. b_c^t představuje výstup paměťového bloku, žádným jiným způsobem se signál nepropaguje následujícím vrstvám. Funkce f je aktivační funkcí bran, funkce g a h jsou vstupně výstupními aktivacemi pamětového bloku.

Vstupní brána

$$a_\iota^t = \sum_{i=1}^{I} w_{i\iota} x_i^t + \sum_{h=1}^{H} w_{h\iota} b_h^{t-1} + \sum_{c=1}^{C} w_{c\iota} s_c^{t-1} \tag{4.3}$$

$$b_\iota^t = f(a_\iota^t) \tag{4.4}$$

Zapomínající brána

$$a_\phi^t = \sum_{i=1}^{I} w_{i\phi} x_i^t + \sum_{h=1}^{H} w_{h\phi} b_h^{t-1} + \sum_{c=1}^{C} w_{c\phi} s_c^{t-1} \tag{4.5}$$

$$b_\phi^t = f(a_\phi^t) \tag{4.6}$$

Stav paměťového bloku

$$a_c^t = \sum_{i=1}^{I} w_{ic} x_i^t + \sum_{h=1}^{H} w_{hc} b_h^{t-1} \tag{4.7}$$

$$s_c^t = b_\phi^t s_c^{t-1} + b_\iota^t g(a_c^t) \tag{4.8}$$

Výstupní brána

$$a_\omega^t = \sum_{i=1}^{I} w_{i\omega} x_i^t + \sum_{h=1}^{H} w_{h\omega} b_h^{t-1} + \sum_{c=1}^{C} w_{c\omega} s_c^t \tag{4.9}$$

$$b_c^t = f(a_\omega^t) \tag{4.10}$$

Výstup paměťového bloku

$$b_c^t = b_\omega^t h(s_c^t) \tag{4.11}$$

Zpětná propagace chyby

LSTM sítě náleží do skupiny rekurentních neuronových sítí, a proto se pro jejich trénování používá algoritmus BPTT, kdy dochází ke zpětnému šíření chyby jak do následující vrstvy tak i do předchozího časového kroku. Pokud budeme uvažovat objektivní funkci O použitou při trénování a funkci definující výstup paměťového bloku jako b_c^t, tak můžeme definovat chybu pro paměťový blok v čase t následovně:

$$\epsilon_c^t \stackrel{\text{def}}{=} \frac{\partial O}{\partial b_c^t} \tag{4.12}$$

Chybu pro stav paměťového bloku s_c^t v čase t definujeme následovně:

$$\epsilon_s^t \stackrel{\text{def}}{=} \frac{\partial O}{\partial s_c^t} \tag{4.13}$$

Označme velikost následující vrstvy K, pak chyba bloku odpovídá chybě propagované z následující vrstvy a také chybě, která je propagovaná z následujícího časového kroku:

$$\epsilon_c^t = \sum_{k=1}^{K} w_{ck}\delta_k^t + \sum_{h=1}^{H} w_{ch}\delta_h^{t+1} \tag{4.14}$$

Tedy δ_w^t můžeme vypočítat následovně:

$$\delta_w^t = f'(a_w^t)\sum_{c=1}^{C} h(s_c^t)\epsilon_c^t \tag{4.15}$$

Chybu pro stav paměťového bloku můžeme získat propagací chyby ze všech komponent, jež jsou napojeny na stav paměťového bloku:

$$\epsilon_s^t = b_\omega^t h'(s_c^t)\epsilon_c^t + b_\phi^{t+1}\epsilon_s^{t+1} + w_{c\iota}\delta_\iota^{t+1} + w_{c\phi}\delta_\phi^{t+1} + w_{c\omega}\delta_\omega^t \tag{4.16}$$

Následný výpočet delty δ_c^t pro paměťový stav buňky:

$$\delta_c^t = b_\iota^t g'(a_c^t)\epsilon_s^t \tag{4.17}$$

Zbylé delty bran je možné vypočítat následovně:

$$\delta_\phi^t = f'(a_\phi^t)\sum_{c=1}^{C} s_c^{t-1}\epsilon_s^t \tag{4.18}$$

$$\delta_\iota^t = f'(a_\iota^t)\sum_{c=1}^{C} g(a_c^t)\epsilon_s^t \tag{4.19}$$

4.3 GRU

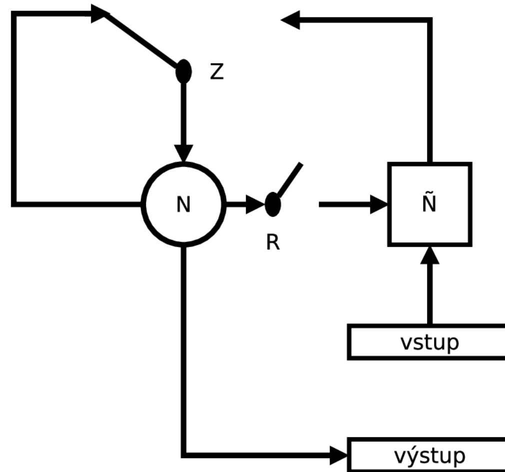
Gated recurrent unit jsou inspirovány částečně architekturou LSTM (sekce 4.2). Podobně jako LSTM se části jediné vrstvy člení do bloků. Jeden blok pak obsahuje dva typy bran r a z, kde brána r plní podobnou funkci jako zapomínající brána v architektuře LSTM a brána z řídí výstupní interpolaci mezi současným a předcházejícím výstupem. Nemají však paměťové buňky a počet parametrů, které obsahují jednotlivé bloky, je výrazně menší. Pokud bychom tedy měli dvě neuronové sítě složené ze stejného počtu bloků GRU a LSTM, pak by se síť, jež tvoří GRU bloky, trénovala rychleji a také zabírala méně paměti než síť tvořená LSTM bloky. GRU architektura byla poprvé publikována v [3]. Její grafická abstrakce je k dispozici na obrázku 4.4. Vrstva tvořená GRU bloky je charakterizovaná následujícími vzorci:

$$z_t = \sigma(W_z x_t + U_z n_{t-1}) \tag{4.20}$$

$$r_t = \sigma(W_r x_t + U_r n_{t-1}) \tag{4.21}$$

$$\tilde{n}_t = \tanh(W x_t + U(r_t \cdot n_{t-1})) \tag{4.22}$$

$$n_t = (1 - z_t) \cdot n_{t-1} + z_t \cdot \tilde{n}_t \tag{4.23}$$



Obrázek 4.4: Grafická reprezentace architektury GRU.

4.4 IRNN

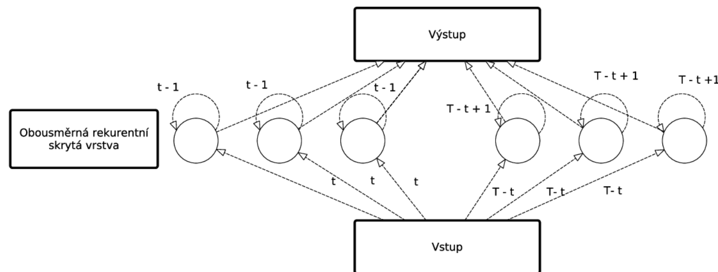
IRNN představují klasickou rekurentní neuronovou síť, jejíž aktivační funkce je ReLU (vzorec 3.6) a na počátku je matice vah rekurentních spojení inicializovaná na jednotkovou matici a vektor biasů je inicializován na nuly. Tento druh inicializace zajišťuje, že chyba, která se propaguje v čase během BPTT, se nemění, a je se tedy schopná adaptovat podobně jako LSTM na události, které jsou od sebe vzdálené. [19]

4.5 Obousměrné rekurentní sítě

V obousměrné rekurentní neuronové síti jsou jednotlivé vrstvy rozděleny do dvou částí, každá z těchto částí zpracovává vstupní posloupnost samostatně v opačném pořadí a jejich konkatenovaný výstup je propagován do následující vrstvy.

Uvažujme množinu vstupů X, kterou tvoří prvky x_t, jež jsou na sobě časově závislé a obsahují celkově T jednotlivých vstupů, pak část první obousměrné skryté vrstvy, která zpracovává sekvenci v dopředném směru, bude zpracovávat při prvním kroku vstup x_1 a zpětná část první obousměrné skryté vrstvy bude zpracovávat prvek x_T, při druhém kroku to bude pro dopřený průchod x_2 a zpětná část bude zpracovávat x_{T-1}. Na výstupu odpovídajícímu prvnímu vstupu pak bude výstup dopředné vrstvy pro první vstup a výstup zpětné části pro celou sekvenci.

V případě trénování pak pro algoritmus zpětného šíření chyby platí, že se nejprve vypočte chyba dopředné části obousměrné skryté vrstvy, následně chyba zpětné části obousměrné skryté vsrtvy, přičemž se jimi chyba propaguje přesně v opačném směru, než tomu bylo v případě dopředného průchodu neuronovou sítí. Tyto chyby se sečtou a následně propagují do nižší vrstvy. Topologie obousměrné sítě je zobrazena na obrázku 4.5. Tohoto mechanizmu bylo poprvé využito v [27].



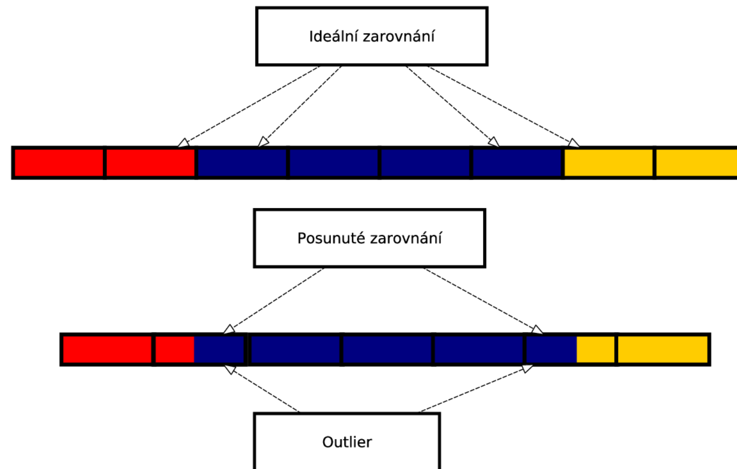
Obrázek 4.5: Architektura obousměrné rekurentní vrstvy.

4.6 Connectionist temporal classification

Connectionist temporal classification (CTC) je objektivní funkce, která se používá pro klasifikaci u sekvenčních dat. Byla publikovaná v [11]. Není u ní vyžadováno přesné zarovnání vstupních dat na výstupní třídy. Jejím cílem je maximalizovat taková ohodnocení posloupností výstupních tříd, která po dekódování vedou na správné řetězce. Tato vlastnost je velice užitečná, protože při zpracovávání sekvenčních dat často nejsou k dispozici výstupní třídy pro konkrétní časové kroky.

Uvažujme nyní problém převodu zvuku na řeč. Pro tento problém se může použít DNN akustický model, který může být trénován s pomocí objektivní funkce cross entropie. Tento přístup vyžaduje znalost výstupní třídy pro každý časový krok, který bývá většinou 10 milisekund. Zarovnání jednotlivých fonémů nemusí být vždy přesné, v jednom rámci se můžou překrývat dva fonémy. Výsledné spektrum takovéhoto rámce neodpovídá ani jednomu z těchto fonémů, a přesto bude neuronová síť donucena se naučit takovýto rámec klasifikovat jako jeden z nich. Tato hodnota bude outlier a bude mít negativní vliv na trénování neuronové sítě. V případě CTC k takovémuto jevu nedochází, protože se klasifikace většinou zaměřuje pouze na jeden konkrétní rámec, u něhož je evidentní, do které třídy patří, a všechny ostatní rámce zařadí do třídy *blank*. Demonstrace tohoto jevu je zobrazena na obrázku 4.6. V podstatě představuje CTC řešení všude tam, kde nejsou jednoznačně určeny hranice mezi jednotlivými výstupními třídami, ale důležitou roli hraje pořadí výstupních tříd, což je typicky problém ve zpracování řeči, ale také v jiných odvětvích.

U CTC se stejně jako v případě cross entropie využívá softmax funkce (vzorec 3.10), která je rozšířena o jednu třídu, která reprezentuje *blank*. Aktivace prvních L neuronů jsou interpretovány jako pravděpodobnosti pozorování daných tříd v daném časovém okamžiku. Poslední neuron potom odpovídá pravděpodobnosti pozorování třídy, která neodpovídá ani jedné z prvních L. Uvažujme vstupní posloupnost x a definujme výstup neuronu odpovídající třídě k v čase t jako y_k^t, dále pak uvažujme množinu všech výstupních neuronů sjednocenou se třídou *blank* a referujme ji jako L'. Potom pravděpodobnost výskytu jakékoliv cesty π, jež tvoří prvky z L' na sobě nezávislé, můžeme vyjádřit následovně:



Obrázek 4.6: Demonstrace posunu vzorkování a vzniku outlierů. V případě CTC by byl klasifikován pouze prostřední rámec, který obsahuje střed fonému, ostatní rámce by mohly být reprezentovány třídou *blank*.

$$p(\pi|x) = \prod_{t=1}^{T} y_k^{\pi_t} \tag{4.24}$$

Uvažujme nyní $L_T^{'}$ jako cesty a referujme je jako π a zaveďme následující mapování β mezi cestami tvořenými prvky z $L^{'}$ a L. V následující ukázce vlastností β se považuje symbol $-$ za třídu *blank* a množina L je tvořena prvky a, b.

- $\beta(aa - abb - -) = aab$
- $\beta(a - -ab-) = aab$

Intuitivně dochází k odstranění několikanásobného výskytu stejné třídy, pokud následuje bezprostředně za sebou, a také dochází k odstranění všech výskytů třídy *blank*.

Nyní můžeme vyjádřit pravděpodobnost výskytu sekvence l, jež tvoří prvky z L, pomocí mapování β následovně:

$$p(l|x) = \sum_{\pi \in \beta^{-1}(l)} p(\pi|x) \tag{4.25}$$

, což je hlavní myšlenkou celého CTC. Snahou následně bude maximalizovat všechny pravděpodobnosti takových zarovnání, které vedou na korektní výslednou posloupnost tříd. Výstupem klasifikátoru by potom měla být taková posloupnost výstupu, která maximalizuje pravděpodobnost dané sekvence vzhledem k vstupním datům:

$$h(x) = \operatorname*{argmax}_{l \in L^{\leq T}} p(l|x) \tag{4.26}$$

Obecně existuje velké množství různých posloupností, které různě kombinují výstupní ohodnocení a vedou na stejný výstupní řetězec. Abychom mohli efektivně vypočíst pravděpodobnost $p(l|x)$, je nutné použít *backward-forward* algoritmus, který se používá u HMM (sekce 2.2). Pro jednodušší finální dekódování se ještě předpokládá, že mezi každou dvojicí

výstupu je vložen symbol *blank* a také na začátek a konec je vložen symbol *blank*. Potom minimální uvažovaná délka možné posloupnosti výstupních tříd l' vedoucí na korektní výsledný řetězec l bude $2 * |l| + 1$.

Dekódování

V originálním článku byly zmíněny dvě metody, pomocí kterých lze najít nějakou preferovanou cestu. První triviální způsob se nazývá *best path decoding*, v nemž se považuje za nejpravděpodobnější výstupní třídu ta třída, jež má v daném časovém okamžiku přiřazenou největší posteriorní pravděpodobnost. Na takto získanou posloupnost se aplikuje mapování β, jehož výsledek představuje optimální cestu.

$$h(x) \approx \beta(\pi^*) \text{ , kde } \pi^* = \operatorname*{argmax}_{p \in N^t} p(\pi|x) \tag{4.27}$$

Je zřejmé, že tento způsob není optimální, protože posloupnost výstupních tříd, jejíž pravděpodobnostní ohodnocení je v jistém časovém okamžiku na stabilní úrovni, může být v každý takovýto okamžik vždy zastíněna jinou třídou, která má pouze momentálně výrazné pravděpodobnostní ohodnocení, které v zápětí ztrácí.

Optimálním algoritmem z teoretického hlediska je *prefix search decoding*. Tento algoritmus vždy expanduje nalezený prefix výstupní posloupnosti tak dlouho, dokud je jeho ohodnocení větší než jakýkoliv jiný prefix. Tento proces je výpočetně náročný a v praxi se často uplatňuje nějaká heuristika, která určí uměle hranice pro délku prefixu.

Objektivní funkce

Aby bylo možno využít CTC při trénování neuronové sítě, tedy používat algoritmus zpětného šíření chyby v čase (sekce 4.1), definujme objektivní funkci jako snahu maximalizovat pravděpodobnost všech výstupních ohodnocení, která vedou na správnou výslednou sekvenci:

$$O_{CTC}(W, b; S) = - \sum_{\{x,z\} \in S} ln(P(z|x)) \tag{4.28}$$

Jednotlivé vstupní sekvence jsou nezávislé, můžeme je tedy zpracovávat nezávisle, derivaci vzhledem k výstupu pro jednu sekvenci definujme následovně:

$$\frac{\partial O_{CTC}(W, b; \{x, z\})}{\partial y_k^t} = -\frac{\partial ln(p(z|x))}{\partial y_k^t}, \tag{4.29}$$

pro efektivní výpočet derivace je nutné získat pravděpodobnosti cest procházejících danou výstupní třídou k v čase t. Pro tento výpočet se opět využívá *backward-forward* algoritmus podobně jako HMM, výsledný vzorec definující chybu je uveden v následující rovnici, pro podrobnosti o jeho odvození je doporučen originální článek [11].

$$\frac{\partial O_{CTC}(W, b; \{x, z\})}{\partial y_k^t} = y_k^t - \frac{1}{y_k^t Z_t} \sum_{s \in lab(z,k)} \hat{\alpha}_t(s) \hat{\beta}_t(s) \tag{4.30}$$

$$Z_t \stackrel{\text{def}}{=} \sum_{s=1}^{l'} \frac{\alpha_t \hat{(s)} \beta_t \hat{(s)}}{y_{l'_s}^t} \tag{4.31}$$

Symboly $\alpha_t\hat{(}s)$ a $\beta_t\hat{(}s)$ značí normalizované dopředné a zpětné proměnné.

Kapitola 5

Implementace

Pro implementaci rekurentních sítí byla zvolena knihovna *torch* [4], která je napsaná v jazyce *lua*. Implementace byla částečně inspirována frameworkem *currennt* [30] a toolkitem *eesen* [22].

5.1 Torch

Torch je knihovna jazyka *lua*, která podporuje širokou škálu matematických operací lineární algebry. Základní třídou je takzvaný *Tensor*, jenž reprezentuje matice a obsahuje efektivní implementace matematických operací lineární algebry na CPU. V pozadí každého objektu typu *Tensor* je objekt typu *Storage*, který představuje skutečnou alokaci místa pro danou matici v paměti RAM. Zajímavou vlastností tohoto mechanizmu je možnost sdílení tohoto objektu napříč více objekty typu *Tensor*. Pokud dojde ke zmenšení obsahu *Tensoru*, neprojeví se to na velikosti objektu typu *Storage*, dojde-li však ke zvětšení matice a v současné době není alokován dostatek místa, dojde k navýšení kapacity objektu typu *Storage*. Tato strategie je zvolena z důvodu efektivní manipulace pamětí na grafické kartě. Knihovna *nn* využívá torch pro implementaci modulů, jež jsou často používány pro budování neuronových sítí.

Pro provádění matematických operací na grafické kartě je nutné použít knihovnu *cutorch* a pro implementaci modulů z knihovny *nn* na grafické kartě je nutno použít knihovnu *cunn*.

Vytváření modelů v knihovně *nn* je založeno na skládání jednoduchých modulů do komplexnějších celků, kdy každý jednoduchý modul má definováno chování pro dopředný a zpětný průchod. Takto složené moduly nejsou nějak optimalizovány po stránce efektivního výpočtu nebo paměťové náročnosti narozdíl od *theana* [2] nebo *TensorFlow* [1], kdy dochází k optimalizaci výpočetního grafu.

Jednoduché moduly se skládají prostřednictvím tří hlavních kontejnerů do komplexnějších celků:

- *nn.Sequential* - kontejner, jenž propaguje vstupní signál sekvenčně napříč svými komponentami
- *nn.Parallel* - kontejner, jenž propaguje vstupní signál paralelně všem komponentám, pokud je k dispozici více jader, dojde k vykonání dopředného a zpětného průchodu paralelně
- *nn.ConcatTable* - kontejner, jenž propaguje identický vstupní signál všem svým komponentám paralelně, opět pokud je k dispozici více jader, dojde k rozložení výpočtu

Každý jednoduchý modul knihovny *nn* musí povinně obsahovat dvě metody, a sice *forward* a *backward*, tyto metody by neměly být nikdy modifikovány. Namísto jejich modifikace by mělo docházet k modifikaci následujících metod:

- *updateOutput* - metoda zajišťuje implementaci dopředného průchodu, kdy je výsledek uložen do pole output
- *updateGradInput* - metoda zajišťuje výpočet chyby pro předchozí vrstvu (vzorec 3.22) a uložení výsledku do pole *gradInput*
- *accGradParameters* - metoda zajišťuje výpočet a akumulaci změn vah v poli *gradWeight* (vzorec 3.15) a výpočet a akumulaci změn biasu v poli *gradBias* (vzorec 3.16)

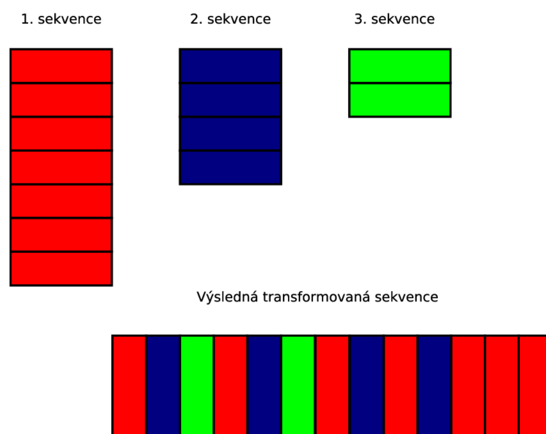
Ukázka vytvoření klasické dopředné neuronové vrstvy, která obahuje 10 neuronů a má pět vstupů s nelineární aktivací $Tanh$ v jazyce *lua* pomocí knihovny *nn*:

```
require 'nn'
local model = nn.Sequential()
model:add(nn.Linear(5, 10))
model:add(nn.Tanh())
```

Tímto postupem s výpomocí několika desítek dalších modulů lze vytvořit i daleko komplikovanější architektury jako například LSTM.

Každý modul má podporu pro několik vstupů, tedy jeho vstupem je několik příznaků, které se označují jako dávka. Tohoto principu bylo využito pro implementaci algoritmu BPTT (sekce 4.1), kdy se jednotlivé rekurentní moduly tváří, jako by zpracovávaly klasickou dávku, ovšem vnitřně je tento vstup rozdělen do jemnější struktury časových dávek. Tento princip umožňuje bezproblémovou integraci s dalšími existujícími moduly, které nejsou rekurentní - typicky klasické dopředné nebo konvoluční vrstvy. Při zpětné propagaci chyby je opět zohledněna časová struktura dávky a k jednotlivým rekurentním částem je propagována chyba i z budoucího časového okamžiku.

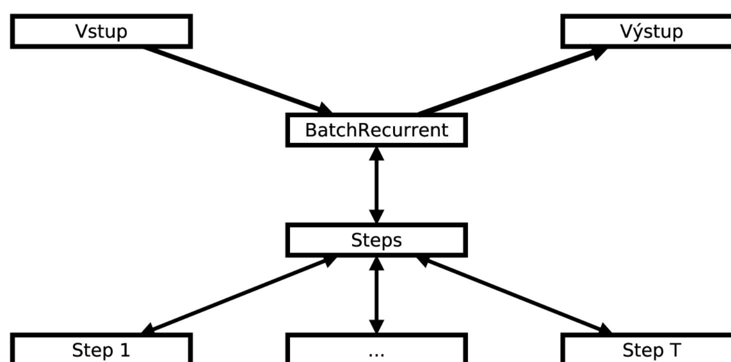
Na obrázku 5.1 je naznačena transformace jednotlivých vstupních sekvencí do jedné konkatenované sekvence, která za sebou vždy obsahuje vektory odpovídající jednomu kroku ve všech sekvencích:



Obrázek 5.1: Transformace několik různě dlouhých sekvencí do jedné určené k trénování neuronové sítě.

5.2 Implementace rekurence

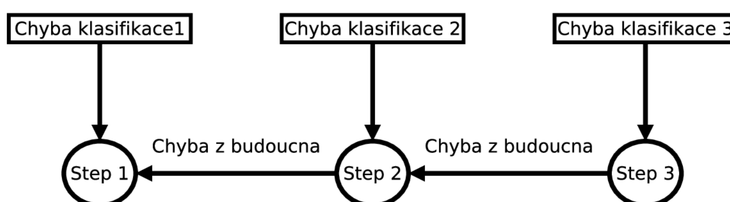
Základem všech implementovaných rekurentních modulů jsou třídy *BatchRecurrent*, *Steps* a *Step*, jejichž deriváty následně tvoří daný typ rekurentní architektury. Každou matematickou operaci v rekurentní architektuře můžeme rozdělit do dvou tříd na základě toho, jestli operace pracuje přímo se vstupními daty a nebo pracuje s výstupem neuronů z minulého časového okamžiku. Pokud pracuje pouze se vstupními daty, je možné tuto operaci předvypočíst napříč všemi časovými okamžiky, uložit výsledky do paměti a zpřístupnit je pro budoucí použití. Tento postup zajišťuje zrychlení výpočtu, dojde ale k navýšení spotřeby paměti. Tento typ operací je reprezentován třídou *BatchRecurrent*. Výpočet druhého typu operací je závislý na předchozím výstupu, a nelze jej tudíž takto paralelizovat. Pokud ovšem zpracováváme vícero sekvencí najednou, je možné v jednom okamžiku vypočíst paralelně výstup pro daný časový okamžik u více sekvencí. Všechny tyto operace jsou reprezentovány třídou *Step*. Pro každý časový krok je vytvořen jeden objekt typu *Step* pomocí objektu typu *Steps*, kdy třída *Steps* zajišťuje správu těchto objektů, ale také distribuci vstupů a chyb do jednotlivých časových kroků reprezentovaných třídou *Step*. Graficky je interakce těchto tříd zobrazena na obrázku 5.2.



Obrázek 5.2: Architektura implementovaného řešení pro reprezentaci RNN.

Algoritmus zpětného šíření chyby v čase

Algoritmus zpětného šíření chyby v čase není explicitně implementován, celý model se opět chová jako typická dopředná neuronová síť. Jednotlivé objekty typu *Step* jsou mezi sebou provázány, uchovávají si odkazy na objekty reprezentující předchozí a budoucí časový okamžik, tvoří obousměrně vázaný seznam. Při dopředném průchodu si každý objekt typu *Step* ke svému standardnímu vstupu zjistí i výstup předchozího časového kroku a zahrne tento vstup do svého výpočtu. V závislosti na komplikovanosti daného typu rekurentní archtitektury se může jednat pouze o celkový výstup nebo i výstup nějakého specifického submodulu. Při zpětném průchodu obdrží každý objekt typu *Step* chybu, která nastala v důsledku špatné klasifikace, a zároveň si vyžádá chybu, která vznikla v budoucím časovém okamžiku a měla by se propagovat dále. Charakter této chyby a její aplikace je opět závislá na složitosti dané architektury. Celý proces je zjednodušeně ilustrován na 5.3.



Obrázek 5.3: Architektura implementovaného řešení pro reprezentaci RNN.

Obousměrné RNN

Obousměrné rekurentní sítě jsou reprezentovány třídou *Bidirectional*, tento modul obsahuje vždy dva deriváty třídy *Steps*. Třída *Steps* obsahuje parametr *revert*, který určuje, v jakém směru se má vstupní posloupnost distribuovat při dopředném průchodu, a také ovlivňuje distribuci posloupnosti chyb při zpětném průchodu do jednotlivých modulů typu *Step*. Pro realizaci obousměrné rekurentní sítě je vždy parametr *revert* nastaven u jednoho objektu *Steps* na pozitivní hodnotu a u druhého na negativní. Výstup jednotlivých objektů *Steps* je konkatenován při dopředném průchodu, při zpětném průchodu je potom chyba distribuovaná každému modulu zvlášť.

5.3 Moduly

V následující sekci budou popsány implementované moduly.

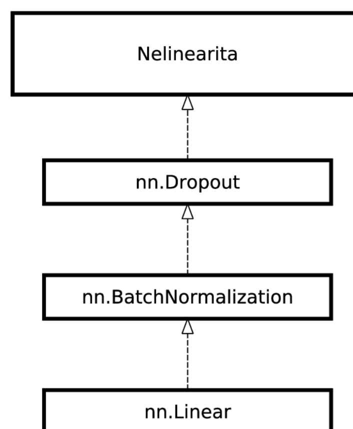
BatchRecurrent

Jak již bylo zmíněno, tento modul zajištuje výpočet všech operací, které je možné provést napříč celou historií vstupů. Z analýzy jednotlivých typů neuronových sítí vyplynul následující seznam operací, které pracují pouze se vstupem z předešlé neuronové sítě.

- RNN
  - $W_{xh}x_t$ (vzorec 4.1)

- LSTM
  - $\sum_{i=1}^{I} w_{i\iota} x_i^t$ (vzorec 4.3)
  - $\sum_{i=1}^{I} w_{i\phi} x_i^t$ (vzorec 4.5)
  - $\sum_{i=1}^{I} w_{ic} x_i^t$ (vzorec 4.7)
  - $\sum_{i=1}^{I} w_{i\omega} x_i^t$ (vzorec 4.9)
- GRU
  - $W_z x_t$ (vzorec 4.20)
  - $W_r x_t$ (vzorec 4.21)
  - $W x_t$ (vzorec 4.22)

Všechny tyto vzorce jsou afinní transformace, jenž mohou být realizovány pomocí modulu *nn.Linear*, liší se pouze množstvím výstupních neuronů. Na všechny takto získané výstupy lze aplikovat techniku batch normalizace pomocí modulu *nn.BatchNormalization* a dropout prostřednictvým modulu *nn.Dropout*. Tyto operace se budou vykonávat sekvenčně. Z tohoto důvodu je rodičem *BatchRecurrent* třída *nn.Sequential*. Experimentálně bylo zjištěno, že umístění batch normalizace do této části výpočtu produkuje nejlepší zlepšení vzhledem k rychlosti trénování. Batch normalizaci je možné umístit téměř kamkoliv, pokud se nejedná o rekurentní vazby a vždy povede k jistému zlepšení, ovšem je nutné počítat s delší dobou trénování. Experimentálně také bylo zjištěno, že umístění techniky dropout do této fáze výpočtu je také nejvhodnější vzhledem k jiným umístěním. Na obrázku 5.4 je graficky zobrazena preferovaná varianta kombinace batch normalizace a dropoutu.



Obrázek 5.4: Obecné schéma pro kombinaci batch normalizace, dropoutu, lineární hladiny a nelinearity.

LinearScale

Jedná se o třídu, která realizuje výpočet *peephole* spojení u architektury LSTM. V rámci knihovny *nn* lze i tento jednoduchý modul považovat za vrstvu neuronové sítě a jako taková musí mít metodu definující její dopředný průchod, zpětný průchod a vnitřně musí realizovat výpočet gradientu vzhledem k parametrům. Její dopředný průchod je realizován následujícím vzorcem:

$$f(x) = W \cdot x, \tag{5.1}$$

kde W je vektor parametrů a x je vstupní vektor. Zpětný průchod můžeme odvodit ze vzorce 3.22:

$$\epsilon^{l-1} = \epsilon^{l} \cdot W. \tag{5.2}$$

Výpočet gradientů vzhledem k parametrům W pak bude definován následovně:

$$\nabla_W f(x) = \epsilon^{l-1} \cdot x. \tag{5.3}$$

AddLinear

Je pozměněnou implementací modulu *nn.Linear*. Modifikace tkví v provedení celé operace v předalokovaném poli. Výsledek není do pole nahrán, ale je přičten ke stávajícím hodnotám. Při zpětné propagaci chyby dojde k duplikaci propagované chyby tak, aby tento modul mohl být klasicky využit pří sestavování komplexnějších modulů.

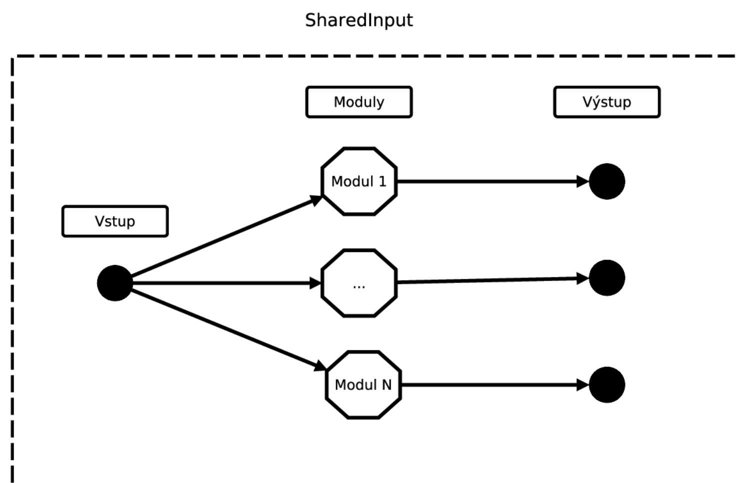
SharedInput

SharedInput je třída, která obaluje několik jiných objektů. Při dopředném průchodu všem svým objektům propaguje hodnoty na vlastním vstupu sekvenčně a takto získané výsledky od všech modulů předá nadřazenému modulu. Při zpětném průchodu distribuuje sekvenčně chybu jednotlivým modulům a akumuluje ji před jejím předáním předcházejícímu modulu. Tento modul bylo nutné implementovat pro efektivní implementaci obousměrných rekurentních sítí (sekce 5.2), kdy se jeden vstup propaguje dvěma dalším objektům. Tato implementace je efektivní z paměťového hlediska, protože není nutné, aby si každý modul udržoval výstupní chybu při zpětném průchodu, jako je tomu u modulu *nn.ConcatTable*. Funkce této třídy je vyobrazena na obrázku 5.5.

LSTM

LSTM architektura je reprezentovaná třídami *LstmSteps*, *LstmStep*, *Lstm*. Rekurentní výpočet je realizován ve třídě *LstmStep*. Pokud nahlédneme na výpočty dopředných průchodů jednotlivých bran a stavu paměťového bloku (sekce 4.2) u architektury LSTM, zjistíme, že je celkový potenciál a vždy tvořen členem, jež zpracovává data z předchozího časového kroku. Konkrétně se jedná o tyto operace:

- $\sum_{h=1}^{H} w_{h\iota} b_h^{t-1}$ (vzorec 4.3),
- $\sum_{h=1}^{H} w_{h\phi} b_h^{t-1}$ (vzorec 4.5),



Obrázek 5.5: Demonstrace dopředného průchodu u třídy SharedInput.

- $\sum_{h=1}^{H} w_{hc} b_h^{t-1}$ (vzorec 4.7),
- $\sum_{h=1}^{H} w_{h\omega} b_h^{t-1}$ (vzorec 4.9).

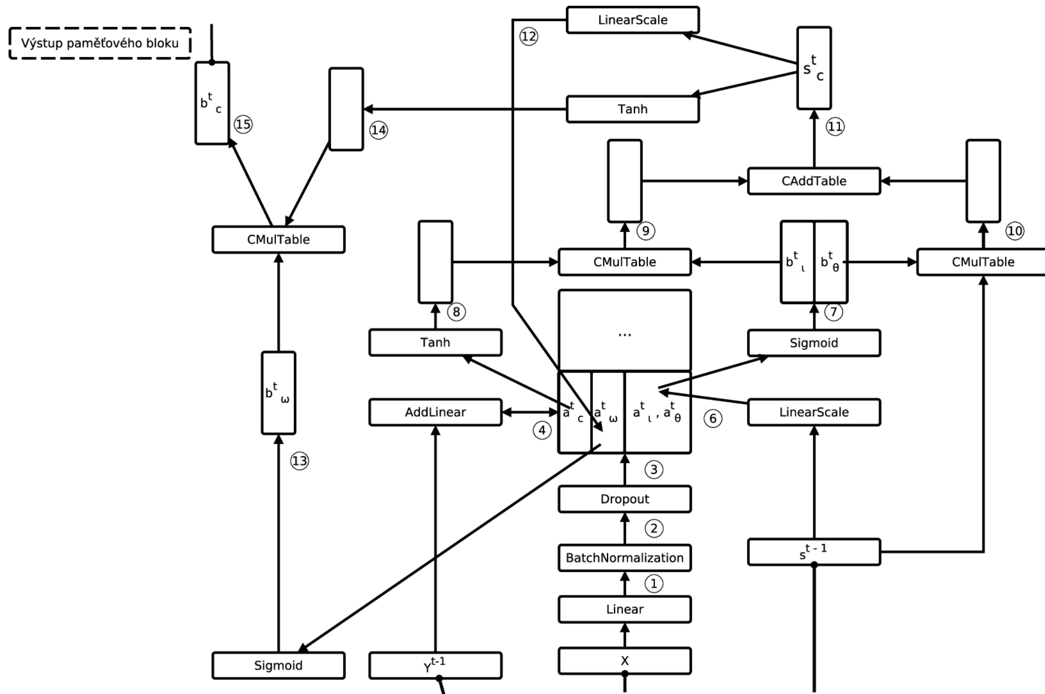
Jedná se o afinní transformace, které mohou být provedeny najednou pomocí modulu *AddLinear* (sekce 5.3). Dochází k jedné modifikaci vůči teoretickému návrhu, a tou je přidání biasu, který je u zapomínající brány inicializován na hodnotu 1. Tento trik by měl umožnit zachycení dlouhodobých vzorů při trénování.

Následně dojde k vypočítání *peephole* spojení pro vstupní a zapomínající bránu pomocí modulu *LinearScale*. V tuto chvíli je již možné provést aplikaci nelinearity logistické sigmoidy pomocí modulu *nn.Sigmoid*, tento výpočet je realizován paralelně pro zapomínající i vstupní bránu. Poté dojde k aplikaci nelinearity pomocí modulu *nn.Tanh* na preaktivaci paměťového bloku. Multiplikativní jednotky jsou realizovány pomocí modulu *nn.CMulTable*. Jejich aplikací dojde k získání stavu paměťového bloku s_c^t. Poté je vypočtena hodnota *peephole* spojení s výstupní branou. Nyní je již možno aplikovat nelinearitu logistické sigmoidy a získat tak konečnou hodnotu od výstupní brány. Na výstup paměťového bloku je aplikovaná nelinearita prostřednictvím modulu *nn.Tanh* a na tento výstup je aplikovaná multiplikativní jednotka spolu s výstupem výstupní brány. Celé toto schéma je názorně zobrazeno na obrázku 5.6.

Z analýzy rovnic reprezentujících zpětných průchod skrz paměťový blok 4.2 vyplývá, že je nutné při zpětné propagaci chyby, přičíst k aktuální chybě propagované z nadřazené vrstvy také chybu, která vznikla v budoucím časovém kroku (vzorec 4.14). A před propagací chyby skrz stav paměťového bloku, je opět nutné přičíst chybu, která vznikla v důsledku použití *peephole* spojení se vstupní a zapomínající branou (vzorec 4.16). Propagace chyby bude mít přesně opačný charakter než má dopředný průchod z obrázku 5.6.

GRU

GRU architektura je reprezentovaná třídami *GruSteps*, *GruStep*, *Gru*. Při analýze rovnic v sekci 4.3 dojdeme k podobnému zjištění jako u architektury LSTM (sekce 5.3) a sice



Obrázek 5.6: **Implementační architektura LSTM vrstvy.** Y^{t-1} představuje výstup této vrstvy z minulého časového kroku, X představuje všechny zpracovávané sekvence v aktuální dávce, s^{t-1} představuje hodnoty stavů jednotlivých paměťových bloků z minulého časového kroku. Jednotlivá čísla u výstupu bloku popisují pořadí operace. Výstupy z operačních bloků představují paměť, do které je výsledek uložen.

brány r, z vykonávájí stejnou operaci nad výstupem z minulého časového kroku. Jedná se o následující operace:

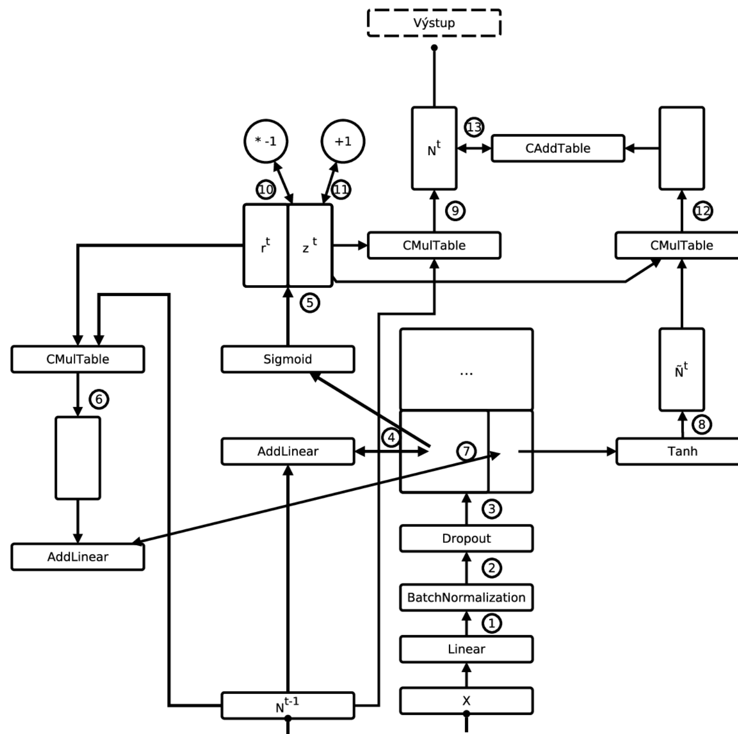
- $U_z n_{t-1}$ (vzorec 4.20),
- $U_r n_{t-1}$ (vzorec 4.21).

Tyto operace mohou být provedeny paralelně pomocí modulu *AddLinear*. Od teoretického návrhu se implementace liší přidáním biasu. Následuje aplikace *nn.Sigmoid* pro získání hodnot r a z. Poté následuje aplikace r na předchozí výstup pomocí modulu *nn.CMulTable* a aplikace modulu *AddLinear*. Po aplikaci nelinearity *nn.Tanh* můžeme přikročit k získání finální interpolované hodnoty. Celý tento proces je graficky zobrazen na obrázku 5.7.

Při zpětné propagaci chyby, která bude mít opět přesně opačný směr toku než dopředný průběh na obrázku 5.7, je nutné k chybě přicházející z nadřazené vrstvy taky přičíst chybu, která vznikla v budoucím časovém kroku.

5.3.1 RNN

Klasickou rekurentní síť představují třídy *RecLayer*, *RecSteps*, *RecStep*. Implementační schéma je zobrazeno na obrázku 5.8 pro aktivační funkci $Tanh$. Modul *AddLinear* reprezentuje operaci $W_{hh}h_{t-1}$ ze vzorce 4.1.



Obrázek 5.7: **Implementační architektura GRU vrstvy.** N^{t-1} představuje výstup této vrstvy z minulého časového kroku, X představuje všechny zpracovávané sekvence v aktuální dávce. Jednotlivá čísla u výstupu bloku popisují pořadí operací. Výstupy z operačních bloků představují paměť, do které je výsledek uložen.

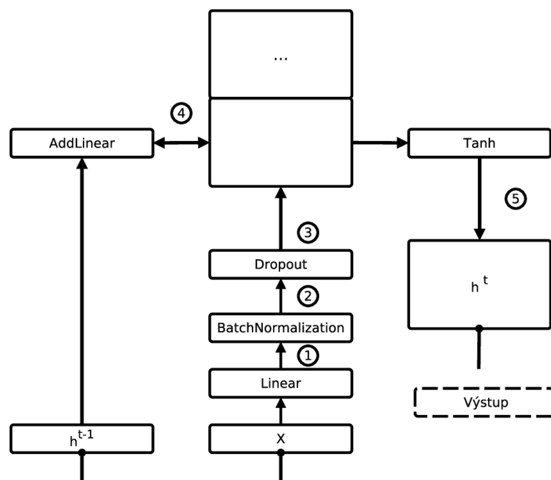
Zpětný průchod bude mít přesně opačný směr než dopředný průběh z obrázku 5.8. K chybě z nadřazené vrstvy bude nutné přičíst chybu z minulého časového kroku.

5.3.2 IRNN

Architektura IRNN je shodná s klasickou RNN. Jediným rozdílem je změna inicializace, kdy jsou biasy na počátku inicializovány na hodnotu 0 a matice parametrů reprezentující rekurentní vazby je inicializováná na jednotkovou matici.

5.3.3 Optimalizace

Pro optimalizaci využívání pole *gradInput*, do kterého je u každého modulu uložena chyba pro podřízený modul, byl zaveden objekt spravující množinu objektů typu *Storage*, kdy je každému typu modulu přiřazena jedna instance tohoto objektu a je následně sdílen mezi všemi vytvořenými objekty daného typu a využit pro uložení chyby z předchozí vrstvy. Tedy pokud se v dané architektuře neuronové sítě vyskytuje vícero instancí daného modulu, což je typické pro hluboké neuronové sítě, bude při zpětné propagaci zabráno maximálně tolik místa, kolik má prvků největší z propagovaných chyb pro daný modul. Pokud by byly vytvořeny dvě vrstvy po sobě tím samým modulem, dojde k chybě, protože by si navzájem začaly přepisovat chybový vektor. Nicméně tato situace by neměla nastat. Následně je ještě potřeba udělat výjimku u modulů, které potřebují získat chyby z vícero souběžně



Obrázek 5.8: **Implementační architektura RNN vrstvy.** $\tilde{h}^{t-1}$ představuje výstup této vrstvy z minulého časového kroku, X představuje všechny zpracovávané sekvence v aktuální dávce. Jednotlivá čísla u výstupu bloku popisují pořadí operací. Výstupy z operačních bloků představují paměť, do které je výsledek uložen.

vykonávaných paralelních modulů. V tomto případě byl vytvořen nový modul, který pracuje sekvenčně a chyby zpracovává také sekvenčně. Nicméně je možné použít i originální modul *nn.ConcatTable*, pro který bude vytvořeno vícero uložišť typu *Storage*. Tento mechanizmus nelze aplikovat na modul *nn.ParalelTable*.

Další paměťová optimalizace se týká rekurentních modulů a jejich replikace v čase. Obecně se v knihovně *nn* můžou hodnoty propagovat mezi jednotlivými moduly ve formě matic nebo tabulek, kdy tabulka je jediný dynamický typ, který jazyk *lua* obsahuje a je využíván jako pole, seznam nebo slovník. Z tohoto plyne, že si jednotlivé moduly mohou ukládat data ve dvou formátech. Knihovna *nn* obsahuje prostředky pro klonování jednotlivých objektů, které mezi sebou mohou sdílet jisté parametry. V případě rekurentního modulu je to ideální způsob jak zajistit efektivní sdílení paměťového prostoru napříč jednotlivými časovými kroky. Jedinou nevýhodou je, že nepodporuje sdílení parametrů, jejichž typ je tabulka, tato funkcionalita byla doplněna.

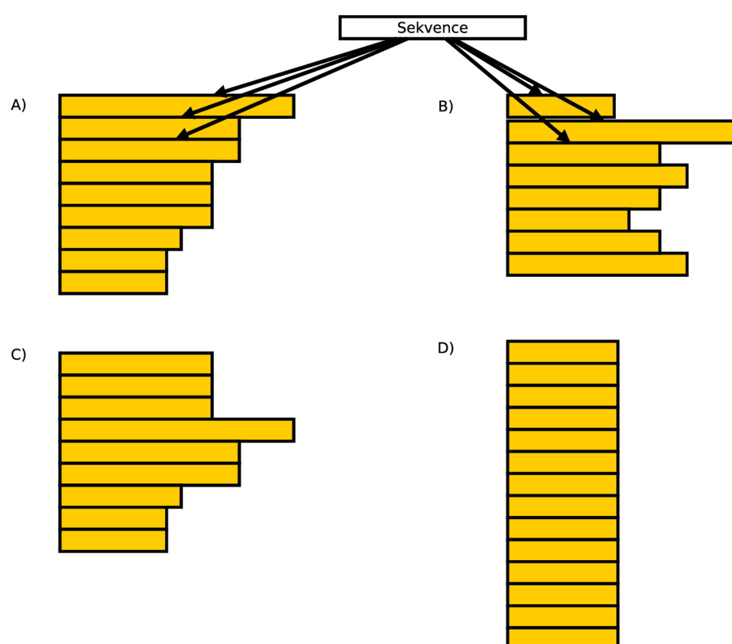
CTC

Pro CTC byla využita knihovna *warp-ctc* od společnosti *baidu*, která obsahuje api pro jazyk *lua* a realizuje výpočet na GPU paralelně pro více sekvencí. Jediné změny, které bylo nutno udělat, je transformovat výstupní sekvence do odpovídajícího formátu, s kterým počítá tato knihovna, a při zpětném průchodu transformovat chyby do původního datového formátu. Ač by měla být tato implementace CTC stabilní, dochází velmi často k získání hodnot *inf* nebo *nan*.

Paralelní zpracování sekvencí

Jak již bylo zmíněno v předešlých sekcích, trénování rekurentních architektur je možné urychlit zpracováním většího množství odlišných sekvencí v jedné dávce tak, že se jednotlivé kroky v historii počítají paralelně pro všechny sekvence. K maximalizaci tohoto efektu

dojde, pokud jsou v současné dávce zpracovávány sekvence o podobné délce. Tohoto lze docílit pomocí seřazení všech sekvencí na základě délky a následně je rozdělit do skupin o předem stanoveném počtu sekvencí. Experimentálně bylo zjištěno, že tento způsob trénování je sice nejrychlejší, ale dochází k výrazné ztrátě generalizace modelu. Extrémem na druhé straně je neudržování žádné preference mezi uspořádáním sekvencí a pravidelným promícháváním sekvencí před každou epochou. Tento přístup má vysoké generalizační schopnosti, ale trénování takovéhoto modelu je extrémně zdlouhavé. Kompromisem mezi těmito dvěmi variantami je seřazení sekvencí podle délky, rozdělení do skupin o stejném počtu sekvencí a při následném trénování vždy pouze promíchání pořadí zpracovávání skupin. Posledním podporovaným typem je umělé zkrácení sekvencí a vytvoření sekvencí nových z přebytků původních sekvencí, pokud zbytek sekvence nemůže vytvořit sekvenci o požadované délce, je doplněn daty z předešlé subsekvence. Tento typ nelze použít vždy, ale pokud je to možné, je trénování výrazně rychlejší než jakýkoliv z předešlých případů. Všechny tyto varianty jsou zobrazeny na obrázku 5.9.

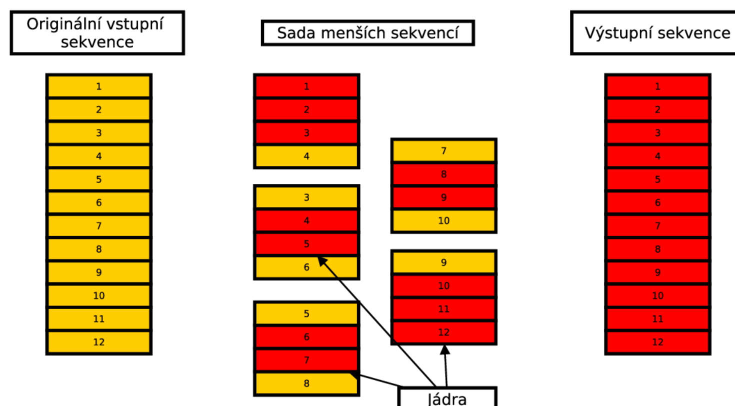


Obrázek 5.9: **Různé typy přístupu při zpracování více sekvencí najednou.** A) Seřazení sekvencí podle délky B) Náhodné zpracování sekvencí C) Vytvoření skupin obsahující sekvence o podobné délce D) Zarovnání všech sekvencí na stejnou délku a vytvoření nových sekvencí z přebytků

Dopředný průchod neuronové sítě

Při aplikaci natrénovaného modelu pro rozpoznávání může dojít u rekurentních modulů k situaci, že je na vstupu rozpoznávače sekvence delší než délka sekvencí, na kterých byl trénován. V tomto případě dojde k rozdělení sekvence na několik menších částí, každá část o délce maximálně podporované délky při trénování, jednotlivé menší časti sekvence se budou překrývat. Dojde ke zpracování menších sekvencí a celkový výsledek bude tvořen konkatenovanými jádry menších sekvencí. Jádra menších sekvencí jsou definována jako oblast v intervalu $< \frac{history}{4}, \frac{3history}{4} >$, kde proměnná *history* představuje maximální podporovanou

délku sekvencí. Tato procedura je vyobrazena na obrázku 5.10.



Obrázek 5.10: Dopředný průchod u testovacích dat pro sekvence delší než je podporovaná historie u rekurentních architektur, velikost historie je 4.

Datasety

Torch je využíván především pro zpracování obrazu a není vybaven prostředky pro práci s řečí, navíc jazyk *lua* obsahuje paměťový limit pro své datové struktury o velikosti 4gb. Z těchto důvodů byl pro uchování datové sady zvolen formát *hdf5*, který umožňuje uchovávat data v binární formátu, který je v současné době podporován skrze knihovnu *torch-hdf5*. K úpravě vstupních dat byl vytvořen skript v jazyce *python*, který umožňuje převod mezi formátem podporovaným toolkitem *kaldi*, *hdf5* a *netCDF*. Script taky zajištuje normalizaci vstupních dat, kdy je směrodatná odchylka a střední hodnota získaná z trénovacích dat následně aplikována na *held-out* a testovací data. Při převodu dat z jednoho formátu do druhého je provedeno seřazení všech sekvencí od největší po nejmenší. Script také umožňuje zarovnat všechny sekvence na požadovanou délku s tím, že větší sekvence budou rozděleny na několik menších sekvencí.

Vytvořený formát pro trénovací a *held-out* sadu obsahuje tyto položky s následujícím významem:

- rows - celkový počet záznamů
- cols - velikost jednoho vektoru rysů
- data - data konkatovaná do jednoho 2d pole
- labels - 1d pole obsahující třídy pro jednotlivé vektory rysů
- seq_sizes - obsahuje velikosti jednotlivých promluv
- lab_sizes - obsahuje velikosti jednotlivých vektorů tříd pro jednotlivé sekvence

Formát pro testovací data je mírně odlišný, protože je nutné rozlišit, která data jsou svázána s jednotlivými promluvami, aby mohla být následně dekódována. Každá promluva je proto reprezentována skupinou, jejíž název odpovídá názvu dané promluvy, každá skupina má následující atributy:

- data - 2d pole obsahující vektory rysů
- rows - obsahuje počet vektorů rysů
- cols - obsahuje velikost jednoho vektoru rysů

Trénovací algoritmy

Pro trénování neuronových sítí byla využita knihovna *optim*, u algoritmu *rmsprop* [29] došlo k rozšíření o podporu klasického (sekce 3.2.1) a Nesterova momenta (sekce 3.2.1).

Dále byla přidaná varianta navrhnuta v [10] a rozšířena o klasické a Nesterovo momentum. Tato varianta bude v následujícím textu označovaná jako *grmsprop*. U klasické implementace algoritmu *rmsprop* se udržuje hodnota průměrných magnitud vypočtených gradientů, nový gradient je touto hodnotou vždy normalizován. V důsledku by mělo dojít k utlumení gradientů, jejichž historie vykazuje velké změny, tedy pokud je povrch chybové funkce výrazně zakřiven a hodnota gradientu osciluje. U *grmsprop* se kromě průměrných magnitud udržuje také průměrná hodnota gradientu, která se odečítá před normalizací od průměrných magnitud. Tímto by měl být odstraněn problém, kdy je magnituda gradientu veliká v důsledku prudkého stoupání a nikoli oscilací.

Pro zabránění přetrénování byla implementována technika early stopping. Tato technika je charakteristická udržováním historie chyb na *held-out* sadě a udržováním parametrů, které vedly k nejnižší chybě na této sadě. V této práci má tento parametr nejčastěji hodnotu 3. Pokud se během časového úseku, který odpovídá velikosti historie, nepodaří získat parametry, které by vylepšily chybu na *held-out* sadě relativně o více než jedno procento, je trénování ukončeno. Výsledkem jsou poté váhy asociované s nejmenší chybou na *held-out* sadě.

K zajištění stability učení byla aplikována technika *gradient clipping*. Jedná se o nejjednodušší techniku pro zamezení fenoménu explodujícího gradientu, kdy dochází u rekurentních sítí při jisté konfiguraci vah pří zpětné propagaci chyby v čase k jejímu neúměrnému zvětšování. Tato technika omezí výsledný gradient do požadovaného rozsahu hodnot, konkrétně byl zvolen rozsah $< -1, 1 >$.

Testování implementovaných modulů

Pro testování rekurentních architektur a celkové funkcionality implementovaného řešení byla vytvořena sada testů. Pro testování správného dopředného a zpětného průchodu jednotlivých architektur byly předvypočítány správné hodnoty výstupu a chyby propagované do vstupní vrstvy při zpětném průchodu pro specifické hodnoty parametrů. S těmito hodnotami byly následně srovnány výstupy implementovaných modulů.

5.4 Ovládání

Vstupem programu je konfigurační soubor, jenž v současné době podporuje následující volby:

- learning_rate - nastavení koeficientu rychlosti učení pro trénovací algoritmus
- momentum - určuje velikost koeficientu momenta (sekce 3.2.1)

- batch_size - umožňuje nastavit velikost dávky
- history - umožňuje nastavit délku kontextu sekvencí, jež bude brána v potaz při trénování
- cuda - specifikuje, zda má být výpočet spuštěn na grafické kartě
- shuffle - specifikuje, zda mají být trénovací data promíchaná
- network - specifikuje soubor ve formátu *json*, který specifikuje model neuronové sítě
- max_epochs - specifikuje maximální počet trénovacích epoch
- validate_every - specifikuje, jak často má být klasifikační model testován na *held-out* sadě
- train_file - specifikuje vstupní data ve formátu *hdf5*
- val_file - specifikuje *held-out* data ve formátu *hdf5*
- optimizer - specifikuje, zda se má jako trénovací algoritmus použít klasický stochastic gradient descent, adadelta, rmsprop, g_rmsprop
- sampling_type - určuje způsob, jakým se získávají data pro jednotlivé trénovací dávky, povolené hodnoty tohoto pole jsou následující:
  - *frame* - volba je vhodná pro klasické dopředné neuronové sítě, dochází k selekci jednotlivých rámců z datových sad
  - *seq* - volba je vhodná pro rekurentní sítě, z datových sad se vybírají sekvence
  - *seq_frac* - volba vhodná pro rekurentní sítě, z datových sad se vybírají sekvence o podobné délce
- decay_every - určuje, jak často má probíhat zmenšování koeficientu rychlosti učení
- autosave_model - určuje, jestli se má po každé epoše ukládat natrénovaný model
- autosave_weights - určuje, zda-li se mají po každé epoše ukládat parametry modelu
- autosave_optimizer - určuje, zda-li ma docházet k ukládání stavu optimalizačního algoritmu po každé epoše
- autosave_prefix - definuje, jakým prefixem budou začínat veškeré automaticky ukládané struktury
- learning_rate_decay - definuje koeficient, který se bude používat pro zmenšování koeficientu rychlosti učení
- split_sequences - definuje, zda-li se mají sekvence, jež překračují maximální povolenou délku, rozdělit, povolené hodnouty jsou *true* nebo *false*

V současné době jsou podporovány následující typy vrstev neuronové sítě:

- lstm - LSTM vrstva (sekce 4.2)
- blst - obousměrná LSTM

- gru - GRU vrstva (sekce 4.3)
- bgru - obousměrná GRU
- rec_tanh, relu, sigmoid, prelu - klasická rekurentní vrstva s různou aktivační funkcí
- brec_tanh, relu, sigmoid, prelu - klasická obousměrná rekurentní vrstva s různou aktivační funkcí
- irec_tanh, relu, sigmoid, prelu - IRNN s různými aktivačními funkcemi (sekce 4.4)
- birec_tanh, relu, sigmoid, prelu - obousměrná IRNN s různými aktivačními funkcemi
- feedforward_tanh,relu,sigmoid,prelu - dopředná vrstva s různými aktivačními funkcemi
- linear - jedná se o lineární hladinu
- multiclass_classification - objektivní funkci CE (vzorec 3.12), ve spojení se softmax aktivační funkcí (vzorec 3.10), musí se jednat o poslední specifikovanou vrstvu
- ctc - představuje CTC objektivní funkci (sekce 4.6)

Architekura neuronové sítě je specifikována pomocí konfiguračního souboru ve formátu *json*, kde má každá vrstva definovaný svůj *typ*, *velikost*, zda-li používá *dropout* nebo *batch normalizaci*. Následuje ukázka definice jednoduché sítě:

```
{
    "layers": [
        {
            "size": 120,
            "type": "input"
        },
        {
            "size": 1024,
            "type": "feedforward_logistic",
            "batch_normalization": false,
            "dropout": 0.25
        },
        {
            "size": 1024,
            "type": "feedforward_logistic",
            "batch_normalization": false,
            "dropout": 0.5
        },
        {
            "size": 4016,
            "type": "linear"
        },
        {
            "size": 4016,
            "type": "multiclass_classification"
```

```
        }
    ]
}
```

5.5 Testování rychlosti a paměťové náročnosti

Implementované architektury byly srovnány s hlavní implementací rekurentních neuronových sítí v prostředí jazyka *lua*, kterou je knihovna *rnn*. Kritickými parametry pro srovnávání bylo množství spotřebované paměti a rychlost trénování.

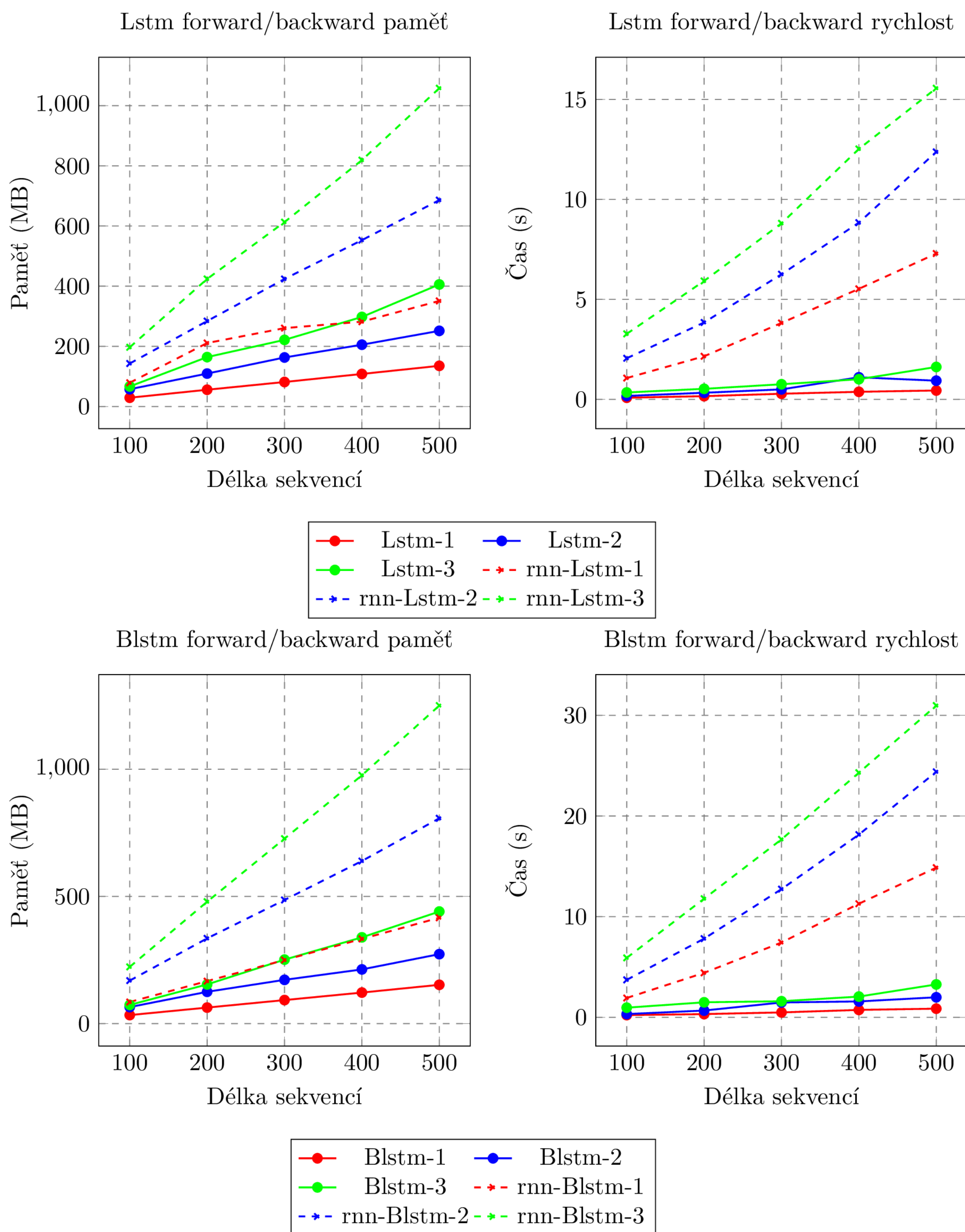
Srovnání se týkalo jak obousměrných variant tak jednosměrných variant LSTM a GRU s následujícími parametry:

- velikost výstupní vrstvy - 128
- velikost vstupní vrstvy - 128
- počet současně zpracovávaných sekvencí - 16

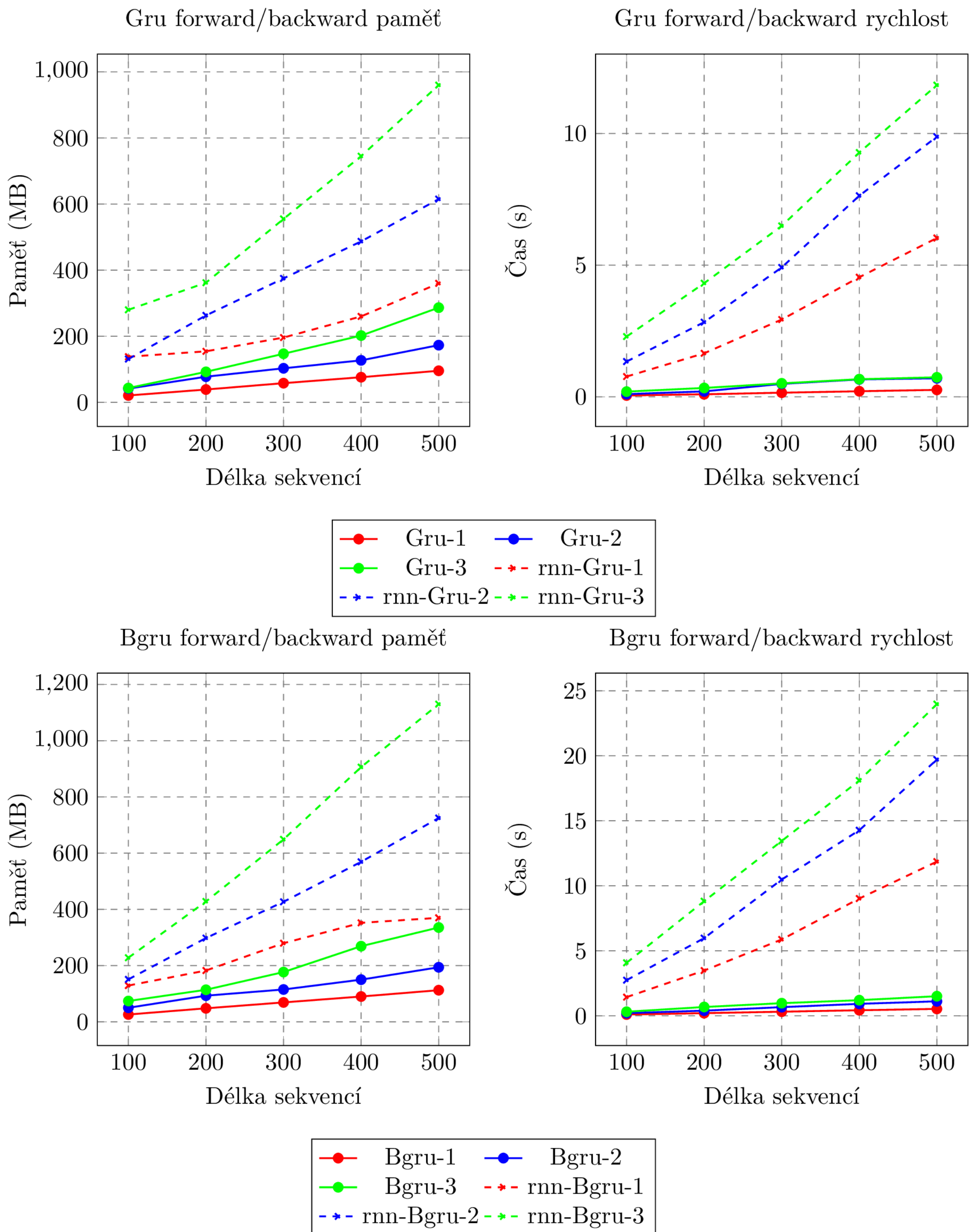
Ve srovnání figurují architektury s hloubkami 1 až 3 vrstvy a různé délky vstupních posloupností počínaje sekvencemi s délkou 100 až po délku 500. V rámci srovnání nebyly aplikovány paměťové optimalizace, které by se daly aplikovat na celou neuronovou síť, protože by stejně dobře mohly být použity i pro knihovnu *rnn*. Předmětem srovnání tedy byla hlavně efektivnost implementace rekurence. Srovnání bylo provedeno na grafické kartě *NVIDIA GeForce 940M*. Tabulka 5.1 shrnuje průměrně dosažené hodnoty pro jednotlivé architektury. Obrázky 5.11 a 5.12 vyobrazují spotřebu paměti a rychlost trénování pro architektury LSTM, obousměrná LSTM (BLSTM), GRU, oubousměrná GRU (BGRU).

Typ architektury	Průměrná redukce paměti	Průměrné zrychlení
LSTM	2.713×	11.5×
BLSTM	2.85×	14.22×
GRU	3.71×	10.93×
BGRU	3.56×	16.09×

Tabulka 5.1: Výsledná redukce paměťové náročnosti a zrychlení trénování pro rekurentní architekury typu LSTM, BLST, GRU a BGRU.



Obrázek 5.11: Grafy vyobrazují průběh paměťové náročnosti a rychlosti jednoho dopředného a zpětného průchodu u architektur LSTM a BLSTM pro různé délky sekvencí a různý počet vrstev. Čárkovaně jsou vyobrazeny hodnoty naměřené při použití knihovny *rnn*, plnou čárou potom implementované řešení.



Obrázek 5.12: Grafy vyobrazují průběh paměťové náročnosti a rychlosti jednoho dopředného a zpětného průchodu u architektur GRU a BGRU pro různé délky sekvencí a různý počet vrstev. Čárkovaně jsou vyobrazeny hodnoty naměřené při použití knihovny *rnn*, plnou čárou potom implementované řešení.

Kapitola 6

Datové sady

6.1 AMI

Představuje soubor 100 hodin nahrávaných zasedání týmu řešící fiktivní projekt. Schůzky se odehrávaly v angličtině, většina diskutujících nebyla rodilými mluvčími. Nahrávalo se ve třech různých místnostech, které měly různé akustické vlastnosti. Záznamy obsahují zvuk, jež byl zachycen jak blízko mluvčích tak i z větší vzdálenosti a také textový přepis, který obsahuje časování jednotlivých slov. Kromě zvuku obsahuje dataset také videozáznamy jednotlivých schůzek, a to jak záznamy snímající jednotlivé osoby tak záznam celé místnosti. Každý účastník měl k dispozici také elektronické pero, jehož záznam je též k dispozici.

Audio záznamy jsou převzorkovány ze 48kHz na 16kHz a jsou k dispozici ve WAV souborech, k dispozici je vždy 24 audio záznamů pro každou schůzku. Pro přepis řeči byl použit shlukovací algoritmus, který rozdělil data do úseků, ve kterých se vyskytuje ticho a ve kterých se vyskytuje řeč, tyto úseky pak byly manuálně přepsány.[21]

6.2 TIMIT

Je jednou z menších databází, obsahuje čtenou řeč od 630 různých čtenářů, mlůvícími 8 hlavními dialekty americké angličtiny. Na rozdíl od jiných databází tohoto charakteru obsahuje také přesný fonetický přepis. Každý čtenář čte 10 foneticky bohatých vět. Celková délka je 5.4 hodiny. [5]

Kapitola 7

Experimenty

V následující kapitole budou popsány provedené experimenty, experimenty byly prováděny s pomocí toolkitu kaldi [26], jejich cílem bylo vyhodnotit vhodnost jednotlivých typů neuronových sítí pro modelování akustické informace při rozpoznávání řeči. Celkově můžeme experimenty rozdělit do tří rozdílných případů:

- CD-DNN-HMM - spojení klasické dopředné neuronové sítě a HMM, kdy HMM slouží pro modelování časových závislostí a neuronová síť slouží k výpočtu věrohodností jednotlivým stavům HMM modelu, pro trénování je použita jako objektivní funkce cross entropie (vzorec 3.12).
- CD-RNN-HMM - podobné jako typ CD-DNN-HMM, ale pro předpověď jednotlivých emisních věrohodností se využívá rekurentní neuronová síť.
- RNN-CTC - tento typ modelu nevyužívá HMM, je použita objektivní funkce CTC (sekce 4.6), která nevyžaduje explicitní zarovnání jednotlivých akustických jenotek, v této práci se uvažují pouze jednoduché fonémy.

Experimenty obsahující HMM byly vykonány na datové sadě AMI (sekce 6.1). Experimenty s modelem RNN-CTC pak byly provedeny na datové sadě TIMIT (sekce 6.2). Všechny experimenty byly prováděny na grafických kartách ve výpočetním gridu Metacentrum [33].

7.1 HMM

Pro vytvoření počátečního zarovnání jednotlivých fonémů byl použit kaldi toolkit a jeho standardní recept pro databázi AMI. HMM model obsahoval 4016 stavů. Velikost poslední výstupní vrstvy je tedy 4016 neuronů.

Příznaky tvořilo 40 mel-filter banků, delta a delta-delta koeficienty, byly vytvořeny pomocí toolkitu kaldi, celková velikost jednoho rámce je tedy 120. Jediný druh předzpracování představovala normalizace, kdy došlo k získání směrodatné odchylky a středních hodnot u trénovacích dat, těmito hodnotami následně byla upravena data ve všech ostatních datových sadách.

Experimenty se zaměřovaly především na nalezení optimální trénovací strategie a porovnání jednotlivých implementovaných rekurentních sítí při modelování akustické informace.

Pro dekódování výsledného výstupu bylo použito kaldi.

10% z trénovacích dat bylo odloženo a sloužilo jako *held-out* sada, v následujícím textu bude tato datová sada označovaná jako *train_cv10*. Trénovací sada bude v následujících experimentech označovaná jako *train_tr90*. Pro experimenty, které sloužily pro získání přehledu o vhodnosti jednotlivých technik pro učení neuronové sítě, sloužila redukovaná trénovací sada, která bude následně označovaná jako *train_tr15*, tvořilo ji $\approx$ 25% z trénovacíh dat. Evaluace úspěšnosti jednotlivých modelů probíhala pomocí metriky WER (sekce 2.4) a to na datových sadách *dev* a *eval*, které se běžně používají v toolkitu kaldi pro evaluaci úspěšnosti. U experimentů s rekurentními sítěmi bylo v některých případech potřeba odstranit z trénovacíh dat sekvence, jejichž délka přesahuje 1000 rámců, aby mohlo dojít k rychlejšímu trénování, kdyby k tomu nedošlo, muselo by být výrazně redukováno množství paralelně zpracovávaných sekvencí, kvůli nedostatečné paměti na GPU.

Pokud se jednalo o trénování s momentem, tak byla jeho hodnota ve všech případech nastavena na 0.9 a učící rychlost byla při těchto experimentech snížena $10\times$.

7.1.1 CD-DNN-HMM

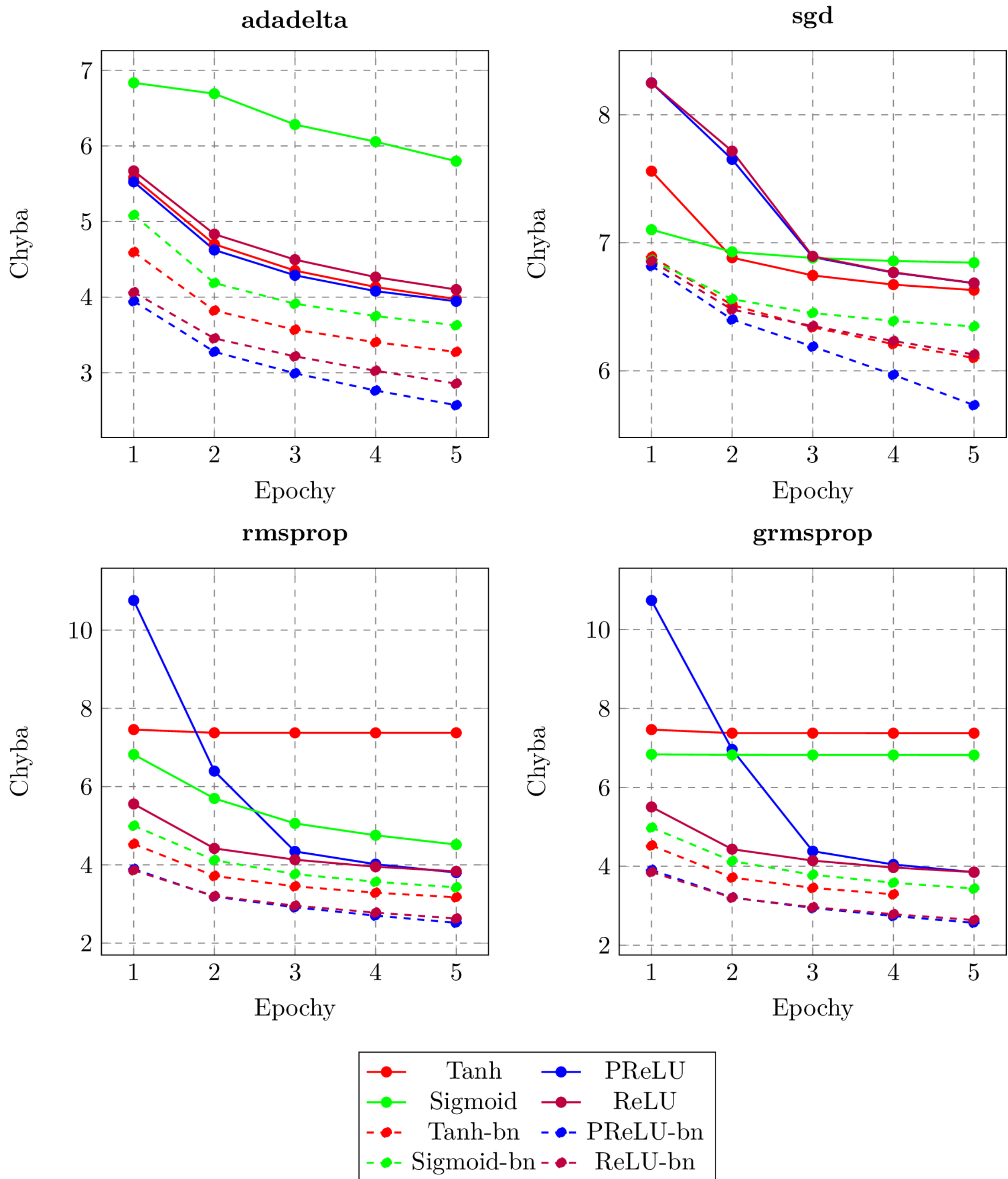
Počáteční experimenty se zaměřily na vybrání optimální aktivační funkce a trénovacího algoritmu ve spojení s batch normalizací. Motivací bylo nalézt takovou kombinaci, která by se co nejlépe adaptovala na trénovací data. Tyto experimenty byly prováděny na datové sadě *train_tr15* s dopřednou neuronovou sítí, jež měla 1024 neuronů v každé vrstvě a celkový počet vrstev byl 6. Maximální počet epoch byl 5, kdy epocha představuje jeden průchod napříč všemi trénovacími daty. Velikost jedné dávky byla 1024 rámců. Pořadí zpracování jednotlivých rámců bylo před každdou epochou náhodně pozměněno. Rychlost učení byla v těchto experimentech stanovena na 0.001, pokud se nejednalo o experimenty s momentem, kde byla rychlost učení 0.0001. V prvních experimentech došlo k otestování následujících trénovacích algoritmů:

- adadelta,
- stochastic gradient descent (sgd),
- rmsprop,
- grmsprop,

spolu s následujícími aktivačními funkcemi:

- Tanh (rovnice 3.4),
- Logistická sigmoida (rovnice 3.2),
- ReLU (rovnice 3.6),
- PReLU (rovnice 3.9), kdy byl parametr a sdílen celou neuronovou vrstvou a byl inicializován na hodnotu 0.25

a technikou batch normalizace. Průběhy jednotlivých experimentů jsou vyobrazeny na obrázku 7.1 a v tabulce 7.1 jsou konkrétní hodnoty pěti nejmenších dosažených chyb. Vliv batch normalizace je patrný u všech experimentů, dochází k výraznému zlepšení adaptace. Z trénovacích algoritmů dosáhl nejlepšího výsledku algoritmus rmsprop, naopak nejhoršího výsledku dosáhl algoritmus sgd. Algoritmy rmsprop a grmsprop dosahovaly podobné



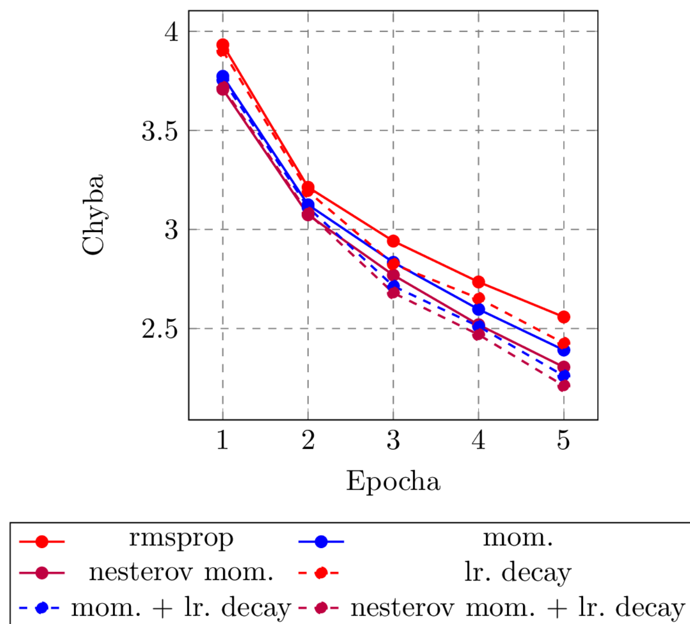
Obrázek 7.1: Jednotlivé grafy obsahují průběhy chyb na trénovací sadě během experimentů s aktivačními funkcemi: PReLU, ReLU, Logistická sigmoida, Tanh a trénovacími algoritmy rmsprop, grmsprop, sgd a adadelta ve spojení s technikou batch normalizace u architektury DNN. Evidentní je vliv zejména batch normalizace (čárkovaně), která vždy vede k lepší adaptaci na trénovací data. Z trénovacích algoritmů dosahuje nejlepšího výsledku algoritmus rmsprop. Z aktivačních funkcí potom PReLU.

úspěšnosti a není mezi nimi výrazný rozdíl. Z aktivačních funkcí dosahuje stabilně ve všech případech nejmenší chyby *PReLU*, následovaná aktivační funkcí *ReLU*.

Pořadí	Typ algoritmu	Dosažená chyba
1.	PReLU + bn + rmsprop	2.5205
2.	PReLU + bn + grmsprop	2.5677
3.	PReLU + bn + adadelta	2.571
4.	ReLU + bn + rmsprop	2.6305
5.	ReLU + bn + grmsprop	2.6371

Tabulka 7.1: 5 nejmenších dosažených chyb během experimentování s různymi typy aktivačních funkcí, trénovacími algoritmy a batch normalizací u architektury DNN.

V dalších experimentech bude využíváno aktivační funkce *PReLU* spolu s batch normalizací a učícím algoritmem rmsprop. Následující experimenty se zaměřují na využívání momenta (sekce 3.2.1) a dynamicky se měnící rychlosti učení. Konkrétně se jednalo o strategii, ve které dochází každé dvě epochy k vynásobení rychlosti učení konstantou, která měla hodnotu 0.5. Vývoj chyby během trénování je vyobrazen na obrázku 7.2. Z výsledků experimentů je patrný pozitivní vliv snižování rychlosti učení. Nejlepších výsledků dosáhla technika využívající Nesterova momenta a dynamického snižování učící rychlosti. Konkrétně byla dosažena po pěti epochách s hodnotou 2.211. Dynamické snižování rychlosti bylo prospěšné ve všech případech.



Obrázek 7.2: Na grafu jsou vyobrazeny průběhy chyb pro kombinace algoritmu rmsprop s Nesterovým momentem, klasickým momentem a dynamickým snižováním rychlosti učení. Dynamické snižování rychlosti učení má pozitivní vliv ve všech případech. Nejmenší chyby je potom dosaženo s Nesterovým momentem.

Následující experimenty, které se zaměřily na nejvhodnější topologii neuronové sítě, využívaly pro trénování Nesterovo momentum a dynamicky se měnící rychlost učení. V

experimentech figurovalo celkově 5 různých architektur s rozličným počtem parametrů. Tabulka 7.2 zobrazuje dosažené chyby po 5 epochách. Nejlepší architekturu tvoří 4 vrstvy, každá s počtem 2048 neuronů.

Pořadí	Dosažená chyba	Počet vrstev	Velikost vrstvy	Počet parametrů
1.	1.4369	4	2048	21 M
2.	1.5802	2	4096	33.7 M
3.	2.0010	2	2048	12.6 M
4.	2.2047	6	1024	9.5 M
5.	2.6823	9	1024	12.6 M

Tabulka 7.2: Nejmenší chyby dosažené v rámci experimentů s rozličnými topologiemi DNN.

Pro závěrečný experiment byla natrénováná DNN s následujícími parametry:

- počet neuronů v jedné vrstvě - 2048
- počet vrstev - 4
- rychlost učení - 0.0001
- Nesterovo momentum - 0.9
- aktivační funkce - $PReLU$
- parametr řídící dropout ve všech vrstvách - 0.1
- velikost dávky - 1024 rámců, před každou epochou byly tyto rámce náhodně promíchány
- na každou vrstvu byla aplikovaná technika batch normalizace

Dosažený WER na *eval* sadě byl 45.9%, na *dev* sadě potom 41.0%.

7.1.2 CD-RNN-HMM

Počáteční experimenty se zaměřily na různé druhy aktivačních funkcí v klasické rekurentní architektuře, různé druhy optimalizačních algoritmů a batch normalizaci. Síť tvořily 4 vrstvy, každá o 1024 neuronech. V jedné dávce se nacházelo 32 sekvencí. Trénování probíhalo na celých sekvencích, které byly seřazeny a rozděleny do skupin, před každou epochou bylo náhodně pozměněno pořadí jednotlivých zpracovávaných skupin (sekce 5.3.3). Experimenty byly prováděny na datové sadě $train_tr15$, ze které byly vyřazeny sekvence, které měly více než 1000 rámců, což představovalo 275 sekvencí o celkové počtu 334371, jedná se o $\approx 8\%$ dat z celkové množství dat. Všechny experimenty, které nepracovaly s momentem, měly parametr rychlosti učení nastavený na hodnotu 0.0001 a v kombinaci s momentem potom na 0.00001. Průběh chyb pro jednotlivé experimenty je zobrazen na obrázku 7.3. V tabulce 7.3 se nachází pět nejlepší výsledků. Z výsledků je opět patrný význam batch normalizace a za nejlepší aktivační funkci lze opět považovat $PReLU$, dosáhla v kombinaci s algoritmem rmsprop a technikou batch normalizace nejmenší chyby, nicméně její postavení není tak dominantní jako v případě DNN. Z trénovacích algoritmů opět dosáhl nejlepšího

Pořadí	Typ algoritmu	Dosažená chyba
1.	PReLU + bn + rmsprop	2.9899
2.	ReLU + bn + grmsprop	2.9918
3.	PReLU + bn + grmsprop	3.007
4.	ReLU + bn + rmsprop	3.192
5.	Tanh + bn + rmsprop	3.3873

Tabulka 7.3: 5 nejmenších dosažených chyb v rámci experimentů s architekturou RNN s různymi typy trénovacích algoritmů, aktivačních funkcí a technikou batch normalizace.

výsledku již zmíněný rmsprop. Varianta grmsprop ani v tomto případě nedosáhla lepších výsledků, aplikované modifikace se tedy jeví jako bezpředmětné.

Následující experimenty (obrázek 7.4) se zaměřily na výběr optimální trénovací strategie podobně jako v případě dopředných neuronových sítí. Nejlepšího výsledku opět dosáhl algoritmus rmsprop v kombinaci s Nesterovým momentem a dynamicky snižovanou velikostí rychlosti učení, nicméně rozdíly mezi jednotlivými variantami jsou minimální.

V následujícím srovnání (obrázek 7.5) figurují jednosměrné a obousměrné varianty rekurentních neuronových sítí pro následující typy architektur: IRNN s aktivační funkcí $PReLU$ a *ReLU* (sekce 4.4), LSTM (sekce 4.2), Gru (sekce 4.3) a klasická rekurentní síť s aktivační funkcí $PReLU$. Z výsledku je patrný pozitivní vliv obousměrných rekurentních neuronových sítí ve všech případech. Rozdíly v adaptaci mezi klasickou rekurentní neuronovou sítí a její alternativou, jejíž parametry jsou na počátku inicializované na jednotkovou matici, nejsou patrné.

Další experimenty probíhaly na datových sadách *train_tr90* a *train_cv10*. V případě zpracování celých sekvencí byly z datové sady odstraněny sekvence, jejichž délka přesahovala 1000 rámců. Celkově byla velikost trénovací sady v tomto případě snížena o $\approx 9\%$. Trénování neuronových sítí bylo omezeno nejvýše na 10 epoch. Pro trénování byl vždy použit algoritmus rmsprop v kombinaci s Nesterovým momentem a dynamickým snižováním rychlosti učení, s počáteční rychlosti učení 0.00001, vždy byla také použita technika batch normalizace.

Další experiment se zaměřil na porovnání dvou rozdílných typů zpracování vstupních sekvencí. V prvním případě došlo k rozdělení sekvencí na podsekvence o délce 64 rámců a v jedné dávce jich bylo zpracováno celkově 64. Pořadí zpracovávaných sekvencí bylo vždy náhodně pozměněno před každou epochou. V druhém případě pak došlo k seřazení všech sekvencí na základě velikosti a rozdělení do skupin o podobné délce. Pořadí zpracovávání těchto skupin bylo před každou epochou náhodně změněno. Tento experiment byl proveden na architektuře BLSTM. Jednotlivé parametry modelu byly následující:

- počet vrstev - 2,
- počet neuronů v každé vrstvě - 1024,
- hodnota parametru řídícího dropout - 0.1 pro první vrstvu a 0.2 pro druhou vrstvu.

Výsledky jsou prezentovány v tabulce 7.4. Zpracovávání celých sekvencí se jeví jako optimálnější strategie. Nicméně jedna epocha při zpracování celých sekvencí trvá přibližně $2\times$ tolik času a model zaměřený na zpracování celých sekvencí zabírá v paměti GPU daleko více prostoru, což výrazně omezuje počet maximálních parametrů modelu.

Typ zpracování sekvencí	Délka jedné epochy	Paměťová náročnost	eval WER	dev WER
Zpracování celých sekvencí	7h	$\approx 3.5G$	31.3%	30.4 %
Rozdělení na podsekvence	3.5h	$\approx 1.5G$	34.0%	31.2 %

Tabulka 7.4: Dosažené výsledky pro architekturu BLSTM u experimentů s různým typem zpracování vstupní posloupnosti.

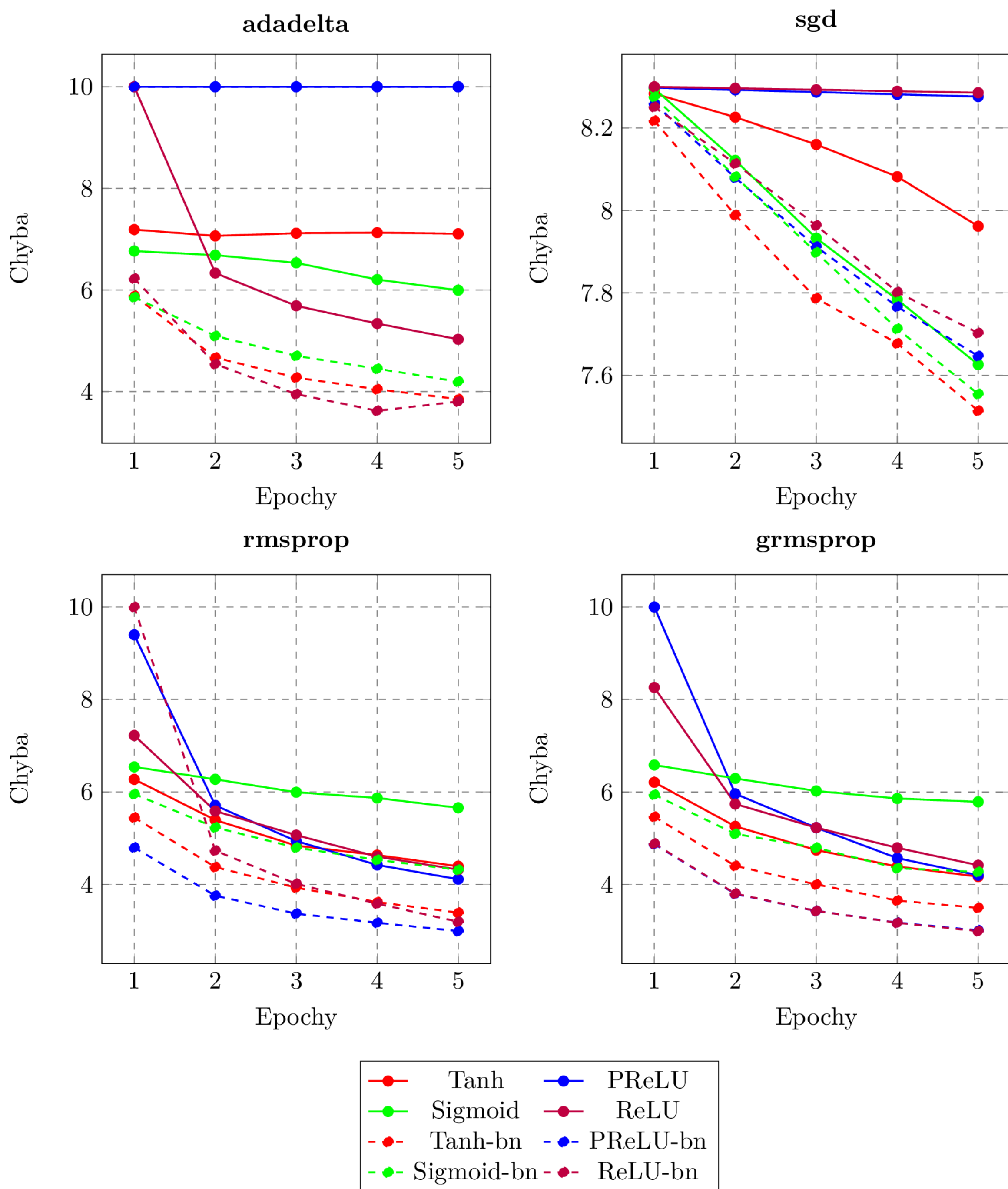
Další experiment se zaměřuje na srovnání rozdílných typů obousměrných rekurentních sítí, konrétně pak GRU (sekce 4.3), LSTM (sekce 4.2), IRNN (sekce 4.4) s aktivační funkci ReLU nebo PReLU, parametry jednotlivých modelů jsou uvedeny v tabulce 7.5. Dosažené výsledky pak v tabulce 7.6. Dominantní postavení zastává architektura BLSTM. Následovaná překvapivě IRNN. Aktivační funkce PReLU v tomto případě dosahuje mírně lepších výsledku. Na posledním místě se umístila architektura BGRU. Nicméně je nutno podotknou, že obsahovala nejmenší počet parametrů a taky došlo ke zvolení moc vysoké hodnoty dropoutu ve všech případech, protože žádná z prezentovaných architektur nedokončila svůj trénink během 10 epoch. Vždy došlo k ukončení trénování z důvodu dosažení maximálního počtu epoch.

Typ architektury	Počet vrstev	Velikost vrstvy	Počet parametrů	Dropout
BLSTM	2	1024	13 M	0.1 a 0.2
BGRU	2	1024	10.7 M	0.1 a 0.2
IRNN + PReLU	6	1024	12.6 M	0.1
IRNN + ReLU	6	1024	12.6 M	0.1

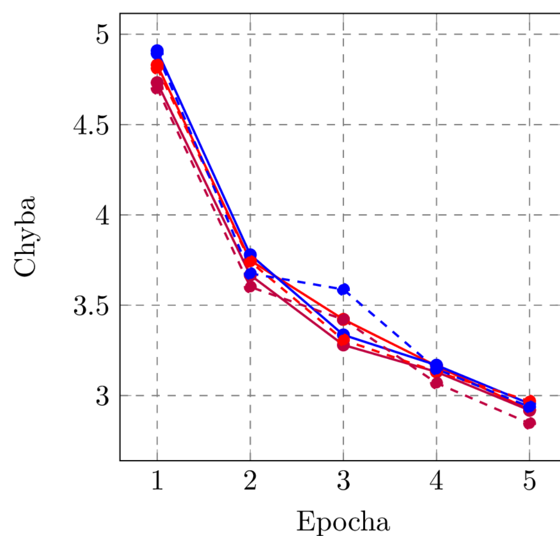
Tabulka 7.5: Parametry RNN modelů při závěrečném srovnání na datové sadě AMI.

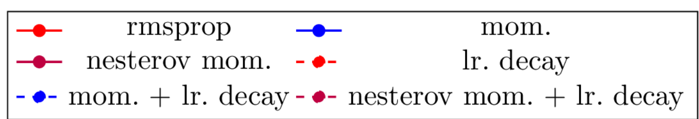
Typ architektury	eval WER	dev WER
BLSTM	31.3%	30.4%
BGRU	35.1%	32.6%
IRNN + PReLU	34.1%	32.1%
IRNN + ReLU	34.2%	32.3%

Tabulka 7.6: Výsledky závěrečného experimentu na korpusu AMI.

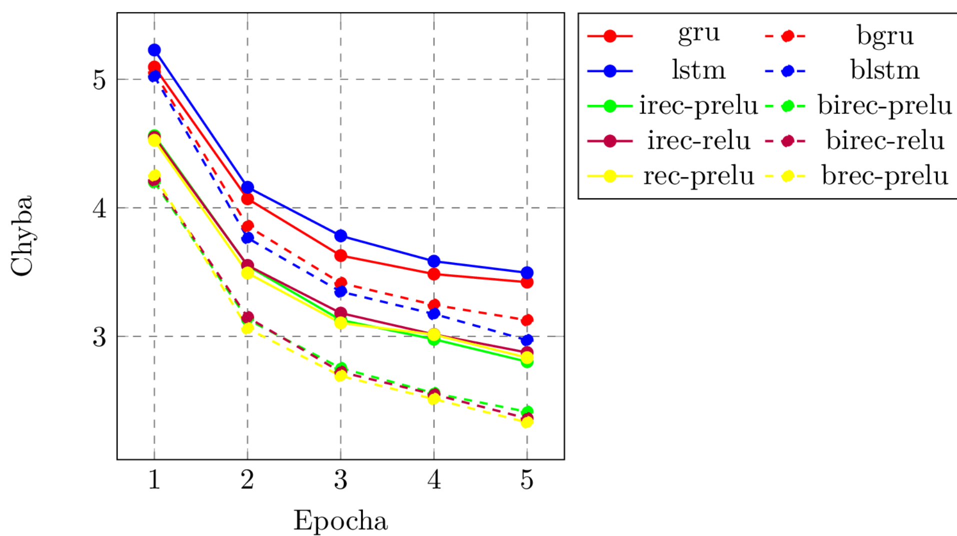


Obrázek 7.3: Na grafech jsou vyobrazeny průběhy chyb u experimentů s RNN pro různé aktivační funkce a trénovací algoritmy ve spojení s technikou batch normalizace. Patrný je vliv techniky batch normalizace, jež příspívá k lepší adaptaci modelu ve všech případech.





Obrázek 7.4: Graf zobrazující průběh chyby během experimentování s různými kombinacemi algoritmu rmsprop s momentem a dynamickou rychlostí učení.



Obrázek 7.5: Grafy vyobrazující průběh chyby při trénování u jednosměrných a obousměrných variant (čárkovaně) rekurentních sítí pro následující typy LSTM(lstm), GRU(gru), IRNN(irnn) a RNN(rnn) s aktivačními funkcemi PReLU(prelu) a ReLU(relu). Obousměrné varianty rekurentních sítí si vedou daleko lépe.

7.2 CTC

Experimenty s objektivní funkcí CTC (sekce 4.6) byly provedeny na databázi TIMIT (sekce 6.2). Trénování probíhalo na sadě 48 fonému a při skórování došlo k jejich přemapování na 39 fonému, ze skórování i trénování byl vyřazen foném reprezentující ticho. Pro skórování byla použita metrika PER (2.4). Výstupní vrstva měla celkově 49 fonému, protože jedna výstupní třída byla přidána pro reprezentaci symbolu *blank*. Pro dekódování byl použit algoritmus *best path decoding* (sekce 4.6). Trénování probíhalo na základě výsledků získaných v sekci 7.1.2. Byl použit algoritmus rmsprop s Nesterovým momentem a dynamickým snižování rychlosti učení. Jediný rozdíl představovala počáteční rychlost učení, která byla v tomto případě zvolena na 0.0001 a také nebyla použita technika dropout, protože nebylo pozorováno přetrénování při jednoduchých experimentech s datovou sadou. Příznaky tvořilo 40 mel-filter banků, delta a delta-delta koeficienty, na které byly uplatněny následující operace fMLLR, CMVN a byly normalizovány (sekce 2.1).

Cílem experimentů bylo zhodnotit úspěšnost jednotlivých typů RNN při využívání CTC. Klasické rekurentní neuronové sítě s aktivační funkcí PReLU a ReLU nebylo možné použít, protože při jejich použití docházelo ke generování invalidních hodnot knihovnou *warp-ctc*, v případě použití hyperbolického tangensu nebo logistické sigmoidy docházelo k výrazně pomalejší konvergenci algoritmu. Z těchto důvodu se srovnání účastní pouze architektura BLSTM a BGRU, obě architektury měli dvě vrstvy o velikosti 512 neuronů. Trénovány byly po dobu 10 epoch. Výsledky jsou prezentovány v tabulce 7.7. Lepšího výsledku dosáhla architektura BLSTM. Dosažené výsledky jsou o něco lepší než v originálním članku o CTC [11], kde bylo dosaženo s algoritmem *best path decoding* chyby 31.47% PER.

Typ architektury	Počet parametrů	eval PER	dev PER
BLSTM	2.3 M	30.85%	28.76%
BGRU	1.7 M	31.74%	29.64%

Tabulka 7.7: Výsledky závěrečného experimentu na datové sadě TIMIT s CTC objektivní funkcí.

7.3 Vyhodnocení

Při experimentování s aktivační funkcí PReLU došlo k potvrzení jejího dominantního postavení na poli klasických dopředných neuronových sítí, v případě rekurentních neuronových sítí dosáhla pouze malého zlepšení oproti klasické ReLU.

Při experimentech s různými trénovacími algoritmy bylo zjištěno, že dominantní postavení zastává algoritmus rmsprop v obou typech architektur, přičemž aplikace momenta vede vždy k rychlejší konvergenci, jeho efekt není tak výrazný u rekurentních architektur. Při aplikování momenta je nutné vždy výrazně snížit rychlost učení. V případě klasické neuronové sítě je pak vidět výraznější zlepšení při použití Nesterova momenta. Modifikace algoritmu rmsprop, označovaná jako grmsprop, nebyla úspěšnější při trénování.

Dynamické snižování rychlosti učení vede k rychlejší konvergenci u obou typů architektur. Kombinací dynamického snižování rychlosti učení a momenta vede k ještě lepším výsledkům.

Experimentálně byla ověřena účinnost obousměrných sítí, kdy došlo k výraznému zlep-

šení při adaptaci modelu na trénovací data, i když došlo ke snížení celkového počtu parametrů.

Experimentálně byla ověřena účinnost techniky batch normalizace, kdy došlo k výraznému zrychlení adaptace modelu na trénovací data ve všech případech. Tato technika byla úspěšně integrována do všech implementovaných rekurentních modulů. Jedním z průvodních jevů zlepšené adaptace modelu na data je přetrénování. Tento jev je výraznější u dopředných neuronových sítí. Tento fenomén byl vyřešen aplikací regularizační techniky dropout.

V případě paralelního zpracování sekvencí u rekurentních sítí byly srovnány dva módy, kdy se jako nejlepší přístup z hlediska rychlosti trénování a úspěšné konvergence jeví mód, ve kterém jsou seskupeny sekvence o podobné délce a následně jsou během trénování tyto skupiny náhodně promíchávány. V případě modelu CD-RNN-HMM, kdy jsou vytvořena počáteční zarovnání, není nutné zpracovávat celou sekvenci najednou a je možné dosáhnout ještě rychlejšího trénování, pokud dojde k umělému rozdělení sekvencí na menší části, které jsou zpracovávány paralelně. Nicméně v případě zpracování řeči tato technika dosahuje horších výsledku a je naopak vhodnější prezentovat co možná nejdelší kontext, pokud ovšem nejsme limitování pamětí, kdy může být vhodnější použít neuronovou síť s větším počtem parametrů a zpracovávat menší sekvence.

Závěrečnému srovnání neuronových sítí dominuje architektura LSTM, která dosáhla nejmenšího WER, jak na datové sadě AMI, tak i na datové sadě TIMIT v případě využití CTC. V porovnáni s dopřednou neuronovou sítí na datové sadě AMI má průměrné zlepšení hodnotu 12.6% WER.

Kapitola 8

Závěr

Výstupem této práce je efektivní a modulární implementace tří typů rekurentních neuronových sítí v prostředí jazyka *lua* s pomocí knihoven *torch* a *nn*, která umožňuje provádět experimenty i nad rozsáhlejšími datovými sadami v rozumném čase, a to hlavně díky efektivnějšímu využívání paměti a větší paralelizaci výpočtu. *Torch* a knihovna *nn* jsou předmětem aktivního vývoje, kdy jsou nově publikované techniky implementovány ve velice krátkem čase. Jedním z cílu této práce bylo umožnit otestování těchto nových technik při rozpoznávání řeči, čehož bylo dosáhnuto vytvořením převodního skriptu, který je schopen konvertovat data mezi datovým formátem podporovaným toolkitem kaldi a formátem *hdf5*, jenž má podporu napříč velkým spektrem programovacích jazyků. Tímto způsobem může dojít k vytvoření akustického modelu v prostředí, které se specificky orientuje na strojové učení. Následně na takovýto model můžou být aplikovány pokročilejší techniky z oblasti zpracování řeči, které jsou dostupné v toolkitu kaldi.

Ze srovnání neuronových sítí při rozponávání řeči se jeví jako nejlepší model BLSTM. Trénovacím technikám pak dominuje algoritmus rmsprop s Nesterovým momentem a dynamicky se snižující hodnotou rychlosti učení.

Další pokračování této práce by se mohlo zaměřit na pokročilejší experimenty s CTC a architekturou LSTM a jejich vyhodnocením na větší datové sadě než je TIMIT. Bylo by také nutné prozkoumat, co stojí za nestabilitou aktuální knihovny použitou pro CTC.

Úplným závěrem bych si dovolil poznamenat, že experimenty s rekurentními neuronovými sítěmi vyžadují nemalé výpočetní nároky. Typické trvání experimentu od extrakce příznaku až po dékódování výsledku na datové sadě AMI bylo v průměru týden, přičemž zhruba 60 % tohoto času tvořilo trénování modelu neuronové sítě na GPU. Bez možnosti využívání výpočetního centra Metacentrum by nemohl žádný ze zde publikovaných experimentů proběhnout. Tento trend bude z hlediska vývoje strojového učení, kdy dochází k vytváření stále větších modelů a zpracovávání stále většího množství dat, pokračovat.

Literatura

[1] Abadi, M.; Agarwal, A.; Barham, P.; aj.: TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. 2015, software available from tensorflow.org.
URL `http://tensorflow.org/`

[2] Bergstra, J.; Breuleux, O.; Bastien, F.; aj.: Theano: a CPU and GPU Math Expression Compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, Červen 2010, oral Presentation.

[3] Cho, K.; van Merrienboer, B.; Bahdanau, D.; aj.: On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. *CoRR*, ročník abs/1409.1259, 2014.
URL `http://arxiv.org/abs/1409.1259`

[4] Collobert, R.; Bengio, S.; Marithoz, J.: Torch: A Modular Machine Learning Software Library. 2002.

[5] Garofolo, J. S.; Lamel, L. F.; Fisher, W. M.; aj.: DARPA TIMIT Acoustic Phonetic Continuous Speech Corpus CDROM. 1993.

[6] Gers, F.; Schmidhuber, J.; Cummins, F.: Learning to forget: continual prediction with LSTM. In *Artificial Neural Networks, 1999. ICANN 99. Ninth International Conference on (Conf. Publ. No. 470)*, ročník 2, 1999, ISSN 0537-9989, s. 850–855 vol.2, doi:10.1049/cp:19991218.

[7] Gers, F. A.; Schraudolph, N. N.; Schmidhuber, J.: Learning Precise Timing with LSTM Recurrent Networks. ročník 3, 2002: s. 115–143.

[8] Glorot, X.; Bengio, Y.: Understanding the difficulty of training deep feedforward neural networks. In *JMLR W&CP: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2010)*, ročník 9, Květen 2010, s. 249–256.

[9] Graves, A.: *Supervised sequence labelling with recurrent neural networks*. Studies in Computational intelligence, Heidelberg, New York: Springer, 2012, ISBN 978-3-642-24796-5.
URL `http://opac.inria.fr/record=b1133792`

[10] Graves, A.: Generating Sequences With Recurrent Neural Networks. *CoRR*, ročník abs/1308.0850, 2013.
URL `http://arxiv.org/abs/1308.0850`

[11] Graves, A.; Fernández, S.; Gomez, F.; aj.: Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, New York, NY, USA: ACM, 2006, ISBN 1-59593-383-2, s. 369–376, doi:10.1145/1143844.1143891.
URL http://doi.acm.org/10.1145/1143844.1143891

[12] Hannun, A. Y.; Case, C.; Casper, J.; aj.: Deep Speech: Scaling up end-to-end speech recognition. *CoRR*, ročník abs/1412.5567, 2014.
URL http://arxiv.org/abs/1412.5567

[13] He, K.; Zhang, X.; Ren, S.; aj.: Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *CoRR*, ročník abs/1502.01852, 2015.
URL http://arxiv.org/abs/1502.01852

[14] He, K.; Zhang, X.; Ren, S.; aj.: Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *CoRR*, ročník abs/1502.01852, 2015.
URL http://arxiv.org/abs/1502.01852

[15] Hinton, G. E.; Osindero, S.; Teh, Y.-W.: A Fast Learning Algorithm for Deep Belief Nets. *Neural Comput.*, ročník 18, č. 7, Červenec 2006: s. 1527–1554, ISSN 0899-7667, doi:10.1162/neco.2006.18.7.1527.
URL http://dx.doi.org/10.1162/neco.2006.18.7.1527

[16] Hochreiter, S.; Schmidhuber, J.: Long Short-Term Memory. *Neural Comput.*, ročník 9, č. 8, Listopad 1997: s. 1735–1780, ISSN 0899-7667, doi:10.1162/neco.1997.9.8.1735.
URL http://dx.doi.org/10.1162/neco.1997.9.8.1735

[17] Huang, X.; Acero, A.; Hon, H.-W.: *Spoken Language Processing: A Guide to Theory, Algorithm, and System Development*. Upper Saddle River, NJ, USA: Prentice Hall PTR, první vydání, 2001, ISBN 0130226165.

[18] Ioffe, S.; Szegedy, C.: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR*, ročník abs/1502.03167, 2015.
URL http://arxiv.org/abs/1502.03167

[19] Le, Q. V.; Jaitly, N.; Hinton, G. E.: A Simple Way to Initialize Recurrent Networks of Rectified Linear Units. *CoRR*, ročník abs/1504.00941, 2015.
URL http://arxiv.org/abs/1504.00941

[20] LeCun, Y.; Bottou, L.; Orr, G. B.; aj.: Effiicient BackProp. In *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*, London, UK, UK: Springer-Verlag, 1998, ISBN 3-540-65311-2, s. 9–50.
URL http://dl.acm.org/citation.cfm?id=645754.668382

[21] Mccowan, I.; Lathoud, G.; Lincoln, M.; aj.: The AMI Meeting Corpus. In *In: Proceedings Measuring Behavior 2005, 5th International Conference on Methods and Techniques in Behavioral Research. L.P.J.J. Noldus, F. Grieco, L.W.S. Loijens and P.H. Zimmerman (Eds.), Wageningen: Noldus Information Technology*, 2005.

[22] Miao, Y.; Gowayyed, M.; Metze, F.: EESEN: End-to-End Speech Recognition using Deep RNN Models and WFST-based Decoding. *CoRR*, ročník abs/1507.08240, 2015.
URL http://arxiv.org/abs/1507.08240

[23] Mikolov, T.: *Statistical Language Models Based on Neural Networks*. Dizertační práce, 2012.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=10158

[24] Mohri, M.; Pereira, F.; Riley, M.: Weighted Finite-State Transducers in Speech Recognition. 2001.

[25] Pascanu, R.; Mikolov, T.; Bengio, Y.: On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, 2013, s. 1310–1318.
URL http://jmlr.org/proceedings/papers/v28/pascanu13.html

[26] Povey, D.; Ghoshal, A.; Boulianne, G.; aj.: The Kaldi Speech Recognition Toolkit. In *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding*, IEEE Signal Processing Society, Prosinec 2011, ISBN 978-1-4673-0366-8, iEEE Catalog No.: CFP11SRW-USB.

[27] Schuster, M.; Paliwal, K.: Bidirectional Recurrent Neural Networks. *Trans. Sig. Proc.*, ročník 45, č. 11, Listopad 1997: s. 2673–2681, ISSN 1053-587X, doi:10.1109/78.650093.
URL http://dx.doi.org/10.1109/78.650093

[28] Sutskever, I.; Martens, J.; Dahl, G.; aj.: On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, ročník 28, editace S. Dasgupta; D. Mcallester, JMLR Workshop and Conference Proceedings, Květen 2013, s. 1139–1147.
URL http://jmlr.org/proceedings/papers/v28/sutskever13.pdf

[29] Tieleman, T.; Hinton, G.: Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

[30] Weninger, F.: Introducing CURRENNT: The Munich Open-Source CUDA RecurREnt Neural Network Toolkit. *Journal of Machine Learning Research*, ročník 16, 2015: s. 547–551.
URL http://jmlr.org/papers/v16/weninger15a.html

[31] Young, S. J.; Kershaw, D.; Odell, J.; aj.: *The HTK Book Version 3.4*. Cambridge University Press, 2006.

[32] Yu, D.; Deng, L.: *Automatic Speech Recognition: A Deep Learning Approach*. Springer Publishing Company, Incorporated, 2014, ISBN 1447157788, 9781447157786.

[33] Šustr, Z.; Sitera, J.; Mulač, M.; aj.: MetaCentrum, the Czech Virtualized NGI. 2009.
URL
http://egee.cesnet.cz/cms/export/sites/egee/cs/info/virtualizace.pdf

Přílohy

Seznam příloh

A Obsah CD **69**

Příloha A

Obsah CD

Přiložený disk k této práci obsahuje následující:

- torch-lstm - implementaci rekurentních neuronových sítí,
- kaldi - verzi toolkitu kaldi a eesen, s upraveným skriptem run.sh pro databází *AMI*, kde je naznačen postup pro provedení experimentů s dopřednou a rekurentní sítí, jak pro objektivní funkci cross entropie tak pro objektivní funkci ctc,
- nn-tools - skripty sloužící pro práci s datasetem,
- zdrojové texty technické práce.