VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

RYCHLÉ DISKRIMINATIVNÍ NEURONOVÉ SÍTĚ PRO OPRAVU TEXTU

FAST DISCRIMINATING NEURAL NETWORKS FOR TEXT CORRECTION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **SEBASTIÁN CHUPÁČ**
AUTHOR

VEDOUCÍ PRÁCE **Ing. JAN KOHÚT**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

148399

Ústav:	Ústav počítačové grafiky a multimédií (UPGM)
Student:	**Chupáč Sebastián**
Program:	Informační technologie
Specializace:	Informační technologie
Název:	**Rychlé diskriminativní neuronové sítě pro opravu textu**
Kategorie:	Umělá inteligence
Akademický rok:	2022/23

Zadání:

1. Prostudujte základy neuronových sítí pro opravu textu.
2. Vytvořte si přehled o diskriminativních architekturách sítí, které umožňují rychlé opravy textu.
3. Vyberte nejvhodnější architektury a navrhněte architektury vlastní.
4. Obstarejte si databázi vhodnou pro experimenty.
5. Implementujte architektury a proveďte experimenty nad datovou sadou.
6. Porovnejte dosažené výsledky a diskutujte možnosti budoucího vývoje.
7. Vytvořte stručné video prezentující vaši práci, její cíle a výsledky.

Literatura:

- Nguyen, T.T.H., Jatowt, A., Coustaty, M. and Doucet, A., 2021. Survey of Post-OCR processing approaches. *ACM Computing Surveys (CSUR)*, *54*(6), pp.1-37.
- G. Chiron, A. Doucet, M. Coustaty and J. -P. Moreux, "ICDAR2017 Competition on Post-OCR Text Correction," *2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR)*, 2017, pp. 1423-1428, doi: 10.1109/ICDAR.2017.232.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3.

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

Vedoucí práce:	**Kohút Jan, Ing.**
Vedoucí ústavu:	Černocký Jan, prof. Dr. Ing.
Datum zadání:	1.11.2022
Termín pro odevzdání:	10.5.2023
Datum schválení:	31.10.2022

Abstrakt

Cieľom tejto práce je návrh a implementácia architektúry rýchlej diskriminatívnej neurónovej siete s jediným dopredným priechodom, ktorá deteguje a opravuje chyby v texte. Boli navrhnuté a implementované viaceré architektúry zvlášť pre detekciu a opravy chýb. Tieto modely využívajú najmä konvolučné a LSTM vrstvy a CTC stratovú funkciu. jednotlivé modely boli trénované a následne vyhodnotené na datasetoch z troch rôznych textových korpusov. Vyhodnotením a experimentami bola ukázaná schopnosť architektúr detegovať a opravovať chyby v texte na úrovni znakov jediným dopredným priechodom.

Abstract

The goal of this work is to propose and implement a fast discriminating neural network with only one forward pass, to detect and correct mistakes in text data. Multiple architectures were implemented for detection and correction separately. These models make use of convolution layers, LSTM layers and CTC loss function. Models were trained and evaluated on datasets made from three different text corpora. Experiments and evaluation present the ability of these models to detect and correct mistakes on character level with only one, fast forward pass.

Kľúčové slová

Neurónové siete, spracovanie prirodzeného jazyka, oprava textu, umelá inteligencia, strojové učenie, konvolučné neurónové siete, LSTM, CTC

Keywords

Neural networks, natural language processing, text correction, artificial intelligence, machine learning, convolutional neural networks, LSTM, CTC

Citácia

CHUPÁČ, Sebastián. *Rychlé diskriminativní neuronové sítě pro opravu textu.* Brno, 2023. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jan Kohút

Rychlé diskriminativní neuronové sítě pro opravu textu

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Jána Kohúta. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Sebastián Chupáč
9. mája 2023

Poďakovanie

Chcel by som poďakovať Ing. Jánovi Kohútovi za vedenie mojej práce, jeho čas a poskytnutie drahocenných rád. Ďalej by som chcel poďakovať Metacentru za poskytnutie výpočetných zdrojov. Výpočtové zdroje poskytol projekt e-INFRA CZ (ID:90140), podporilo Ministerstvo školstva, mládeže a telovýchovy ČR. Tiež by som chcel poďakovať mojej rodine za podporu pri štúdiu a vypracovaní tejto bakalárskej práce.

Obsah

1 Úvod **2**

2 Neurónové siete a ich využitie pri oprave textu **3**
2.1 Neurónové siete . . . 3
2.2 Konvolučné neurónové siete . . . 5
2.3 Rekurentné neurónové siete . . . 7
2.4 Long-Short Term Memory . . . 8
2.5 Connectionist Temporal Classification . . . 10
2.6 Súčasné riešenia v oblasti opravy textu . . . 11

3 Implementácia dátovej sady a architektúr pre opravu textu **14**
3.1 Korpus textu a tvorba dátovej sady . . . 14
3.2 Návrh riešenia . . . 16
3.3 Generovanie chýb . . . 17
3.4 Architektúry pre detekciu chýb . . . 18
3.5 Architektúry pre opravu chýb . . . 20
3.6 Trénovanie a trénovací skript . . . 25
3.7 Validačné a ostatné skripty . . . 27

4 Testovanie a experimenty **28**
4.1 Absolútna presnosť . . . 28
4.2 Experimenty . . . 28
4.3 Vyhodnotenie . . . 32

5 Záver **34**

Literatúra **35**

A Obsah priloženého pamäťového média **38**

Kapitola 1

Úvod

Táto práca sa zaoberá využitím rýchlych diskriminatívnych neurónových sietí pre nájdenie a opravenie chýb v texte. Umelá inteligencia a strojové učenie napredujú nevídaným tempom. Využívajú sa na riešenie problémov v mnohých oblastiach, jednou z nich je aj porozumenie ľudskému jazyku.

V písanom texte na počítači, prípadne smartfóne či inom zariadení, sa chyby najčastejšie prejavujú vo forme preklepov alebo nesprávnych gramatických zápisov slov. Textové dáta však môžu vznikať aj iným spôsobom, napríklad prevodom skenovaných dokumentov na textové dáta, rozpoznávaním textu z fotografií či rozpoznávaním reči zo zvukových záznamov. Takto získaný text môže tiež obsahovať množstvo chýb. Tie už ale niesu spôsobené človekom, ale často nedokonalou kvalitou vstupných dát a (ne)presnosťou použitej metódy prevodu. Práve neurónové siete sa v minulosti osvedčili ako jeden z najlepších nástrojov na riešenie problému korekcie textu. Zameral som sa na detekciu a opravy chýb na úrovni znakov, ktoré je možné rozdeliť do 3 kategórií: zamenený znak, nadbytočný znak a vynechaný znak. Cieľom práce je predstaviť alternatívnu architektúru k bežne používaným postupom auto-regresívneho generovania textu opakovanými priechodmi robustnými rekurentnými modelmi, založenú na jednom rýchlom doprednom priechode sieťou. Tento prístup by mohol umožniť generovať opravený text v real-time čase aj bez prístupu k výkonným grafickým kartám priamo na hardvéry používateľa či bez pripojenia na internet.

V kapitole 2 je krátky úvod do problematiky neurónových sietí a využitých technológií ako LSTM a CTC. V sekcií 2.6 je stručný prehľad o aktuálne využívaných autoregresívnych prístupoch v oblasti korekcie textu. Kapitola 3 obsahuje detailný popis návrhu a implementácie dátovej sady a generovania chýb, sú tu predstavené konkrétne architektúry pre detekciu a opravu chýb v texte. V kapitole 4 sú popísané vykonané experimenty a dosiahnuté výsledky.

Kapitola 2

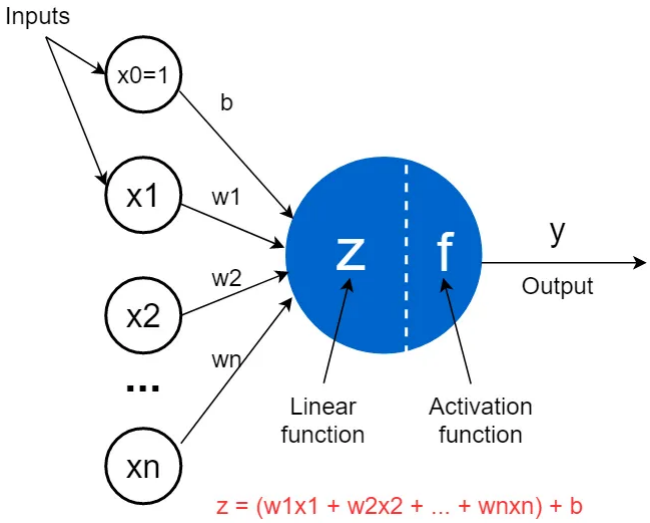
Neurónové siete a ich využitie pri oprave textu

Táto kapitola obsahuje potrebné teoretické informácie pre pochopenie klasických, konvolučných a rekurentných neurónových sietí, ktorých prvky sú využívané v mojom riešení. Sú tu vysvetlené aj princípy fungovania Long-Short Term Memory vrstiev a Connectionist Temporal Classification. Na konci je prehľad aktuálne používaných prístupov k oprave textu za využitia neurónových sietí.

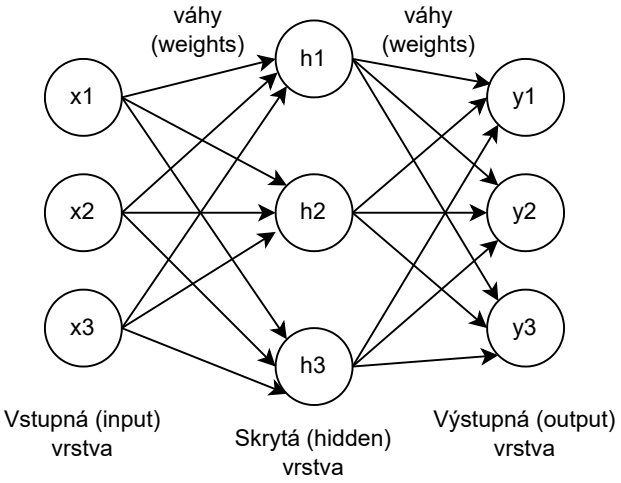
2.1 Neurónové siete

Základná myšlienka neurónových sietí vychádza zo spôsobu akým informácie spracováva biologický nervový systém, konkrétne ľudský mozog. Ten obsahuje približne 10^{11} buniek zvaných neuróny, ktoré medzi sebou komunikujú vysielaním elektrických signálov. Neurón sa môže nachádzať v jednom z dvoch stavov: pasívny alebo aktívny. Každý neurón je prepojený s tisíckami ďalších v miestach nazývaných synapsie, ktoré podľa ich sily (váhy) upravia hodnotu signálu, ktorý cez ne prechádza. Pri komunikácii neurón v podstate vypočíta vážený súčet všetkých výstupných signálov neurónov, ktoré sú s ním spojené, a základe tejto hodnoty určí svoj vlastný stav, ktorý odošle ďalším neurónom [2].

Umelé neuróny, tiež nazývané perceptrony, sú matematické modely silne inšpirované práve biologickými neurónmi. Schéma umelého neurónu je na obrázku 2.1. Štruktúru a chovanie perceptronu môžme popísať nasledovne: vstupné hodnoty $x_1 \dots x_n$ sú každá vynásobená príslušnou váhou $w_1 \dots w_n$ a sčítané do jednej hodnoty, ku ktorej sa ešte pripočíta hodnota *bias* b. Toto je lineárna funkcia perceptronu. Jej výsledok je vstupom pre ne-lineárnu (aktivačnú) funkciu. Aktivačná funkcia určuje, aké hodnoty môže umelý neurón nadobúdať. Hlavnou úlohou aktivačnej funkcie je ale priviesť do modelu neurónovej siete nelineárnu závislosť, ktorá sa bežne vyskytuje v dátach ktoré neurónová sieť spracováva [20]. Medzi najpoužívanejšie aktivačné funkcie patria Sigmoid, Tanh, ReLu a Leaky ReLu [5].



Obr. 2.1: Matematická štruktúra umelého neurónu. Hodnota bias sa môže zobrazovať ako ďalší vstup s hodnotou 1 a váhou hodnoty bias. Prevzaté z [20].



Obr. 2.2: Štruktúra doprednej (feed-forward) neurónovej siete. V každej vrstve môže byť rôzny počet neurónov. Ak má neurónová sieť viac ako jednu skrytú vrstvu, môžme hovoriť o hlbokej neurónovej sieti.

Umelé neuróny sú v základnom modely doprednej (feed forward) neurónovej siete usporiadané vo vrstvách. Zjednodušený model doprednej neurónovej siete, s 3 vrstvami, v každej s 3 neurónmi, je zobrazený na obrázku 2.2. Vstupné hodnoty sú privedené na vstupnú vrstvu (input layer), prejdú cez jednu alebo viacero skrytých vrstiev (hidden layers) až sa dostanú na poslednú výstupnú vrstvu (output layer) [1]. Neuróny sú prepojené medzi sebou vždy medzi dvoma za sebou nasledujúcimi vrstvami. Výsledná štruktúra je vlastne parametrizovateľná matematická funkcia, ktorej parametre sa snažíme odhadnúť aby sme dosiahli požadovanú funkcionalitu. Odhadujeme ich numericky, pretože analytické riešenie nieje možné. Tento prístup sa dá aplikovať na rôzne problémy, ja ho využívam na opravu chýb v texte. Architektúry ktoré som implementoval využívajú konvolučné vrstvy popísané

v sekcií 2.2, rekurentné LSTM vrstvy popísané v sekcií 2.4 a CTC stratovú funkciu popísanú v sekcií 2.5.

Trénovanie neurónových sietí. Keď sa neurónová sieť trénuje (učí), upravujeme jej parametre, teda hodnoty váh a hodnoty bias tak, aby dosiahli ich optimálne hodnoty pre riešenie danej úlohy [20]. Trénovanie zvyčajne prebieha tak, že minimalizujeme nejakú stratovú (loss) funkciu, hľadáme jej globálne minimum. Nájsť globálne minimum v praxi je však takmer nemožné, preto trénovacie prístupy hľadajú nejaké dostatočne dobré lokálne minimum. Existuje množstvo loss funkcií a je potrebné vybrať tú správnu podľa problému ktorý riešime a architektúry ktorú používame. Často používané loss funkcie sú napríklad Mean Absolute Error Loss, Mean Squared Error Loss pre regresívne problémy, Cross-Entropy Loss pre klasifikáciu.

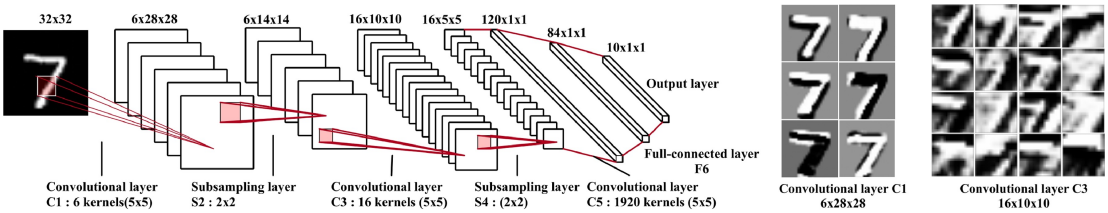
Na minimalizovanie hodnoty stratovej funkcie sa využíva algoritmus nazývaný *Gradient Descent*. Tento algoritmus vypočíta gradient, čiže vektor parciálnych derivácií funkcie vzhľadom na aktuálne vstupné hodnoty (parametre) funkcie. Aby sa hodnota funkcie posunula bližšie k lokálnemu minimu stratovej funkcie, odčíta sa od parametrov tento gradient vynásobený vhodne zvoleným učiacim koeficientom (learning rate). Ten udáva veľkosť kroku v danom smere a jeho hodnota je kľúčová pri trénovaní neurónovej siete. Príliš malý learning rate spôsobí pomalú konvergenciu, zatiaľ čo príliš veľký preskočí lokálne minimum a spôsobí oscilovanie parametrov. Vypočítať gradient pre každú položku z dátovej sady a každý parameter siete je extrémne náročné na výpočetný čas a zdroje, preto sa používa *Stochastic Gradient Descent* (SGD), ktorý do stratovej funkcie pošle iba časť položiek a vypočíta teda iba odhad gradientu pre celú dátovú sadu [10, 4].

Otázkou je ako vypočítať gradienty pre skryté vrstvy neurónovej siete a pre ktorýkoľvek parameter. O toto sa stará algoritmus zvaný Backpropagation, ktorý využíva reťazové pravidlo derivácií (chain rule). Toto pravidlo využíva známe hodnoty derivácií funkcií pre výpočet derivácií zložených funkcií a je aplikované rekurzívne na ďalšie funkcie. Backpropagation algoritmus pre každú váhu a bias vypočíta gradient, ktorý potom môže využiť optimalizačný algoritmus napríklad SGD na upravenie parametrov modelu. Backpropagation využíva výpočtový graf siete, ktorým spätne propaguje hodnoty od výstupnej vrstvy po vstupnú a postupne počíta gradienty [4].

2.2 Konvolučné neurónové siete

Konvolučné neurónové siete sú špeciálny typ neurónových sietí navrhnuté tak, aby pracovali s 2-dimenzionálnymi dátami ako sú napríklad obrázky, ale dajú sa použiť s obecne n-dimenzionálnymi dátami [3].

Prvým typom vrstvy v konvolučných neurónových sieťach, po ktorom sú aj konvolučné neurónové siete pomenované, je konvolučná vrstva. Táto vrstva má za úlohu naučiť sa ako sú vo vstupných dátach reprezentované jednotlivé vlastnosti (features). Na obrázku 2.3 v časti (b) sú zobrazené mapy vlastností. V časti (a) je schéma známej LeNet-5 konvolučnej architektúry. Podobne ako v bežných neurónových sieťach aj v tejto vrstve dochádza k násobeniu vstupných dát do tejto vrstvy s množinou váh. Pri 2-D konvolúcii dochádza k výpočtu skalárneho súčinu vstupného poľa a menšieho 2-D poľa váh, ktoré sa nazýva filter alebo kernel. Keďže je kernel menší ako vstupné dáta a výsledkom skalárneho súčinu je jedna hodnota, tento filter sa aplikuje viac krát postupne na celé pole vstupných dát tak, že sa postupne posúva zľava doprava a z hora dolu. Takýmto postupom získame 2-D



(a) LeNet-5 network (b) Learned features

Obr. 2.3: Obrázok (a) zobrazuje populárnu LeNet-5 konvolučnú sieť. Obrázok (b) zobrazuje naučené mapy vlastností. Prevzaté z [13].

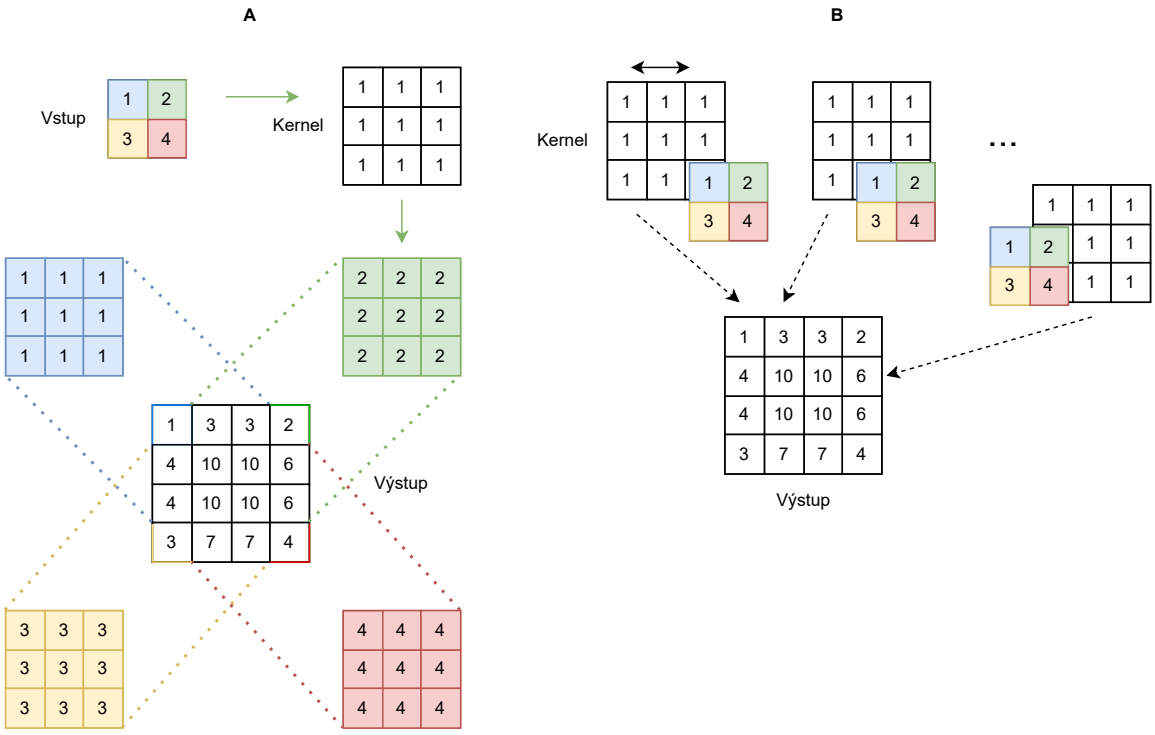
pole výsledkov všetkých týchto operácií, ktoré sa nazýva mapa vlastností (feature map). Posledným krokom je aplikácia nelineárnej (aktivačnej) funkcie na mapu vlastností, napríklad sigmoid či ReLu. Konvolučné vrstvy sa typicky zapájajú za sebou, a postupne dokážu z dát získavať vlastnosti stále vyššej a vyššej úrovne. Prvé vrstvy ktoré pracujú priamo s pixelmi obrázka z neho dokážu získať informácie o čiarach, a nasledujúce vrstvy dokážu rozpoznávať vlastnosti zložené z týchto čiar, napríklad tvary [3].

Ďalšou typicky používanou operáciou v konvolučných neurónových sieťach je pooling. Táto vrstva počíta nejakú štatistiku z výstupu nelinearity. Napríklad operácia max pooling so šírkou kernelu 3 vypočíta maximálnu hodnotu z každých 3 susediacich výstupov z predošlej aktivačnej funkcie. Average pooling počíta rovnakým spôsobom priemernú hodnotu. Pooling nám do siete prináša čiastočnú invariantnosť voči posunutiu a transformácií vstupu. Často sa tiež využíva na redukciu veľkosti pomocou parametru stride, ktorý ovláda o koľko hodnôt sa kernel posunie medzi výpočtami [10].

Po niekoľkých konvolučných a pooling vrstvách môže nasledovať skupina plne prepojených (dense) vrstiev. Tieto vrstvy prepájajú každý neurón z predchádzajúcej vrstvy s každým neurónom v danej vrstve. To umožňuje zachovať sémantické informácie naprieč celým vstupom [13].

Princíp transponovanej konvolúcie. Transponovaná konvolúcia niekedy označovaná aj ako dekonvolúcia je operácia, ktorá sa často používa v strojovom učení ako opačná operácia ku konvolúcií. Nejedná sa však o pravú inverznú operáciu ku konvolúcií. Keď je na vstupné dáta aplikovaná konvolúcia a na výstup z nej transponovaná konvolúcia s rovnakými parametrami (padding, stride, kernel), výstupom z transponovanej konvolúcie nebudú rovnaké dáta ako pôvodne vstúpili do konvolúcie. Výstupné dáta budú mať však rovnaké rozmery ako pôvodné vstupné dáta do konvolúcie. Tak ako konvolúcia spôsobuje redukciu rozmerov dát, transponovaná konvolúcia spôsobuje zväčšenie rozmerov.

V transponovanej konvolúcií dochádza k úplne rovnakej operácií ako v klasickej konvolúcií, ale parametre stride a padding tu fungujú opačným spôsobom, aby došlo k opačnej zmene rozmerov ako pri konvolúcií, pri zachovaní rovnakých hodnôt týchto parametrov. Na obrázku 2.4 je vizualizovaný príklad transponovanej konvolúcie so vstupom, ktorý má rozmery 2x2, jadro (kernel) o rozmeroch 3x3, bez paddingu a stride 1. Vznikne výstup s rozmermi 4x4. V časti A je vidieť ako vzniká výstup keď každú hodnotu zo vstupu navzorkujeme násobením s kernelom, a výsledné matice uložíme cez seba vždy vedľa o 1 pozíciu (stride = 1). V časti B je rovnaká operácia vyzualizovaná ako konvolúcia postupným posúvaním jadra a počítaním skalárneho súčinu. Padding má v transponovanej konvolúcií rovnakú funkciu ako pri obyčajnej konvolúcí – zmeniť rozmery výstupu. Keďže transponovaná konvolúcia prirodzene zväčšuje rozmery výstupu pretože sa kernel pohybuje ako keby



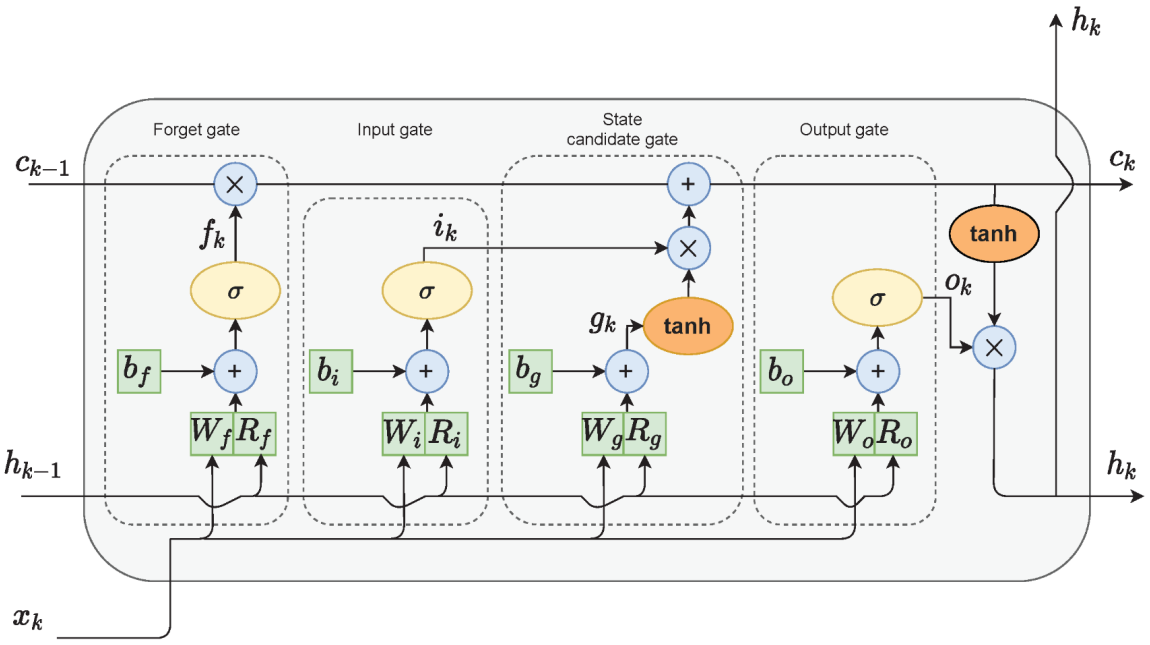
Obr. 2.4: Príklad dekonvolúcie vyzualizovaný intuitívne ako opačná operácia ku konvolúcií (A), a ako konvolúcia s opačným fungovaním parametrov padding a stride (B).

už bol full padding aplikovaný na vstup, zvýšenie hodnoty paddingu zabráni kernelu pohybovať sa až úplne po okraj a veľkosť výstupu sa zmenší. Podobne je to aj s parametrom stride, ktorý v obyčajnej konvolúcií spôsobí pohyb jadra o viac pozíc v jednom kroku, čím sa rozmery výstupu ešte viac znižujú. V transponovanej konvolúcií stride riadi pridanie nulových hodnôt do vstupných dát čo spôsobí presne opačný efekt ako stride v bežnej konvolúcií [21].

2.3 Rekurentné neurónové siete

Rekurentné neurónové siete (RNN) sú špeciálne navrhnuté na spracovanie postupnosti hodnôt (sekvenčných dát) v čase. Môže sa napríklad jednať o spracovanie zvuku (postupnosť vzorkov signálu) či spracovanie prirodzeného jazyka (postupnosť znakov a slov). Pri výpočte aktuálneho stavu totiž rekurentné neurónové siete využívajú nielen aktuálne dáta na vstupe, ale aj informácie o svojom predchádzajúcom stave a teda aj vstupných dátach z predošlých časových krokov. To znamená že výstup RNN nieje závislí iba na aktuálnych vstupných dátach, ako u klasických dopredných neurónových sietí. Časové kroky nemusia predstavovať skutočný čas, môže sa jednať len o poradie v postupnosti.

Ďalšou vlastnosťou ktorou disponujú RNN je zdieľanie parametrov (váh). Keď RNN počíta výstup pre aktuálny vstup, aplikuje sadu váh na vstup a ďalšiu sadu váh na svoj vnútorný stav z minulého časového kroku a výsledok potom prejde aktivačnou funkciou. Tieto sady váh sú však zdielané a teda rovnaké pre všetky časové kroky. Zdieľanie parametrov prináša niekoľko výhod: menej parametrov ktoré je potrebné natrénovať, lepšia generalizácia modelu a nižšie riziko pretrénovania (overfitting), ako aj schopnosť spracovať



Obr. 2.5: Schéma LSTM bloku. Prevzaté z [26] a upravené.

vstup ľubovoľnej dĺžky. Pri trénovaní je potom potrebné sčítať hodnoty loss funkcie cez všetky časové kroky, keďže boli použité rovnaké parametre. Toto prináša jeden z hlavných problémov RNN – vyhasínajúci prípadne explodujúci gradient. Tento problém sa prejavuje najmä pri dlhých vstupných sekvenciách, kedy sa gradient používaný na aktualizovanie hodnôt váh postupne zmenšuje až sa hodnoty váh nemenia a model sa neučí, alebo zväčšuje čo spôsobí že hodnoty váh začnú oscilovať [10, 25].

2.4 Long-Short Term Memory

Long-Short Term Memory (LSTM) architektúra bola predstavená v článku [14] ako riešenie problému vyhasínajúceho a explodujúceho gradientu v bežných rekurentných neurónových sieťach. Jadrom LSTM architektúry sú pamäťové bunky (memory cells), ktoré sú zapojené do mechanizmu s názvom pamäťový blok (memory block). Schéma pamäťového bloku je na obrázku 2.5. Pamäťový blok v LSTM vrstve obsahuje skrytý stav (hidden state), na obrázku označený h, ktorý využíva ako krátkodobú pracovnú pamäť, a bunkový stav (cell state), na obrázku označený c, ktorým si pamätá dôležité informácie zo vzdialenejšej minulosti. V bloku sa vyskytujú dve aktivačné funkcie – tanh, ktorá vracia hodnoty z intervalu $(-1, 1)$ a sigmoid, ktorá vracia hodnoty z intervalu $(0, 1)$. Aby sa predišlo problému vyhasínajúceho a explodujúceho gradientu, sú na bunkový stav c aplikované len operácia sčítania a operácia násobenia s výstupom funkcie sigmoid. Práve keď je výstup sigmoid funkcie blízko 1 dochádza k zachovaniu gradientu inak k jeho redukcií. Súčasťou pamäťového bloku je mechanizmus brán (gates), ktoré riadia zmeny bunkového stavu a prúdenie informácií v bloku [26].

Na obrázku 2.5 je jeden LSTM blok. Obsahuje 4 brány, ktoré majú nasledujúcu funkcionalitu:

- Zabúdacia brána f (forget gate) – rozhoduje, ktoré informácie z predošlého bunkového stavu c (cell state) budú zabudnuté.
- Vstupná brána i (input gate) – vyberá aké množstvo informácií z predošlého skrytého stavu h (hidden state) a aktuálneho vstupu x sa zapíše do bunkového stavu a teda do dlhodobej pamäti.
- State candidate gate g – reguluje tok informácií z predošlého skrytého stavu a aktuálneho vstupu využitím tanh aktivačnej funkcie, potom jej výstup vynásobí s výstupom funkcie sigmoid zo vstupnej brány, a pripočíta k aktuálnemu bunkovému stavu čiže dlhodobej pamäti.
- Výstupná brána (output gate) – aktuálny vstup a predchádzajúci skrytý stav sčíta a pošle do sigmoid funkcie, aktuálny bunkový stav do tanh funkcie, tieto dve hodnoty vynásobí, a výsledkom je nový skrytý stav a zároveň výstup z LSTM pre aktuálny časový krok.

Symbol W označuje váhy aplikované na vstupný signál x, R označuje rekurzívne váhy aplikované na predošlí skrytý stav a b označuje hodnoty bias. V každom časovom kroku k sú v LSTM vrstve sekvenčne vypočítané nasledujúce hodnoty:

$$i(k) = \sigma(W_{\mathrm{i}}x(k) + R_{\mathrm{i}}h(k-1) + b_{\mathrm{i}}), \tag{2.1}$$

$$f(k) = \sigma(W_{\mathrm{f}}x(k) + R_{\mathrm{f}}h(k-1) + b_{\mathrm{f}}), \tag{2.2}$$

$$g(k) = \tanh(W_{\mathrm{g}}x(k) + R_{\mathrm{g}}h(k-1) + b_{\mathrm{g}}), \tag{2.3}$$

$$o(k) = \sigma(W_{\mathrm{o}}x(k) + R_{\mathrm{o}}h(k-1) + b_{\mathrm{o}}). \tag{2.4}$$

Po výpočte hodnôt pre všetky brány je vypočítaný nový bunkový stav v čase k:

$$c(k) = f(k) \circ c(k-1) + i(k) \circ g(k). \tag{2.5}$$

Nakoniec sa vypočíta skrytý stav resp. výstup v čase k:

$$h(k) = o(k) \circ \tanh(c(k)). \tag{2.6}$$

Symbol $\circ$ značí násobenie matíc po prvkoch (Hadamard product) [26].

Implementácia LSTM architektúry v PyTorch je takmer identická predstavenému modelu, líši sa tým, že v každej bráne je ešte jedna hodnota bias, ktorá sa pripočíta pred aktivačnou funkciou.

Obojsmerné (bidirectional) LSTM je vhodné využiť v problémoch, kedy chceme generovať výstup nielen na základe hodnôt z minulosti, ale aj tých z budúcnosti. Napríklad pri oprave textu môžeme využiť celú vstupnú sekvenciu na generovanie výstupu. Tento postup bol predstavený v [11]. V podstate sa jedná o dva LSTM modely s rovnakou vstupnou sekvenciou, ale pre druhý model je sekvencia otočená. Výstupy oboch modelov sa v každom časovom kroku spoja (concat) takže obsahujú skryté stavy z oboch architektúr [25].

2.5 Connectionist Temporal Classification

Connectionist Temporal Classification (CTC) je spôsob ako mapovať jednu vstupnú sekvenciu na inú výstupnú sekvenciu, využívaný najmä v rekurentných neurónových sieťach. Zahŕňa spôsob ako enkódovať vstupnú sekvenciu časových krokov, vypočítať vhodnú stratovú (loss) funkciu pre neurónovú sieť, a dekódovať výstup z RNN na požadovanú sekvenciu napríklad znakov či slov. Využíva sa primárne pri rozpoznaní reči alebo textu z obrazu, kedy by bolo veľmi náročné presne pozične anotovať všetky trénovacie dáta [24].

Keď chceme využiť CTC s rekurentnou neurónovou sieťou, a vytvoriť tak CTC neurónovú sieť (CTCNN), je potrebné najskôr zaistiť, aby výstupom neurónovej siete bola matica obsahujúca pravdepodobnosti pre každý znak z použitej abecedy pre každý časový krok. To znamená že jednotlivé hodnoty musia byť z intervalu $(0, 1)$ a súčet všetkých týchto hodnôt v každom časovom kroku sa musí rovnať 1. Toho dosiahneme aplikovaním softmax funkcie na výstupnú vrstvu siete. Ukážka CTC pravdepodobnostnej matice je na obrázku 2.6. Stĺpce v tejto matici predstavujú rozloženie pravdepodobnosti pre jeden časový krok naprieč abecedou znakov. Táto matica nám umožňuje vypočítať pravdepodobnosti pre všetky možné zarovnania výstupnej sekvencie. Zarovnaním (path) je myslený výber jedného znaku z abecedy pre všetky časové kroky. Pri trénovaní sa počítajú len zarovnania ktoré odpovedajú ground truth. Voláme ich validné zarovnania. Ostatné zarovnania, teda také, ktoré po dekódovaní niesu rovné ground truth, sú nevalidné. Pravdepodobnosť konkrétnej sekvencie zistíme sčítaním pravdepodobností jej všetkých možných zarovnaní [12].

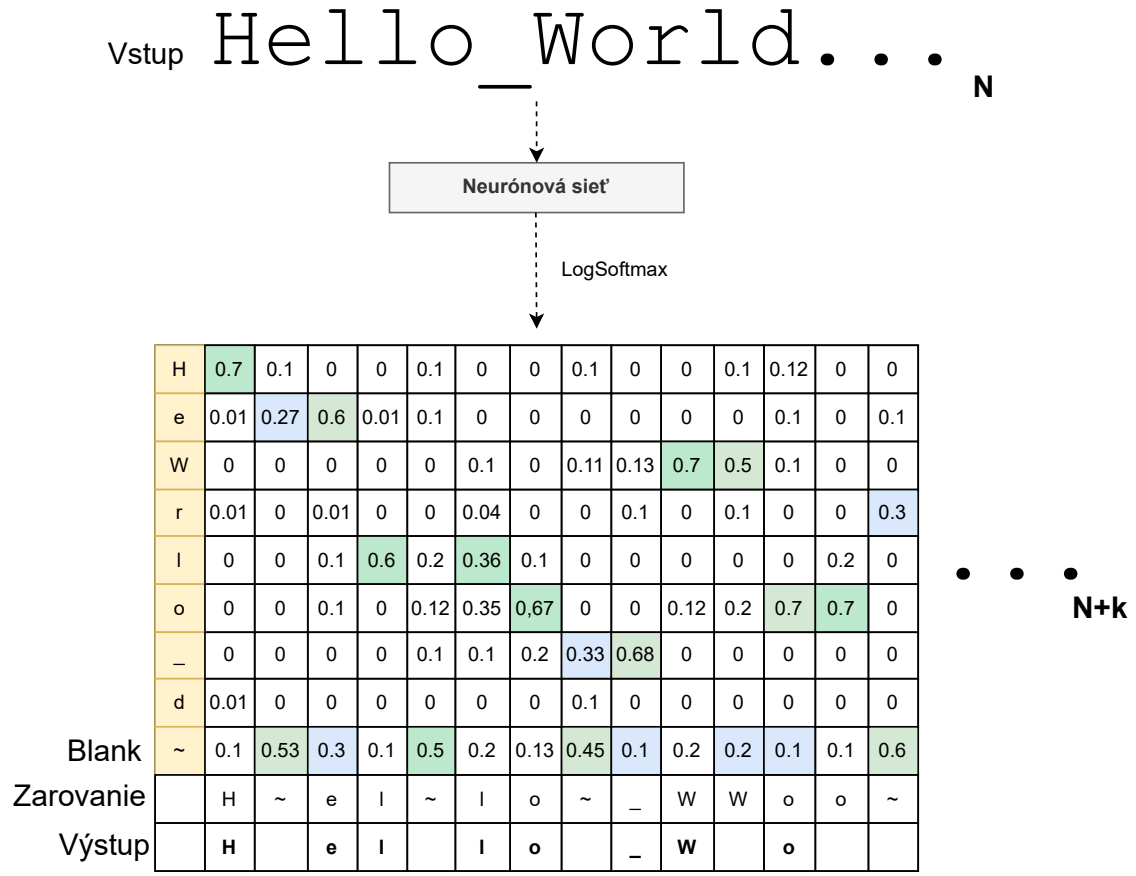
CTC stratová funkcia vyžaduje v abecede znakov vyhradiť jeden znak, ktorý sa použije ako "prázdny"znak ("blank"), na obrázku ho reprezentuje ~. Ten umožňuje CTC neurónovej sieti generovať aj sekvencie obsahujúce viac rovnakých znakov za sebou, tým že medzi rovnaké znaky vloží blank, bez neho by boli tieto sekvencie rovnakých znakov zredukované len na jeden znak pri dekódovaní. Za zmienku stojí aj fakt, že dĺžka výstupu z neurónovej siete musí byť minimálne taká veľká, ako dĺžka ground truth sekvencie, ktorú chceme dosiahnuť.

Najjednoduchší spôsob ako dekódovať výstup z CTCNN je vybrať znak s najvyššou hodnotou pravdepodobnosti (score) pre každý časový krok. Na obrázku je toto zarovnanie vyznačené zelenou farbou, a iné validné zarovnanie modrou. Táto sekvencia znakov sa upraví zredukovaním všetkých postupností rovnakých znakov len na jeden znak, a následným odstránením všetkých znakov blank. Toto riešenie známe pod názvom best path decoding ale negarantuje, že výsledkom bude sekvencia s najvyššou pravdepodobnosťou (najvyšším súčtom pravdepodobností všetkých možných zarovnaní tejto sekvencie) [24].

Pri trénovaní CTCNN sa využíva CTC stratová funkcia. Tá sa vypočíta ako suma záporných log všetkých validných zarovnaní vzhľadom ku gorund truth vzťahom:

$$L(p(l|x)) = -\sum_{t=1}^{T} \ln(C_t), \tag{2.7}$$

kde l je výstupná sekvencia, x je vstupná sekvencia, C_t je validné zarovnanie [12].



H	0.7	0.1	0	0	0.1	0	0	0.1	0	0	0.1	0.12	0	0
e	0.01	0.27	0.6	0.01	0.1	0	0	0	0	0	0	0.1	0	0.1
W	0	0	0	0	0	0.1	0	0.11	0.13	0.7	0.5	0.1	0	0
r	0.01	0	0.01	0	0	0.04	0	0	0.1	0	0.1	0	0	0.3
l	0	0	0.1	0.6	0.2	0.36	0.1	0	0	0	0	0	0.2	0
o	0	0	0.1	0	0.12	0.35	0,67	0	0	0.12	0.2	0.7	0.7	0
_	0	0	0	0	0.1	0.1	0.2	0.33	0.68	0	0	0	0	0
d	0.01	0	0	0	0	0	0	0.1	0	0	0	0	0	0
~	0.1	0.53	0.3	0.1	0.5	0.2	0.13	0.45	0.1	0.2	0.2	0.1	0.1	0.6
	H	~	e	l	~	l	o	~	_	W	W	o	o	~
	H		**e**	**l**		**l**	**o**		**_**	**W**		**o**		

Obr. 2.6: Ukážka pravdepodobnostnej matice teda výstupu z CTCNN. Zarovnanie s najvýššou pravdepodobnosťou je vyznačené zelenou farbou. Alternatívne validné zarovnanie modrou.

2.6 Súčasné riešenia v oblasti opravy textu

Spracovanie prirodzeného jazyka (Natural languege processing – NLP) je odvetvie umelej inteligencie, ktoré využíva robustné štatistické modely a strojové učenie na porozumenie a prácu s ľudským jazykom. Popularitu aj v laickej verejnosti si získali projekty ako Chat-GPT či Microsoft Bing, ktoré sú schopné viesť komplexné konverzácie a odpovedať na najrôznejšie otázky užívateľa. Keď užívateľ požiada Chat-GPT alebo Bing o skontrolovanie nejakej vety alebo textu, jazykový model mu takmer vždy v odpovedi uvedie správnu verziu textu. Jednými z hlavných nevýhod týchto sofistikovaných modelov, obsahujúcich miliardy trénovaných parametrov, je ich veľkosť a výpočetná náročnosť. Sú to autoregresívne modely, ktoré sú aj kvôli svojej veľkosti pomerne pomalé.

Optické rozoznávanie znakov (Optical Character Recognition – OCR) je metóda prevodu obrazových dát tlačených alebo písaných ručne, do textovej editovateľnej podoby. Text ktorý takto vznikne často obsahuje množstvo chýb, čo môže byť spôsobené hlavne kvalitou vstupných dokumentov alebo použitím staršej OCR metódy [6]. Prístupy ktoré riešia detekciu a opravu chýb v post-OCR textoch sú naozaj rozmanité a zahrňujú manuálne prepisovanie a opravu textov, a (semi-)automatické prístupy. (Semi-)automatické prístupy sa dajú deliť na prístupy ktoré pracujú s izolovanými slovami, a prístupy ktoré

využívajú nejaký kontext [19]. V tejto kapitole sa zameriam najmä na prístupy využívajúce kontext a neurónové siete.

Prístupy využívajúce kontext dokážu opraviť aj real-word chyby, to znamená chyby ktoré zo správneho slova vytvoria nejaké iné skutočné slovo. Túto vlastnosť im umožňuje práve kontext okolitých slov.

D'hondt a spol. [8] predstavili obojsmerný (bidirectional) LSTM jazykový model pracujúci na znakovej úrovni. Ten je aplikovaný na Francúzske klinické digitalizované texty. Autori vytvorili 2 modely jeden *randomNoise*, na náhodné úpravy mazaním, vkladaním a výmenou znakov v texte, druhý *confusionPair*, zameraný na často zamieňané znaky. Presnosť prvého udávajú ako 73% a druhého 71%. Opravovanie chýb typu nadbytočný znak je obtiažne pre oba modely. Väčšinou sa jednalo o vkladanie medzier do slov. Preto majú modely taktiež problémy detegovať chyby týkajúce sa hraníc slov [8].

Niektoré prístupy považujú opravu textu za problém strojového prekladu (machine translation – MT) a môžme ich označiť ako neurónové Seq2Seq modely. Tieto riešenia prekladajú post-OCR text na opravený text v rovnakom jazyku [19].

Nastase a Hitschler [18] využívajú open-source nástroj *nematus*[1], ktorý poskytuje state-of-the-art attention-based Seq2Seq machine translation model. Využívajú tento model na znakovej úrovni, na opravovanie chybných hraníc slov (word boundary errors). Vstupné trénovacie dáta boli vytvorené odstránením všetkých medzier z textu a následným vložením medzery za každý znak. Výstupné trénovacie dáta boli vytvorené nahradením medzier v texte špeciálnou sekvenciou (##), a následným vložením medzery za každý znak okrem znakov ##. Dosiahnuté výsledky na testovacích dátach sú presnosť (word-level precision) 0.955 recall hodnota 0.950. Najčastejšie chyby zaznamenali v matematických vzorcoch, ktoré keď do výpočtov nezahrnuli, hodnoty presnosti a recall boli 0.979 a 0.974 [18].

Prístup využívajúci Seq2Seq model na znakovej úrovni predstavili Dong a Smith [9]. Využívajú skutočnosti, že množstvo korpusov OCR textov obsahuje duplicitné texty v rôznej kvalite. Rovnaké dokumenty prevedené OCR metódami v rôznych edíciách, obsahujú rôzne chyby, pretože rôzne edície toho istého dokumentu môžu mať rôzny font, kvalitu a rozloženie. Pri trénovaní detegujú tieto duplicitné texty v post-OCR korpuse, a používajú ich ako viaceré vstupy do architektúr s cieľom získať lepšie výsledky, ako pri opravovaní iba na základe jedného vstupného textu. Enkóder je obojsmerná rekurentná neurónová sieť ktorá konvertuje vstupnú sekvenciu na sekvenciu spojených dopredných a spätných skrytých stavov (hidden states) RNN. Dekóder je RNN ktorá pri predikcií výstupnej sekvencie využíva aj kontextový vektor pre daný časový krok. Ich výsledky ukazujú, že učenie bez učiteľa (unsupervised learning) dosahuje porovnateľné výsledky ako učenie s učiteľom (supervised learning), pri využití duplicitných textov v post-OCR korpusoch [9].

Mokhtar a spol. [17] experimentujú s enkóder-dekóder architektúrami na úrovni slov aj znakov. Architektúry obsahujú 3 a 4 vrstvy LSTM buniek v enkóder a dekóder častiach, a počet LSTM skrytých stavov je 1024. Keďže architektúra na úrovni slov má problémy so slovami, ktoré niesu v slovníku, predstavujú normalizačný algoritmus, ktorý je aplikovaný na chybný post-OCR text a na korešpondujúcu predikciu, a po slovách postupne vyberá buď predikciu alebo pôvodné slovo. Tento algoritmus značne vylepšil výsledky architektúry pracujúcej na úrovni slov, no tie stále neboli dosť dobré na praktické využitie. Model pracujúci na úrovni znakov dosahoval lepšie výsledky ako ostatné state-of-the-art prístupy na všetkých dátových sadách. Aj výstup tohoto modelu bol mierne vylepšený normalizačným algoritmom [17, 19].

[1] https://github.com/EdinburghNLP/nematus

Neuspell [15] je open-source sada nástrojov (toolkit), zdarma pre verejné využitie, poskytujúca 10 modelov na opravu anglického textu. Tieto modely sú zamerané hlavne na správne využitie kontextu v okolí chybných slov. Modely sú naučené pre každé slovo vo vstupnej sekvencií vrátiť rozloženie pravdepodobnosti naprieč konečným slovníkom. Niektoré modely ktoré Neuspell obsahuje: SC-LSTM [23] využíva semi-znakovú (semi-character) reprezentáciu dát a bi-LSTM sieť. Semi-znaková reprezentácia slova znamená spojenie one-hot embeddingov pre prvý, posledný a všetky vnútorné znaky slova. CHAR-LSTM-LSTM [16] model reprezentuje slová po jednotlivých znakoch, ktoré sú vstupom do bi-LSTM architektúry. Opravu slov generuje ďalšou bi-LSTM architektúrou. BERT [7] využíva pred-trénovanú transformer architektúru a klasifikátor na predikciu opravy. Tento model dosahoval konzistentne dobré výsledky v testoch, a pri porovnaní s plateným profesionálnym nástrojom Grammarly, BERT dosiahol lepšie výsledky pri opravovaní textu [15].

Ďalšie riešenia detekcie a opravy chýb v texte sa predstavili na súťažiach ICDAR 2019 Competition on Post-OCR Text Correction [22] a ICDAR 2017 Competition on Post-OCR Text Correction [6].

Kapitola 3

Implementácia dátovej sady a architektúr pre opravu textu

Táto kapitola dokumentuje postup implementácie, vrátane návrhu, získania dát a tvorby dátovej sady (dataset), implementácie trénovacích a testovacích skriptov, generovania chýb a samotných modelov neurónových sietí. Využíva sa Python s knižnicou PyTorch a knižnica Levenshtein na počítanie editačnej vzdialenosti medzi reťazcami.

3.1 Korpus textu a tvorba dátovej sady

V oblasti strojového učenia sú dáta veľmi dôležitým faktorom, ktorý má veľký vplyv na celkovú kvalitu výsledku. Pri implementácií a experimentoch som využil viacero textových korpusov v plain text formáte v anglickom jazyku. Jeden korpus ktorý som využil je SciFi Stories Text Corpus[1], ktorý pôvodne vytvoril Robin Sloan. Ďalším zdrojom textových dát bol anglický dump wikipedie[2]. Posledný korpus je výstup z projektu 1-billion-word-language-modeling-benchmark[3] a jedná sa o anglické texty zo správ a spravodajstva.

Každý korpus som rozdelil na testovacie a trénovacie dáta. Tieto dáta sú použité ako vstup pre skript, ktorý z nich vytvorí jednotlivé položky (item) a uloží ich do nového súboru. Tieto položky sa skladajú z troch častí: identifikačné číslo položky, úsek textu bez chýb čiže ground truth (label), a rovnaký úsek textu ale s chybami (sample – vzorka). Počas trénovania sa chyby väčšinou generujú na základe groud truth riadku, v niektorých experimentoch sa využili predgenerované chybné riadky priamo zo súboru položiek, no viedlo to k pretrénovaniu (overfitting). Každá z týchto troch častí je uložená v plain text súbore na novom riadku, čo umožňuje jednoduché načítanie týchto dát do polí. Úseky textu sú dlhé približne 50 znakov vrátane medzier. Podľa architektúry bol text z korpusu delený buď na úseky dlhé presne 50 znakov bez ohľadu na hranice slov, alebo na úseky o dĺžke 50-60 znakov delené vždy na hranici slov. Úseky ktoré sú takto delené na hranici slov, majú priemernú dĺžku takmer 53 znakov. Na trénovanie sa najčastejšie využili súbory obsahujúce 2 milióny položiek. Na testovanie sa využívajú súbory obsahujúce 1 000 alebo 2 000 položiek. Tieto súbory slúžia ako vstup do mojej implementácie triedy Dataset, s názvom MyDataset, ktorá z nich načíta dáta a uchová ich vo vhodnom formáte pre trénovanie a validáciu neurónovej siete. Príklad 3 položiek z jedného z validačných súborov:

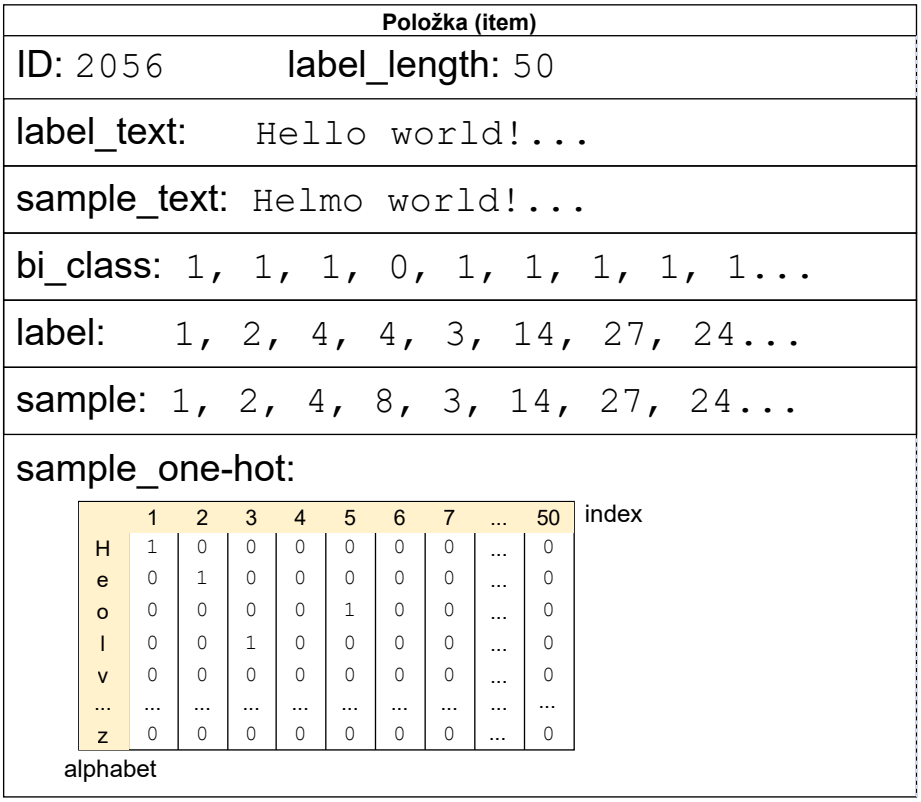
[1] https://www.kaggle.com/datasets/jannesklaas/scifi-stories-text-corpus
[2] https://dumps.wikimedia.org/
[3] https://github.com/ciprian-chelba/1-billion-word-language-modeling-benchmark

```
ID2742021
or. It was not until later that I realized that th
or. It was not nntil axer that I realizebd trat th
ID2742022
e line which separated the black from the color wa
e line which fseparaed the bnack from yhe color wa
ID2742023
s advancing slowly from my right while retreating
s advancins slowly frolumy right whila retreating
```

Zoznam znakov. Pri implementácií triedy MyDataset bolo potrebné implementovať funkciu na inicializáciu, funkciu na vrátenie jednej položky (item) z dátovej sady, a funkciu, ktorá vráti počet položiek v dátovej sade. Trieda MyDataset obsahuje tiež abecedu, teda zoznam znakov, ktoré dokáže neurónová sieť rozpoznať a pracovať s nimi. Tento zoznam sa mierne líši v závislosti na použitej architektúre, primárne obsahuje malé a veľké písmená anglickej abecedy, číslice, interpunkčné znamienka a niektoré špeciálne znaky. Pokiaľ sa využíva padding a CTC stratová funkcia, obsahuje abeceda aj padding a blank znaky. Ukážka abecedy ktorú využívajú CTC architektúry:

```
self.charlist_extra_ctc = [blank, 'A', 'B', 'C', 'D', 'E', 'F',
'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S',
'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f',
'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r','s',
't', 'u', 'v', 'w', 'x', 'y', 'z', '0', '1', '2', '3', '4', '5',
'6', '7', '8', '9', ' ', '!', '?', ',', '.','\'','"', '-',':',';',
'(', ')', '%','/', '-', '-', '"', '"', '+', '=', '[', ']',
"', '&', '*', '#', pad]
```

Uloženie dát. Neurónová sieť nedokáže pracovať priamo so znakmi, a tak je treba textové dáta správne interpretovať a uložiť. Príklad reprezentácie jednej položky v datasete je na obrázku 3.1. Trieda MyDataset teda uloží pre každú položku zo vstupného súboru položiek nasledujúce dáta: identifikačné číslo `ID` priamo prečítané zo súboru, dĺžku úseku textu bez chýb `label_length`, to je potrebné iba ak položky majú rôznu dĺžku, textový reťazec pre ground truth `label_text` aj pre chybný riadok `sample_text` prečítané priamo zo súboru, `label` a `sample` sú polia, ktoré obsahujú poradové číslo (index) každého znaku z úsekov textu v abecede datasetu, `bi_class` je pole binárnej klasifikácie, pre každý znak v chybnom texte určí, či je rovnaký, ako znak na rovnakej pozícií v ground truth, využíva sa iba pri detekcií chýb alebo architektúre kde nedochádza k posúvaniu znakov, a `sample_one-hot` je 2 rozmerné pole, ktoré pre každý znak vo vzorke (sample_text) uchováva jeho reprezentáciu ako one-hot vektor, teda vektor plný núl s jednotkou na indexe daného znaku v abecede datasetu. Práve toto 2-rozmerné pole, o šírke dĺžky chybného textu, a výške počtu znakov v abecede, je vstupom do neurónovej siete. Trieda MyDataset je vždy vytvorená, a všetky dáta sú uložené do pamäti hneď na začiatku trénovacieho alebo testovacieho skriptu. Tá je potom obalená v `torch.utils.data.DataLoader`, ktorý poskytuje podporu pre iterovanie cez dataset, batch a náhodný výber položiek.



Obr. 3.1: Príklad uloženia jednej položky v triede MyDataset.

3.2 Návrh riešenia

Chyby, ktoré v texte vznikajú sa delia na 3 typy:

- **zamenený znak** – jedná sa o zámenu jedného znaku za iný znak, pri tejto chybe nedochádza k žiadnej zmene pozícií znakov v texte ani zmene dĺžky textu, opraví sa zámenou chybného znaku za správny (pôvodný) znak,
- **vynechaný znak** – vynechanie resp. zmazanie znaku z textu spôsobí skrátenie celkovej dĺžky textu resp. posunutie textu za miestom vynechaného znaku o pozíciu doľava, opraví sa vložením chýbajúceho znaku do textu na správnu pozíciu, čo spôsobí zvýšenie dĺžky textu na pôvodnú dĺžku pred zavedením chyby resp. posunie všetky znaky za miestom chyby o pozíciu doprava,
- **nadbytočný znak** – vznikne vložením znaku do textu a spôsobí zväčšenie dĺžky textu resp. posunutie všetkých znakov za pozíciou chyby doprava, opraví sa zmazaním nadbytočného znaku z textu čo skráti dĺžku celého textu na pôvodnú hodnotu resp. posunie znaky za pozíciou chyby doľava.

Riešenie sa skladá z niekoľko pod-problémov, ktoré som postupne riešil. Navrhnuté architektúry sú uspôsobené na jeden dopredný priechod, počas ktorého vygenerujú zo vstupnej sekvencie celú výstupnú sekvenciu, aby dosiahli požadovanú rýchlosť. Najskôr som pracoval s architektúrami, ktoré sú na úrovni znakov pozične viazané 1 ku 1 medzi korektným a chybným úsekom textu. To znamená že obidva úseky textu musia mať rovnakú dĺžku a nieje

možné vkladať či mazať znaky iba z jedného úseku. Pozične viazaná architektúra prináša množstvo limitácií, pracoval som s ňou hlavne zo začiatku kvôli jej jednoduchosti a priamočiarosti. Architektúra na detekciu chýb klasifikuje každý znak vstupného úseku textu tak, že mu priradí hodnotu 1 – znak je správny, alebo 0 – znak je nesprávny. Táto architektúra využíva konvolučné a obojsmerné LSTM rekurentné vrstvy. Dokáže detegovať chyby typu zamenený znak. Chyby typu nadbytočný znak sa dajú transformovať na zámenu znakov po vložení špeciálneho znaku (#) reprezentujúceho nadbytočný znak do ground truth na pozíciu kam bol vložený znak v chybnom texte.

Rovnaká architektúra bola spočiatku použitá aj na korekciu textu, kde namiesto binárnej klasifikácie, sieť vráti rozloženie pravdepodobností cez celú abecedu, pre každú pozíciu. Aj táto sieť je stále pevne závislá na pozíciach, a teda nieje možné vkladať alebo mazať znaky bez zmeny ground truth.

Chyby typu vynechaný znak, kedy od siete vyžadujeme aby správny chýbajúci znak do sekvencie vložila, predstavujú ešte závažnejší problém. Najtriviálnejším riešením tohoto problému by bolo upraviť dáta tak, aby na výstupe sieť generovala na začiatku sekvencie, a za každým znakom miesto navyše napríklad veľkosti jedna, kam by sa generoval prípadný znak na vloženie. Toto riešenie by ale prinieslo množstvo limitácií. Sieť by síce dokázala vložiť znak kdekoľvek do výstupnej sekvencie, avšak medzi ktorékoľvek 2 susedné znaky vo vstupnej sekvencií, by bolo možné vložiť iba jeden znak. Ak by sme tento limit chceli zvýšiť, dĺžka výstupnej sekvencie narastá lineárne, a v praxi by sa využila len malá časť tohoto priestoru.

Rozhodol som sa tento problém vyriešiť pomocou CTC stratovej funkcie, ktorá je popísaná v kapitole 2.5 nakoľko by si mala byť schopná poradiť s rôznymi veľkosťami vstupnej a ground truth sekvencie, ako aj s vkladaním viacerých písmen kdekoľvek do výstupnej sekvencie. Stačí jej poskytnúť väčší priestor na generovanie výstupu, než je veľkosť vstupu, a výstup už nebude nijak závislí na pevných pozíciach zo vstupu. To nám umožní opravovať aj chyby typu vynechaný znak, či generovať rôzne počty chýb, ktoré vkladajú a mažú znaky a menia tak dĺžku vzorky. Tiež sa v tejto architektúre využíva padding na oba úseky textu, čo umožňuje zachovať rovnaké rozmery dátových štruktúr a využiť implicitné dávkovanie dát (batch), ktoré poskytuje trieda DataLoader, aj pri rôznej dĺžke jednotlivých textových vzoriek. Vzorky rôznej dĺžky zase umožňujú vzorkovať korpus podľa hraníc slov, čo eliminuje chyby na začiatku a konci úsekov textu, ktoré je v podstate nemožné správne detegovať či opraviť.

Návrh riešenia opravy textu s využitím CTC loss vychádzal z pôvodnej pozične viazanej architektúry pre korekciu. V prvej časti modelu sú konvolučné vrstvy a po nich obojsmerné LSTM vrstvy. V druhej časti modelu sa už využívajú aj de-konvolučné vrstvy, ktoré postupne alebo skokovo zväčšia rozmery dát až k výstupnej vrstve.

3.3 Generovanie chýb

Táto práca sa nezameriava konkrétne na preklepy či na chybne rozpoznané znaky po OCR, preto som nesimuloval presnejšie ani jeden z týchto prípadov tvorby chýb. Myslím si, že zatiaľ čo post-OCR texty budú obsahovať chyby, ktoré vznikli na základe optickej podobnosti znakov, použitého fontu či spôsobu vzniknutia pôvodného dokumentu (či išlo o tlačený alebo perom písaný dokument), text písaný elektronicky bude obsahovať preklepy, ktoré sú zase ovplyvnené úplne inými faktormi, ako pozícia písmen na klávesnici, použité zariadenie (kancelárska klávesnica, notebook, smartfón), jazyk či zvyky a schopnosti jednotlivých používateľov.

Generovanie chýb bolo z týchto dôvodov implementované viacerými spôsobmi, no vždy sa jednalo o dosť náhodný proces zamieňania, vkladania a mazania znakov v jednotlivých textových úsekoch. Rozhodol som sa generovať znaky pre zámenu a vkladanie len zo znakov malej anglickej abecedy, pretože si myslím, že takéto preklepy sa vyskytujú v texte najčastejšie. Malé písmená sú taktiež najzastúpenejšia skupina znakov vo využitých textových korpusoch, v porovnaní s veľkými písmenami, číslami či ostatnými znakmi. Pri trénovaní sa chyby generovali takmer vždy priamo v trénovacom skripte (online), pri testovaní sa využili predgenerované chybné riadky v testovacích súboroch (offline), pričom sa v oboch prípadoch využil rovnaký algoritmus pre danú architektúru, aby bolo rozloženie, typ a frekvencia chýb v trénovaní a testovaní rovnaká.

Algoritmus generovania chýb konštantného počtu. Pre každý textový úsek dĺžky 50 bolo vygenerovaných osem čísel rovnomerne funkciou `numppy.random.randint()`, štyri v intervale $\langle 0, 49\rangle$, ktoré reprezentujú pozície chýb, a štyri čísla rovnomerne v intervale $\langle 97, 122\rangle$, ktoré reprezentujú štyri znaky malej anglickej abecedy. Tri znaky na daných pozíciach sú nahradené troma znakmi z vygenerovaných znakov (vzniknú maximálne tri chyby typu zamenený znak), a na štvrtú vygenerovanú pozíciu je vložený štvrtý vygenerovaný znak (vznikne maximálne jedna chyba typu nadbytočný znak). Takto sa generujú chyby v súboroch s príponou `...typos.txt` Priemerná editačná vzdialenosť medzi ground truth a chybným riadkom v týchto súboroch je 3,80.

Algoritmus generovania chýb náhodného počtu. Pre každý textový úsek dĺžky d bolo vygenerované číslo x normálnym (Gaussovím) rozložením funkciou `numppy.random.normal()` so strednou hodnotou c, smerodajnou odchýlkou s, zaokrúhlené funkciou `numpy.round()`, a ohraničené na hodnotu z intervalu $\langle 0, 8\rangle$, reprezentujúce počet chýb v úseku textu, x čísel v intervale $\langle 97, 122\rangle$ vygenerovaných rovnomerne funkciou `numppy.random.randint()`, ktoré reprezentujú znaky malej abecedy, x čísel rovnomerne podľa dĺžky textového úseku, ktoré reprezentujú pozície chýb, typ chyby je pre každú chybu vygenerovaný funkciou `numpy.random.choice()` ako výber z troch možností (zamenený znak, nadbytočný znak, vynechaný znak). Takto sa generujú chyby v súboroch s príponou `...typosRF.txt`. Číslo za RF v názve súboru značí druhy chýb ktoré obsahuje (1 – zámeny, 2 – zámeny, nadbytočné znaky, 3 – zámeny, nadbytočné znaky, vynechané znaky). Ak je v názve za RF zátvorka, súbor obsahuje všetky tri typy chýb, a hodnoty v zátvorke značia strednú hodnotu c a smerodajnú odchýlku s pri generovaní počtu chýb. V experimentoch používam ako hodnoty dvojice c a s tieto dvojice: (6,2), (5,2), (4,3), a (2,1).

3.4 Architektúry pre detekciu chýb

Ako prvé som sa rozhodol implementovať neurónovú sieť, ktorá deteguje chyby vo vstupnom texte. Úseky textu majú konštantnú dĺžku 50 znakov. Sieť pre každý znak vo vstupnej vzorke textu, ktorá obsahuje chyby vyhodnotí, či je znak správny alebo nie. Sieť teda vráti pole binárnej klasifikácie dĺžky 50, kde hodnota 1 znamená korektný znak, a hodnota 0 znamená chybný znak. Limitácie tohoto riešenia sú zjavné – sieť dokáže detegovať len chyby typu zamenený znak, prípadne transformovať chybu typu nadbytočný znak na zámenu ako bolo popísané vyššie. Abeceda, nad ktorou pracuje táto architektúra má dĺžku 69 znakov, obsahuje malé a veľké písmená anglickej abecedy, číslice, základné interpunkčné znaky, znak pre reprezentáciu nadbytočného znaku (#) a znak, ktorým sú nahradené všetky ostatné

Názov súboru	D	Typy chýb	ED
wiki_test_test_1k_typos.txt	50	1, 2	3,79
wiki_test_test_1k_typos2.txt	50	1, 2	3,81
scifi_test_test_1k_typos.txt	50	1, 2	3,81
scifi_test_test_1k_typos_2M.txt	50	1, 2	3,79
wiki_test_test_1k_typosRF2.txt	50	1, 2	4,57
scifi_test_test_1k_typosRF2.txt	50	1, 2	4,68
news_test_RLOAWP2_2k_typosRF(2,1)_CTC.txt	52,83	1, 2, 3	1,94
news_test_RLOAWP2_2k_typosRF(4,3)_CTC.txt	52,88	1, 2, 3	3,79
news_test_RLOAWP2_2k_typosRF(6,2)_CTC.txt	52,84	1, 2, 3	5,57

Tabuľka 3.1: Tabuľka pred-generovaných validačných súborov položiek obsahujúcich ID, vzorku textu a ground truth. Počet položiek v súbore je v názve súboru 1k – 1 000 položiek, 2k – 2 000 položiek, stĺpec D udáva priemernú dĺžku úseku textu jedného vzorku, typy chýb: 1 – zamenený znak, 2 – nadbytočný znak, 3 – vynechaný znak, stĺpec ED udáva priemernú editačnú vzdialenosť medzi ground truth (úsek bez chýb) a úsekom textu s chybami.

znaky, ktoré niesu v abecede zahrnuté (§)(v niektorých starších architektúrach sa tieto znaky nahrádzali medzerou, a v novších sa zasa odstránili z korpusu textu pri tvorbe dátovej sady). Celá abeceda:

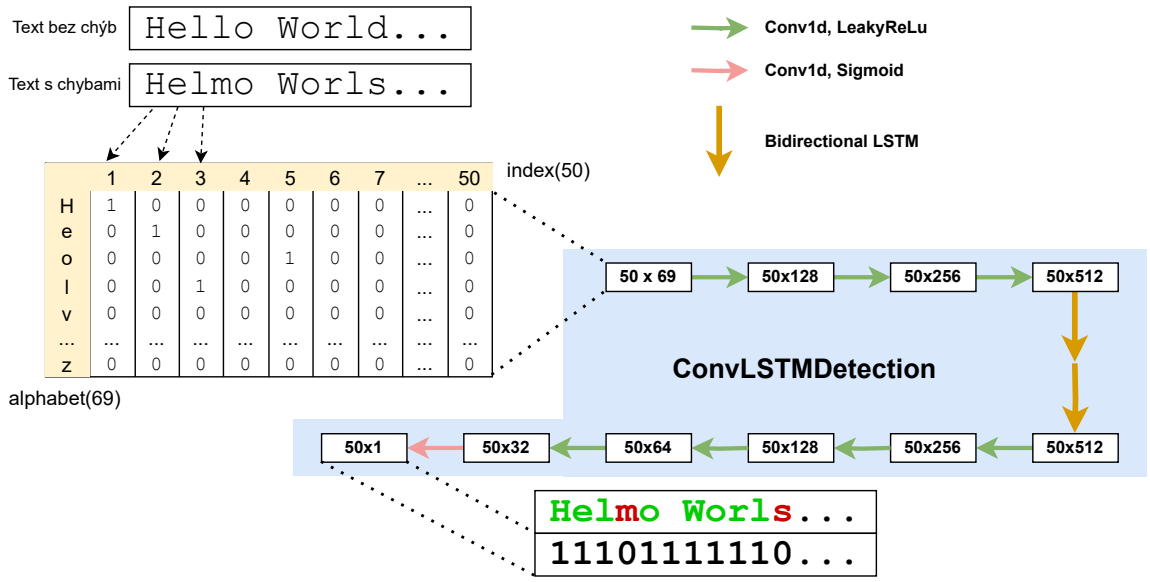
```
self.charlist_base = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J',
'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',
'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l',
'm', 'n', 'o', 'p', 'q', 'r','s', 't', 'u', 'v', 'w', 'x', 'y', 'z',
'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', ' ', '!', '?', ',',
'§','.', '#']
```

Architektúru som nazval ConvLSTMDetection, a jej schéma je na obrázku 3.2. Jedná sa o konvolučnú architektúru, ktorá využíva 2 obojsmerné LSTM vrstvy v strede. Po vstupnej vrstve nasledujú 3 konvolučné vrstvy, ktoré postupne zväčšujú počet kanálov (channels) až na počet 512. Keďže je k dispozícii celá sekvencia, môžeme využiť obojsmerné LSTM vrstvy v najširšom bode architektúry. Po 2 LSTM vrstvách nasleduje 5 konvolučných, ktoré redukujú počet kanálov až na 1, takže dosiahneme rozmer výstupu 50×1, jedna hodnota pravdepodobnosti pre každý znak textu. Konvolučné vrstvy majú veľkosti jadra (kernel) 9, aby zachytili dostatočne veľký kontext v okolí každého znaku. Posledné 2 vrstvy majú veľkosť jadra 5, padding je vždy nastavený tak, aby nedošlo k zmene dĺžky dát. Za každou konvolučnou vrstvou nasleduje batch normalizačná vrstva, ktorá optimalizuje model pri trénovaní, a aktivačná Leaky Rectified Linear Unit (LeakyReLu) funkcia, ktorá má v PyTorch prednastavenú hodnotu negative_slope 0.01. Za poslednou výstupnou vrstvou je namiesto LeakyReLu aktivačná Sigmoid funkcia, nakoľko chceme generovať hodnoty medzi 0 a 1.

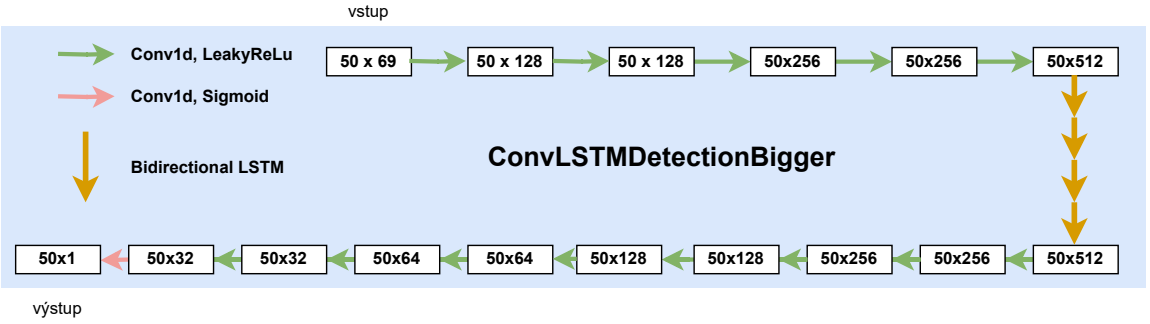
Ako stratová funkcia pri trénovaní sa využíva Binary Cross Entropy Loss, ktorá je v PyTorch popísaná touto rovnicou:

$$l(x, y) = L = \{l_1, l_2, \dots l_n\}, \; l_n = -w_n[y_n \times \log x_n + (1 - y_n) \times \log(1 - x_n)], \tag{3.1}$$

kde x sú hodnoty predikované sieťou, y je ground truth v mojom prípade `bi_class` z datasetu, w_n je váha konkrétnej položky z datasetu (vždy 1). Výstupom zo siete je hodnota



Obr. 3.2: Architektúra ConvLSTMDetection spolu so vstupom a výstupom



Obr. 3.3: Architektúra ConvLSTMDetectionBigger

v intervale $\langle 0, 1\rangle$ pre každý znak, ktorá je v prípade inferencie a testovania upravená na hodnoty 0 alebo 1 použitím jednoduchej prahovej (threshold) funkcie:

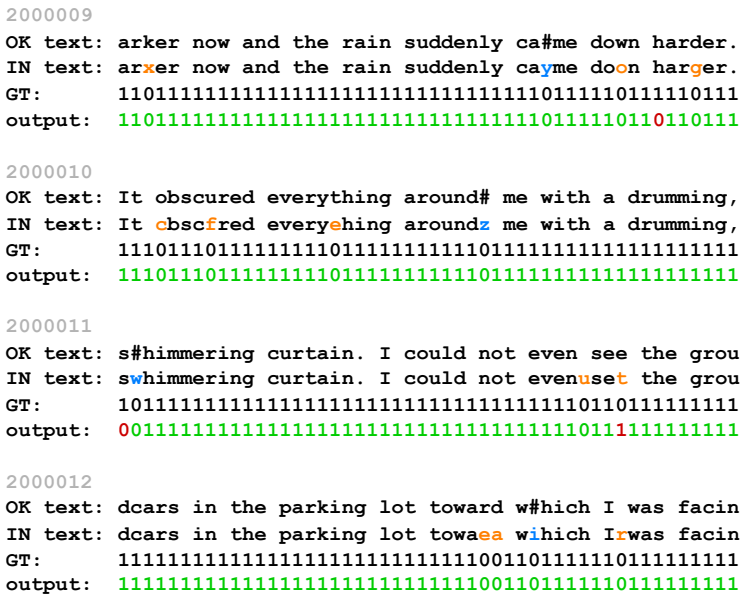
$$\mathrm{F}(x) = \begin{cases} 1 & \text{ak } x \geq 0,6 \\ 0 & \text{inak.} \end{cases} \tag{3.2}$$

Príklad výstupu z testovania tejto siete je na obrázku 3.4.

Implementovaná bola aj druhá verzia tejto architektúry ConvLSTMDetectionBigger, zobrazená na obrázku 3.3, ktorej princíp je veľmi podobný pôvodnej verzií, ale využíva viac konvolučných a LSTM vrstiev. Konkrétne 5 konvolučných vrstiev v prvej časti, 4 obojsmerné LSTM vrstvy v strede a 9 konvolučných vrstiev v druhej časti.

3.5 Architektúry pre opravu chýb

Pri implementovaní architektúr pre korekciu chýb som vychádzal z architektúry pre detekciu. Rozdielom je hlavne zmena z binárnej klasifikácie na klasifikáciu cez celú abecedu pre



Obr. 3.4: Príklad výstupu architektúry ConvLSTMDetection pre 4 vzorky z testovacieho súboru `scifi_test_test_1k_typos_2M.txt`. Prvý riadok je ID položky, *OK text* je korektný text (label_text), *IN text* je vstupný text s chybami (sample_text), *GT* je ground truth resp. hodnota `bi_class` z datasetu a riadok *output* je výstup z neurónovej siete upravený threshold funkciou 3.2.

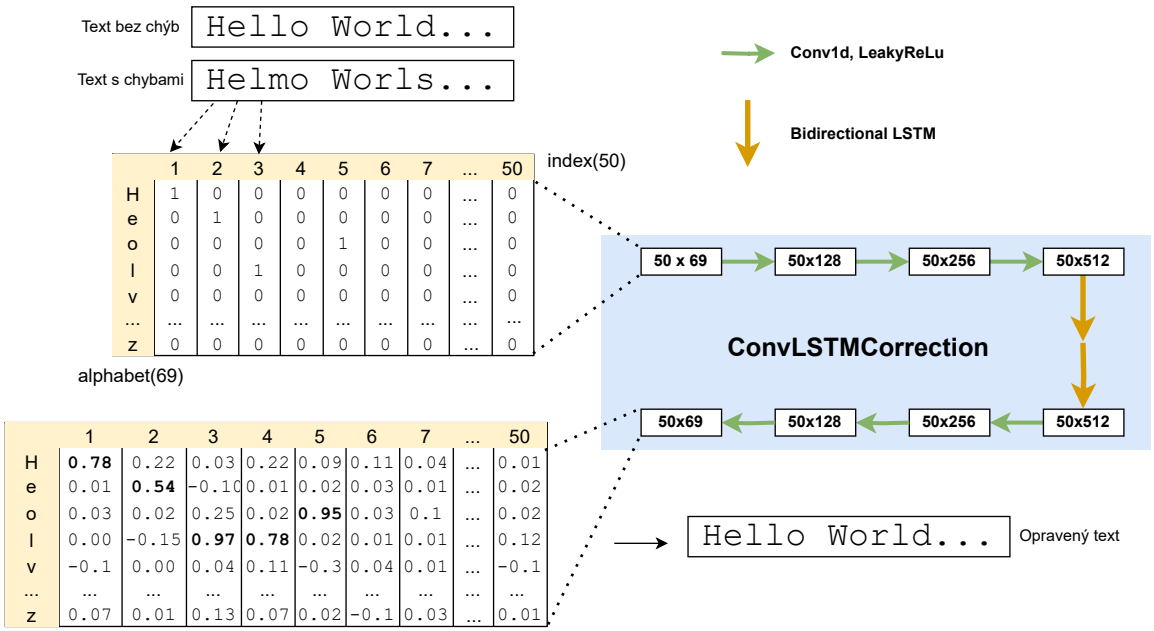
každý znak. Vstupné dáta do nasledujúcich architektúr sa nezmenili, stále ide o one-hot vektory jednotlivých znakov textu s chybami. Sieť len na základe tohoto textu generuje výsledný opravený text. Ground truth je korektný text, resp. jeho reprezentácia ako indexy z abecedy, na čo sa môžme pozerať aj ako čísla tried pre jednotlivé pozície v texte. Výstupom z architektúr je pravdepodobnostná matica, ktorej prvý rozmer sú jednotlivé pozície znakov v texte, a druhý abeceda.

Pozične viazané architektúry. Pozične viazané architekúry pre korekciu využívajú ako stratovú funkciu Cross Entropy Loss, ktorá je v PyTorch implementovaná rovnicou:

$$l(x,y) = L = \{l_1, l_2, \dots l_n\},\ l_n = -w_{yn} \times \log \frac{e^{x_{n,yn}}}{\sum_{C=1}^{C} e^{x_{n,C}}}, \tag{3.3}$$

kde x je výstup zo siete, y je ground truth vo formáte indexov tried v rozsahu $[0..C)$, C je počet tried (v našom prípade počet znakov v abecede), w je váha triedy (vždy 1), n je index položky v batch. Táto loss funkcia je ekvivalentná s využitím kombinácie Log Softmax funkcie a Negative Log Likelyhood stratovej funkcie, preto ani nieje vyžadované, aby výstupná matica z neurónovej siete obsahovala iba hodnoty z intervalu $(0, 1)$, či aby súčet týchto hodnôt pre každú pozíciu v texte bol 1.

Architektúra ConvLSTMCorrection na obrázku 3.5 využíva 3 konvolučné vrstvy, ktoré postupne rozširujú svoj výstup a extraktujú vlastnosti (features) z textu. Nasleduje rekurentná časť s dvomi obojsmernými LSTM vrstvami, ktorých výstup je zase postupne zmenšený tromi konvolučnými vrstvami na potrebné rozmery. Výsledná matica pri trénovaní je normalizovaná až v samotnej stratovej funkcií 3.3. Každý stĺpec výslednej matice



Obr. 3.5: Architektúra ConvLSTMCorrection so vstupom a výstupom. Hodnoty vo výstupnej matici niesu normalizované pretože sa nevyužíva LogSoftMax funkcia, ktorá je súčasťou stratovej funkcie.

predstavuje pozíciu vo výslednom texte. Jednotlivé prvky v každom stĺpci predstavujú pravdepodobnosti pre jednotlivé znaky z abecedy, že sa majú nachádzať na danej pozícií. Pri validácií/inference modelu je z každého stĺpca vybraná maximálna hodnota ako znak, ktorý bude vo výslednom texte. Abeceda je rovnaká ako abeceda pre detekciu chýb v kapitole 3.4.

Podobne ako v prípade detekcie, bola aj v prípade korekcie implementovaná väčšia verzia tejto architektúry ConvLSTMCorrectionBigger. Tá pridáva do oboch častí pôvodnej architektúry 2 konvolučné vrstvy, ktoré nemenia počet kanálov. Väčšia architektúra dosiahla v experimentoch lepšie výsledky. Príklad výstupu z testovania tejto architektúry je na obrázku 3.6.

Architektúry s CTC. Zavedením chýb typu nadbytočný znak a vynechaný znak, sa úloha pre neurónovú sieť značne komplikuje. Dochádza k posúvaniu rôznych častí textu a zmene jeho dĺžky, pričom sú stále prítomné aj chyby typu vymenený znak. Celkovo boli implementované 3 architektúry využívajúce CTC stratovú funkciu, ktorej princíp je vysvetlený v kapitole 2.5. Tieto architektúry sú trénované a testované na vylepšenej verzií dátovej sady, ktorá využíva padding, čo umožňuje v jednom vzorku generovať rôzny počet chýb, ktoré menia dĺžku reťazca (vkladanie a mazanie) a deliť text podľa hraníc slov. Padding je priamo v textovej podobe v súboroch s položkami. V chybnom riadku sa jedná o 3 znaky na začiatku vzorky, nasleduje textový úsek dlhý 50 až 60 znakov a ďalší padding až po pozíciu 73. Toto zabezpečí, že aj keby sa pri generovaní chýb vygeneruje maximálny počet chýb – 8, a všetky by boli typu nadbytočný znak, dĺžka vzorky bez paddingu na konci nikdy nepresiahne hodnotu 73. Motiváciou pre využitie CTC stratovej funkcie a paddingu v textových vzorkách je vyriešenie týchto problémov:

- Oprava chýb typu nadbytočný znak a vynechaný znak – upustenie od 1:1 pozičnej viazanosti medzi vstupnou vzorkou a ground truth textom (label).

```
2000033
OK text: ns on St. Marie.  He gaze#d at me.  Didn t you kno
IN text: ns on St. iarif.  He gazeid at me.  Didn t you kno
output:  ns on St. Maris.  He gaze#d at me.  Didn t you kno

2000034
OK text:  you were risking your life, coming he#re today?
IN text:  you werc risking your lifv, coming hegre today?
output:   you were risking your life, coming he#re today?

2000035
OK text:  opened my mo#uth to deny it. Then I realized I ha
IN text:  opened my mokuthkto deny it. Then I realizedsI hl
output:   opened my mo#uth to deny it. Then I realized I ha

2000036
OK text:  known.  What if someone shou#ld call out to them,
IN text:  known.  What if someone shouwld cpll out xosthem,
output:   known.  What if someone shou#ld call out to them,
```

Obr. 3.6: Príklad výstupu architektúry ConvLSTMCorrectionBigger pre 4 vzorky z testovacieho súboru `scifi_test_test_1k_typos_2M.txt`. Prvý riadok je ID položky, *OK text* je korektný text zároveň aj ground truth (label_text), *IN text* je vstupný text s chybami (sample_text), a riadok *output* je výstup z neurónovej siete resp. pre každú pozíciu je vybraný znak z abecedy s najvyššou hodnotou.

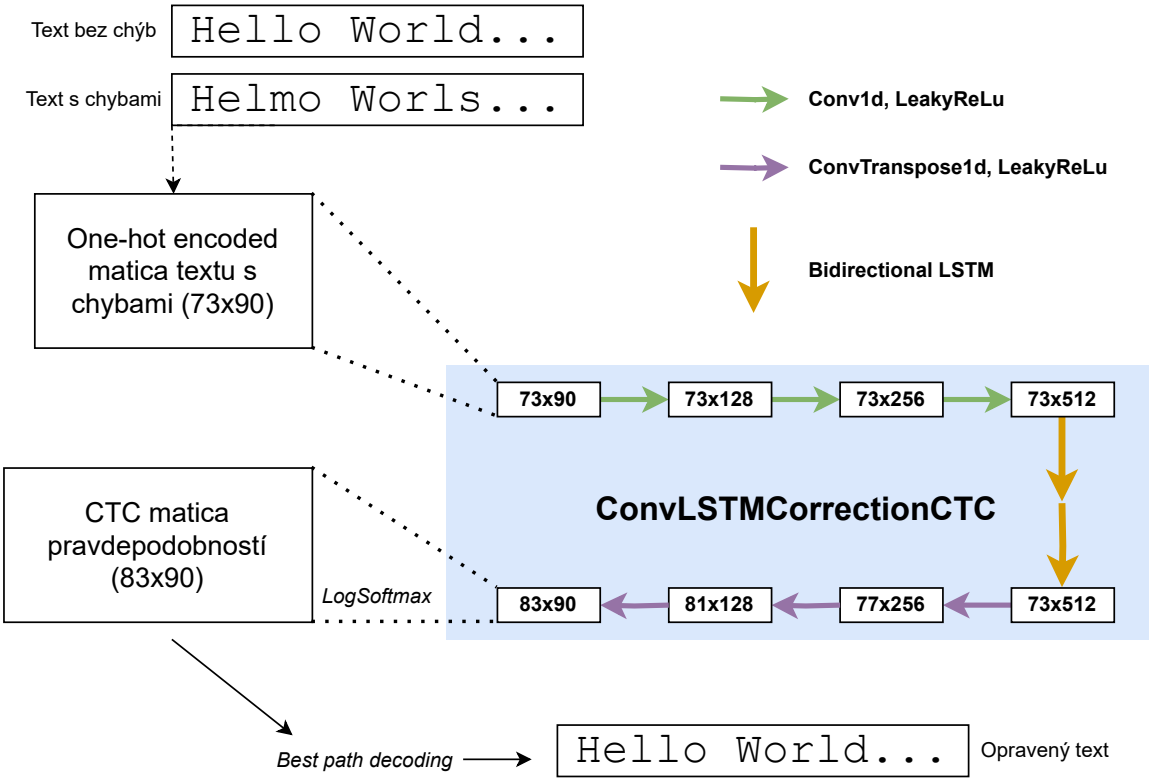
- Vzorky začínajú a končia na hranici slov – nevznikajú falošné chyby kvôli useknutým slovám na začiatku a konci vzoriek.

Všetky 3 implementované architektúry využívajú rozšírenú abecedu obsahujúcu 90 znakov ukázanú v sekcií 3.1. Padding a blank znaky sú znaky z ruskej azbuky, ktoré sa nevyskytujú v textovom korpuse.

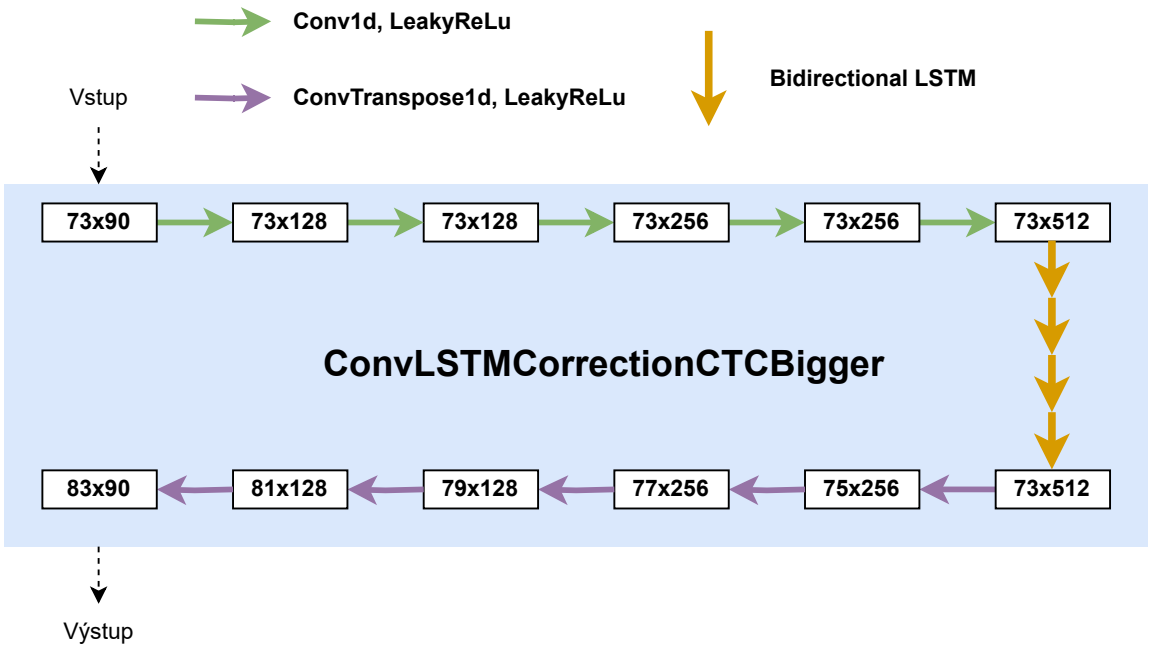
Prvá architektúra s názvom ConvLSTMCorrectionCTC má prvú konvolučnú časť podobnú doteraz predstaveným architektúram – 3 konvolučné vrstvy s LeakyReLu aktivačnou funkciou, ktoré postupne zvyšujú počet kanálov. Rekuretná časť obsahuje 2 obojsmerné LSTM vrstvy. Rozdiel je v druhej konvolučnej časti, kde sú namiesto obyčajných konvolúcií využité transponované konvolučné vrstvy. Tieto vrstvy sú použité pretože je nutné zväčšiť dĺžku výslednej sekvencie aby sme poskytli priestor potrebný na vloženie znakov pri opravovaní chýb. V tejto architektúre je pôvodná dĺžka vstupu s paddingom 73 a je postupne rozšírená na 83. Priemerná dĺžka ground truth sekvencií je 53 znakov. Schéma architektúry je zobrazená na obrázku 3.7. Na výstup zo siete je aplikovaná funkcia

$$\text{LogSoftmax}(x_i) = log\left(\frac{e^{x_i}}{\sum_j e^{x_j}}\right), \tag{3.4}$$

ktorá ho transformuje na rozloženie pravdepodobností naprieč abecedou.



Obr. 3.7: Architektúra ConvLSTMCorrectionCTC.



Obr. 3.8: Schéma modelu ConvLSTMCorrectionCTCBigger

```
2000142
OK text: devices . International observers said the vote largely
IN text: dvices . Intrnatisnal observers said the vonst largeny
RAW: aaaadevvicees  . Inteernaationall  obbservverss saiid the  coasst  larrgellyyaaaa
output:  devices . International observers said the cost largely

2000143
OK text: met their standards . Some reports said the bomber
IN text: met thneuir standawrds . Some reports sai the bomber
RAW: aaaameatt  theeiir  sstanndarrdss  . SSome  reeporttss saaid  thee  bommberraaaaaa
output:  met their standards . Some reports said the bomber

2000144
OK text: targeted a security check point in the area , while
IN text: targeped  securiuty chgeck oint wn the area , whle
RAW: aaaataarggetted  a  seccuritty  cchecck  poinnt  in  the  areea  ,  whileeaaaaaa
output:  targeted a security check point in the area , while

2000145
OK text: other said the explosion took place near an army school
IN text: other said the explosion tolk plce near an army scvhool
RAW: aaaaottherr  said  the  expplosiion  tooaok  pllace  nearr ann aarmmy sschoaaolaaa
output:  other said the explosion took place near an army school
```

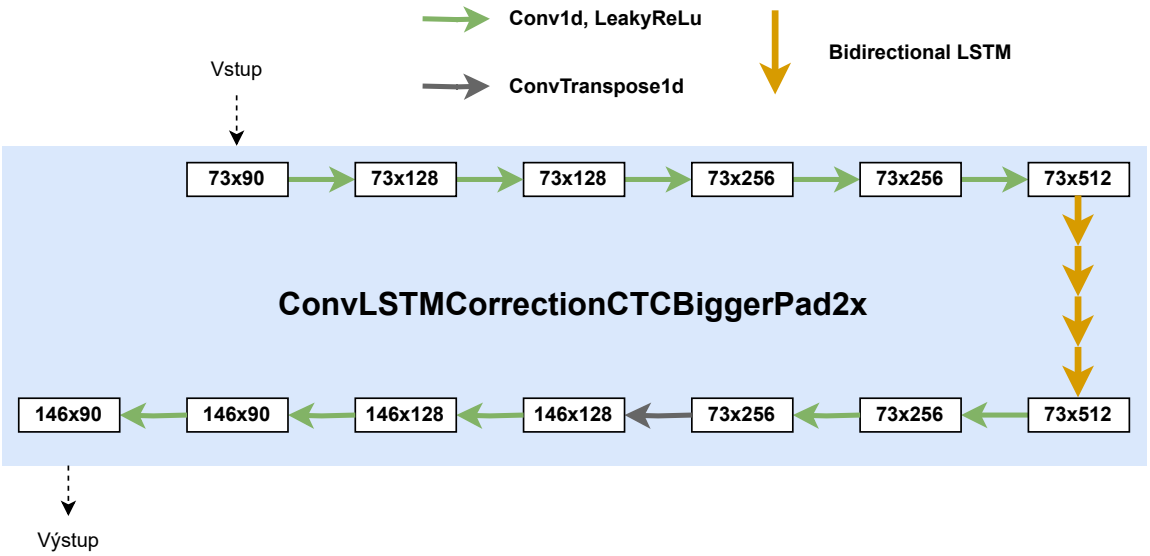
Obr. 3.9: Príklad výstupu z testovania architektúry ConvLSTMCorrectionCTCBigger. Riadok RAW zobrazuje výstup z neurónovej siete pred tým, ako sú odstránené výskyty rovnakých znakov za sebou a znak blank.

Druhá architektúra s názvom ConvLSTMCorrectionCTCBigger je opäť zväčšená verzia pôvodnej, využíva rovnaké rozmery vstupu a výstupu aj tú istú stratovú funkciu. Rozdielom je pridanie 2 konvolučných vrstiev do 1. konvolučnej časti, 2 obojsmerných LSTM vrstiev do rekurentnej časti a 2 vrstiev transponovanej konvolúcie do 2. konvolučnej časti. Jej schéma je na obrázku 3.8. Príklad výstupu z testovania tejto architektúry je na obrázku 3.9.

Posledný natrénovaný model ConvLSTMCorrectionCTCBiggerPad2x je úpravou druhého modelu, a líši sa v druhej konvolučnej časti. Schéma je na obrázku 3.10. Miesto postupného rozširovania dĺžky výstupu piatimi vrstvami transponovanej konvolúcie, vymieňa tieto vrstvy za klasickú konvolúciu, ktorá nemení dĺžku výstupu, ale po dvoch konvolučných vrstvách je vložená jedna vrstva transponovanej konvolúcie, ktorá zdvojnásobí dĺžku výstupu. To je docielené nastavením parametrov šírky jadra na 2 a hodnoty stride na 2. Konkrétne má vstup do siete dĺžku 73 a výstup z nej je dlhý 146 znakov. Tento väčší výstup by mal umožniť architektúre jednoduchšie vkladanie znakov do výstupu.

3.6 Trénovanie a trénovací skript

Trénovanie architektúr je implementované v troch súboroch podľa typu architektúry (`training_detection.py`, `training_correction.py`, `training_CTC.py`). Pomocou argumentov príkazovej riadky je možné nastaviť rôzne parametre trénovania. Dá sa takto upraviť veľkosť batch, maximálny počet iterácií, učiaci koeficient (learning rate), hodnoty na úpravu učiaceho koeficientu, čiže váha, ktorou je možné ho násobiť a počet iterácií, po ktorých dôjde k násobeniu, či sa budú chyby generovať počas trénovania alebo sa využijú chybné riadky z trénovacieho súboru, stredná hodnota a smerodajná odchýlka, podľa ktorých sa bude generovať počet chýb počas trénovania, predtrénovaný model ak chceme pokračovať



Obr. 3.10: Schéma modelu ConvLSTMCorrectionCTCBiggerPad2x

v jeho trénovaní, súbor, do ktorého bude natrénovaný model uložený, trénovací, validačný a testovací súbor. Ostaté parametre ako napríklad triedu modelu resp. výber architektúry, dĺžku úsekov textu, stratovú funkciu či optimalizátor je potrebné zmeniť upravením skriptu.

Trénovací vstupný súbor obsahujúci jednotlivé vzorky mal zvyčajne veľkosť 2 000 000 položiek. Testovací a validačný súbor mali veľkosti 1 000 alebo 2 000 položiek. Validačný súbor obsahoval rovnaké vzorky ako trénovací, testovací obsahoval vzorky z testovacej časti rovnakého korpusu. Vždy bol využitý optimalizátor Adam, a veľkosť jednej batch bola 50 položiek. Prejsť celým trénovacím súborom tak trvalo 40 000 iterácií. Z časových dôvodou sa trénovalo po etapách (jedna etapa znamená jedno spustenie trénovacieho skriptu) trvajúcich 600 000 iterácií. Za využitia výpočetných zdrojov najmä výkonných grafických kariet, ktoré mi poskytlo MetaCentrum[4], trvala jedna etapa osem až dvadsať hodín v závislosti na architektúre, po koľkých iteráciach prebiehala validácia a pod. Natrénovanie jednej architektúry trvalo približne 1 800 000 iterácií resp. 3 etapy.

Trénovací skript rozpozná, či je k dispozícii grafická karta s podporou technológie CUDA[5], a ak áno, presunie model neurónovej siete na grafickú kartu, čo môže podľa jej výkonu výrazne zrýchliť trénovanie. Následne sa v trénovacom skripte vytvorí trieda `MyDataset` a `DataLoader` pre trénovací, validačný aj testovací súbor. Nasleduje vytvorenie triedy modelu z importovaného súboru `models.py` a prípadné načítanie predtrénovaných parametrov modelu, ak bol zvolený argument `-load_model`. Za zmienku ešte stojí priradenie stratovej funkcie do premennej `loss_fn` a optimalizéra `torch.optim.Adam()` do premennej `optimizer`. Potom nasleduje implementácia funkcií na generovanie chýb počas trénovania, staršia `add_typos(item)` funkcia generuje konštantný počet chýb a `add_typos_RF(item)`, ktorá generuje „náhodný“ počet chýb. Generovanie chýb je podrobnejšie popísané v kapitole 3.3.

Nasleduje funkcia `train`, v ktorej sa nachádza samotný trénovací cyklus. V ňom sa každú iteráciu získa z triedy `data_loader` dávka (batch) trénovacích položiek do premennej

[4] https://metavo.metacentrum.cz/
[5] https://developer.nvidia.com/cuda-downloads

`item`. Ak je argument `online` nastavený na 1, dôjde ku generovaniu chýb pre dané vzorky. Následne dôjde k poslaniu vstupu teda one-hot matice textu s chybami do neurónovej siete a získaniu výstupu. Z neho a groud truth je vypočítaná hodnota stratovej funkcie, nad ktorou sú metódou `loss.backward()` vypočítané gradienty vzhľadom na všetky parametre modelu a uložené v `parameter.grad`. Funkcia `optimizer.step` následne iteruje cez tieto parametre modelu, a podľa gradientov mení ich hodnoty. Každých niekoľko sto alebo tisíc iterácií môže dochádzať k vypisovaniu hodnoty stratovej funkcie a učiaceho koeficientu, zníženiu učiaceho koeficientu či testovaniu nad validačnými a testovacími dátami.

Ako posledná je v trénovacom skripte implementovaná funkcia `test(data_loader)` na testovanie, ktorej parametrom je trieda `data_loader` vytvorená nad validačným alebo testovacím datasetom. Pred volaním tejto funkcie je potrebné model prepnúť do testovacieho režimu pomocou funkcie `model.eval()`, a pred pokračovaním trénovania ho vrátiť do trénovacieho režimu zavolaním `model.train()`. Vo funkcií `test(data_loader)` teda prebieha testovanie modelu, ako aj počítanie rôznych štatistík a vypisovanie príkladov jednotlivých vstupov a výstupov.

3.7 Validačné a ostatné skripty

Zoznam a popis ostatných skriptov ktoré sú súčasťou riešenia:

- `models.py` – súbor obsahuje definície tried jednotlivých architektúr založených na základnej triede `torch.nn.Module`.
- `dataset.py` – implementuje triedu `MyDataset`, ktorá zo vstupného súboru textových úsekov popísaného v kapitole 3.1 vytvorí a uloží dáta vo formáte popísanom v kapitole 3.1.
- `process_corups.py` – tento skript obsahuje funkcie pre spracovanie korpusu a vytváranie súborov s textovými úsekmi. Funkcia `process_corpus(file:str)` vytvorí z textového súboru nový, kde text bude rozdelený na úseky konštantnej alebo náhodnej dĺžky, každý úsek má ID a 2 riadky (prvý ground truth, druhý na vloženie chýb) rovnakého textu. Funkcia `process_corpus_split_by_word(file:str)` robí podobnú činnosť ale delí text na hraniciach slov. Funkcia `pad(file:str)` pridá padding do súboru s úsekmi textu, funkcia `filter_non_alpha(file:str)` vytvorí nový súbor, ktorý nebude obsahovať znaky, ktoré niesu v premennej v tejto funkcií, vhodné ju aplikovať priamo na korpus pred jeho rozdelením na úseky. Skript tiež obsahuje viacero funkcií na generovanie chýb do súborov s úsekmi textu.
- `inference.py` – skript na testovanie, ktorý do súboru zaznamená jednotlivé vstupy, groud truth a výstupy z neurónovej siete v textovom formáte, pre počítanie štatistík.
- `eval.py` – skript na počítanie štatistík z výstupu `inference.py`, prípadne štatistík trénovacích a testovacích súborov s úsekmi textu.
- `ansi_print.py` – obsahuje funkciu `a_print(text, truth, corr_color, err_color)`, ktorá využíva ansi kódovanie farieb[6] v terminály pre farebný výpis textu do terminálu, využiteľné hlavne pri pozične viazaných riešeniach.

[6] https://ss64.com/nt/syntax-ansi.html

Kapitola 4

Testovanie a experimenty

V tejto kapitole definujem využitú metriku pri testovaní, ako aj spôsob počítania množstva chýb v texte, popíšem vykonané experimenty a celkové výsledky natrénovaných architektúr.

4.1 Absolútna presnosť

Najzákladnejšou metrikou, ktorú je možné vypočítať pri detekcií aj korekcií textu je **absolútna presnosť** (absolute accuracy – AAC). Táto presnosť sa vypočíta vzťahom:

$$AAC = \frac{corrected - created}{all} \tag{4.1}$$

kde *correcred* je počet správne opravených alebo detegovaných chýb, *created* je počet vytvorených chýb alebo správnych znakov označených ako chyby, *all* je pôvodný počet chýb vo vstupnom texte. 100-percentnú presnosť by teda dosiahla architektúra, ktorá by opravila resp. detegovala všetky pôvodné chyby v texte, a pritom nezaviedla do neho žiadne nové chyby resp. neoznačila žiadne správne znaky ako chybné.

V pozične viazaných architektúrach je možné chyby a štatistiky počítať triviálne postupným porovnávaním znakov medzi dvoma reťazcami. V architektúrach využívajúcich CTC ale aj v pozične viazaných architektúrach na opravu textu je výhodné použiť výpočet Levenštejnovej editačnej vzdialenosti medzi dvoma reťazcami. Táto hodnota udáva najmenší počet zmien znakov (zámen, vložení a zmazaní), ktoré treba aplikovať na jeden reťazec, aby sa rovnal druhému reťazcu. Na výpočet tejto hodnoty využívam funkcie z python knižnice Levenshtein. Táto hodnota sa nemusí rovnať skutočnému počtu chýb, ktorý bol vygenerovaný, ten však nezistíme ani iným spôsobom keďže sa chyby môžu generovať na rovnakých pozíciach, zmazanie znaku môže zmazať nadbytočný znak, či zámena môže vygenerovať rovnaký znak ako bol ten pôvodný. Výpočtom editačnej vzdialenosti tak môžeme jednoducho určiť počet chýb resp. „chybovosť“ jednotlivých testovacích a trénovacích súborov, ako aj výstupov z neurónovej siete voči ground truth.

4.2 Experimenty

V tejto časti sú popísané vykonané experimenty nad predstavenými architektúrami, ktoré sa sústredia na rôzne aspekty riešenia. Typicky je porovnávaná absolútna presnosť pretože je to konzistentná metrika naprieč detekciou, pozične viazanou korekciou aj korekciou využívajúcou CTC.

Experiment 1: Vplyv počtu trénovacích položiek pri detekcií chýb. V prvom experimente sa snažím overiť jednoduchý predpoklad, že architektúra pre detekciu s vyšším počtom položiek v trénovacom súbore, dosiahne lepšie výsledky. Testovaná je architektúra ConvLSTMDetection na datasete SciFi Stories Text Corpus (scifi) v prípadoch, kedy trénovací súbor obsahoval 500 000 položiek a 2 000 000 položiek.

Architektúra	**Trénovací súbor**	**Testovací súbor**	**AAC**
ConvLSTMDetection	scifi_500k	scifi_test_test_1k_typos_2M	77,63%
ConvLSTMDetection	scifi_2M	scifi_test_test_1k_typos_2M	79,32%

Tabuľka 4.1: Tabuľka porovnáva rozdiel v absolútnej presnosti architektúry ConvLSTMDetection pri trénovaní na 500 000 a 2 000 000 položkách.

Výsledky experimentu potvrdzujú predpoklad, že architektúra pre detekciu chýb trénovaná na väčšom počte vzoriek dosahuje lepšie výsledky, viď tabuľka 4.1. Bol použitý algoritmus generovania chýb konštantného počtu, ktorý je popísaný v sekcií 3.3.

Experiment 2: Pozične viazaná oprava chýb a 2 rôzne datasety Druhý experiment som vykonal nad pozične viazanou architektúrou ConvLSTMCorrectionBigger. Zaujímal ma rozdiel pri použití iného textového korpusu. Predpokladal som mierne horšie výsledky pri využití korpusu z wikipedie, pretože tento text obsahuje veľké množstvo mien, dátumov, odborných výrazov a cudzích slov či pozostatky tabuliek a zoznamov z wikipédie, ktoré nijak neuľahčujú neurónovej sieti naučiť sa kontext alebo nejaké súvislosti medzi slovami. Sci-fi korpus zase obsahuje množstvo priamej reči či vymyslené mená pre postavy, rasy a miesta.

Architektúra	**TS**	**Testovací súbor**	**AAC**
ConvLSTMCorrectionBigger	wiki_2M	wiki_test_test_1k_typos_RF2	72,87%
ConvLSTMCorrectionBigger	scifi_2M	scifi_test_test_1k_typos_RF2	74,94%

Tabuľka 4.2: Tabulka porovnáva výsledky architektury ConvLSTMCorrectionBigger trénovanej na 2 rôznych datasetoch. Stĺpec TS udáva trénovací súbor.

Ako je možné vidieť v tabuľke 4.2, model trénovaný a testovaný na datasete z wikipédie má naozaj horšie výsledky ako model využívajúci dataset scifi. Pripisujem tento rozdiel zložitejšej štruktúre textu v korpuse získaného z anglickej wikipédie. Bol použitý algoritmus generovania chýb náhodného počtu s parametrami (5,2), ktorý je popísaný v sekcií 3.3.

Experiment 3: Porovnanie offline a online generovania chýb v architektúre s CTC. Pri online generovaní chýb, sa chyby v trénovacích úsekoch textu generovali priamo v trénovacom skripte (online), a teda keď sieť spracovávala tú istú vzorku dát viac krát, boli v nej zakaždým iné chyby, a sieť sa nemohla „naučiť“ konkrétne chyby v konkrétnych úsekoch textu. Pri offline generovaní, chýb sú chyby predgenerované v súboroch s úsekmi textu, čo by znížilo potrebnú réžiu pri trénovaní. Cieľom tohoto experimentu bolo zistiť rozdiely v kvalite výsledkov sietí trénovaných oboma spôsobmi.

Architektúra	Počet položiek(korpus)	Chyby	AAC
ConvLSTMCorrectionCTCBiggerPad	2 000 000(scifi)	online	61,78%
ConvLSTMCorrectionCTCBiggerPad	2 000 000(scifi)	offline	52,21%

Tabuľka 4.3: Tabuľka porovnáva absolútnu presnosť (AAC) architektúry ConvLSTMCorrectionCTCBiggerPad pri generovaní chýb online a offline. Experiment bol vykonaný nad scifi datasetom. Trénovacím súborom pre online generovanie bol súbor `scifi_RLOAWP2_2M.txt` a pre offline generovanie `scifi_RLOAWP2_2M_typosRF3_CTC.txt` Obidve architektúry boli testované na súbore `scifi_RLOAWP2_test_1k_typosRF3_CTC.txt`.

V tabuľke 4.3 môžme vidieť, že architektúra trénovaná generovaním chýb počas trénovania (online) dosahuje značne lepšie výsledky, ako rovnaká architektúra s predgenerovanými chybami (offline). Tento výsledok bol očakávaný keďže pri druhej spomínanej architektúre došlo k pretrénovaniu (overfitting) na trénovacie dáta a chyby, ktoré sa v nich vyskytovali. Použitá architektúra ConvLSTMCorrectionCTCBiggerPad je takmer identická s architektúrou ConvLSTMCorrectionCTCBigger zobrazenou na obrázku 3.8, líši sa v dĺžke výstupu, ktorá je 75 pri dĺžke vstupu 73 miesto 83.

Experiment 4: Vplyv počtu chýb na výsledky CTC architektúr. V 4. experimente som natrénoval tri predstavené CTC architektúry na textovom korpuse 1-billion-word-language-modeling-benchmark, ktorý obsahuje spravodajské správy v angličtine. Cieľom bolo zistiť aký vplyv bude mať rôzny počet chýb na presnosť opravovania. Architektúry boli trénované a testované na vzorkách s rôznou hustotou chýb – malá hustota (1,94 chýb na úsek textu, testované na súbore `news_test_RLOAWP2_2k_typosRF(2,1)_CTC.txt`), stredná hustota (3,79 chýb na úsek textu, testované na súbore `news_test_RLOAWP2_2k_typosRF(4,3)_CTC.txt`) a veľká hustota (5,57 chýb na úsek textu, testované na súbore `news_test_RLOAWP2_2k_typosRF(6,2)_CTC.txt`). Úsek textu má priemernú dĺžku takmer 53 znakov. Všetky architektúry boli trénované na súbore `news_train_RLOAWP2_2M.txt` online generovaním chýb s príslušnou hustotou ako je popísané v sekcií 3.3.

Architektúra	ED: 1,94	ED: 3,79	ED: 5,57
ConvLSTMCorrectionCTC	72,82%	68,13%	69,83%
ConvLSTM...Bigger	71,41%	71,18%	72,28%
ConvLSTM...BiggerPad2x	66,59%	65,46%	64,87%

Tabuľka 4.4: Tabuľka zobrazuje výsledky CTC architektúr pri rôznom počte chýb. ED udáva priemernú editačnú vzdialenosť medzi vstupným textom a ground truth.

V tabuľke 4.4 môžeme pozorovať, že najviac konzistentné výsledky dosahuje architektúra ConvLSTMCorrectionCTCBigger. Výsledky naprieč rôznou hustotou chýb sú podobné, najlepšia priemerná presnosť je pri malej hustote chýb.

Experiment 5: Podiel opravených chýb podľa typu. Experiment bol vykonaný na najviac konzistentnej architektúre zo 4. experimentu ConvLSTMCorrectionCTCBigger. Chcel som zistiť, aká časť chýb jednotlivých typov je opravená. Funkciou `editops()` z knižnice Levenshtein som získal počty chýb podľa typu (zamenený znak, nadbytočný

znak, vynechaný znak) v texte pred a po priechode neurónovou sieťou. Z týchto hodnôt som vypočítal aká percentuálna časť chýb daného typu bola opravená vzťahom:

$$1 - \frac{\text{počet chýb po priechode}}{\text{pôvodný počet chýb}}. \tag{4.2}$$

Experiment som vykonal so všetkými tromi verziami architektúry ConvLSTMCorrectionCTCBigger natrénovanými v 4. experimente. Trénovacie a testovacie súbory sú v ňom popísané.

ED	**AAC**	**Oprav. zamenený**	**Oprav. nadbytočný**	**Oprav. vynechaný**
1,94	71,41%	70,54%	81,31%	63,04%
3,79	71,18%	68,70%	83,69%	61,47%
5,57	72,28%	69,52%	83,25%	64,58%

Tabuľka 4.5: Tabuľka zobrazuje výsledky architektúry ConvLSTMCorrectionCTCBigger. ED udáva priemernú editačnú vzdialenosť medzi vstupným textom a ground truth. Stĺpec oprav. zamenený udáva aká časť chýb typu zamenený znak bola opravená. Stĺpec oprav. nadbytočný udáva aká časť chýb typu nadbytočný znak bola opravená zmazaním tohoto znaku z textu. Stĺpec oprav. vynechaný udáva aká časť chýb typu vynechaný znak bola opravená vložením správneho znaku na správne miesto.

Z tabuľky 4.5 môžeme vidieť, že pri všetkých troch hustotách chýb sa sieti najlepšie darilo opravovať chyby typu nadbytočný znak, následne chyby typu zamenený znak, a nakoniec chyby typu vynechaný znak. Myslím si, že tieto štatistiky odrážajú zložitosť opravy jednotlivých typov chýb. Chyba typu nadbytočný znak si vyžaduje proste tento znak zmazať resp. posunúť zvyšok textu doľava alebo proste generovať namiesto tohoto znaku blank či okolité znaky, chyba typu zamenený znak nevyžaduje hýbať so sekvenciou no je potrebné vybrať správny znak z abecedy, a chyba typu vynechaný znak vyžaduje „natiahnutie“ sekvencie a vybratie správneho znaku na vloženie, preto si myslím, že je najzložitejšia na opravenie. Je nutné podotknúť, že tieto štatistiky nemusia presne reflektovať množstvo opravených chýb, napríklad pri čiastočnej oprave posledného spomenutého typu, čiže vloženia znaku na správne miesto no tento znak nieje správny, vzniká chyba typu zamenený znak a chyba vynechaný znak sa počíta za opravenú.

Prikladám aj tabuľku rozloženia chýb podľa typu v testovacích súboroch, pretože aj keď je typ chyby v algoritme generovania náhodného počtu chýb generovaný rovnomerne medzi týmito troma typmi, skutočné rozloženie sa líši keďže každý typ chyby mení textový úsek iným spôsobom a chyby sa môžu generovať na rovnaké miesta. Výsledky sú v tabuľke 4.6

Súbor	**Zam.**	**Nadbyt.**	**Vynech.**
news_test_RLOAWP2_2k_typosRF(2,1)_CTC	0,3488	0,3364	0,3148
news_test_RLOAWP2_2k_typosRF(4,3)_CTC	0,3659	0,3164	0,3176
news_test_RLOAWP2_2k_typosRF(6,2)_CTC	0,3621	0,3218	0,3160

Tabuľka 4.6: Tabuľka zobrazuje skutočný pomer typov chýb v troch testovacích súboroch.

4.3 Vyhodnotenie

Celkové výsledky všetkých vykonaných pokusov, trénovaní ako aj štatistiky trénovacích a testovacích súborov je možné nájsť v súbore TabulkaVysledkov.xsl.

Architektúra	**Absolute Accuracy**
ConvLSTMDetection	**79.32%**
ConvLSTMDetectionBigger	76.89%

Tabuľka 4.7: Najlepšie výsledky pre pozične viazanú detekciu chýb dosiahla architektúra ConvLSTMDetection na datasete scifi. Obidve architektúry trénované na súbore `scifi_2M.txt`, a testované na súbore `scifi_test_test_1k_typos_2M.txt`

Architektúra	**AAC (scifi)**	**AAC (wiki)**
ConvLSTMCorrection	71,03%	62,60%
ConvLSTMCorrectionBigger	**75,89%**	**67,58%**

Tabuľka 4.8: Najlepšie výsledky pre pozične viazanú opravu chýb dosiahla architektúra ConvLSTMCorrectionBigger naprieč oboma textovými korpusmi.

Celkovo najlepšie výsledky pre detekciu dosiahla architektúra ConvLSTMDetection, bola trénovaná a testovaná na textovom korpuse scifi s priemerným počtom chýb na vzorku 3.79. Architekúra dosahuje absolútnu presnosť takmer 80% a ak z výpočtu vynecháme správne znaky detegované ako chyby, tak z pôvodných chýb deteguje správne ako chyby 89.25% chybných znakov.

V pozične viazanej architektúre mala navrch architektúra ConvLSTMCorrectionBigger. Pri oboch textových korpusoch bola priemerná hustota chýb v jednom vzorku dlhom 50 znakov 3,8, a po opravení touto architektúrou klesla hustota chýb na 0,91 chýb na vzorok.

V tabuľke 4.4 môžeme vidieť, že úplne najlepšiu absolútnu presnosť (72,82%) pri oprave chýb všetkých troch druhov, a pri úsekoch textu delených na hranici slov dosiahla architektúra ConvLSTMCorrectionCTC pri malom počte chýb (1,94), ktorá tento počet zredukuje na 0,53 chýb na vzorku. Lepšie výsledky pri strednom a veľkom počte chýb však dosahuje architektúra ConvLSTMCorrectionCTCBigger. Dosiahnuté hodnoty presnosti sú síce nižšie ako pri pozične viazaných architektúrach, je však potrebné myslieť na to, že architektúry, ktoré využívajú CTC, a dokážu opravovať všetky 3 typy chýb, tak riešia ďaleko komplexnejší problém ako pozične viazané architektúry a boli by oveľa lepšie prakticky využiteľné.

Boli testované viaceré architektúry založené na princípe konvolučných a LSTM rekurentných vrstiev na detekciu aj korekciu textu, s rôznymi spôsobmi generovania chýb, na 3 rôznych textových korpusoch. Bolo ukázané, že sa týmto prístupom, pri využití iba jedného dopredného priechodu neurónovou sieťou, dá dosiahnuť presnosť detekcie tesne pod 80%, a presnosť opravy chýb v rozmedzí 60-75%. Pri pokračovaní v práci, by som sa určite sústredil na chytrejšiu reprezentáciu znakov pre neurónovú sieť, aby sa znížili pomerne vysoké pamäťové nároky pri trénovaní architektúr, na modely, ktoré by sa viac líšili od predstavených modelov, a na zaujímavejšie experimenty s druhom a počtom chýb.

V praxi by sa navrhnuté architektúry dali využiť pre real-time detekciu a navrhovanie opráv chýb, vďaka ich rýchlosti a nízkym výpočetným nárokom v porovnaní s robustnými autoregresívnymi modelmi. Mohli by sa nasadiť priamo na hardvér užívateľa, a pracovať

aj bez pripojenia na internet. V kombinácií s fixným slovníkom by sa mohla výstupná matica sofistikovanejšie dekódovať, a navrhovať ako opravy konkrétne slová z tohoto slovníka, napríklad podľa najmenšej editačnej vzdialenosti.

Kapitola 5

Záver

Cieľom práce bolo navrhnúť, natrénovať, otestovať a porovnať achitektúry neurónových sietí, ktoré miesto autoregresívneho postupného generovania tokenov, využívajú jeden dopredný priechod, pre rýchle generovanie opraveného textu z pôvodného textu obsahujúceho chyby. Po zoznámení sa s problematikou bola vytvorená datová sada z troch textových korpusov, obsahujúca rôzne množstvo a rozloženie chýb, a navrhnuté základné pozične viazané architektúry pre detekciu chýb. Následne som implementoval pozične viazané riešenie aj na opravu textu, a zoznámil sa s technológiou CTC pre zarovnávanie sekvencií, ktorú som využil pri implementácií architektúr, ktoré dokážu vkladať kdekoľvek do textu aj väčší počet znakov. V následne vykonaných experimentoch boli porovnané vybrané prístupy. Bola dosiahnutá presnosť takmer 80% pri detekcií chýb v texte, viac ako 75% presnosť opravy chýb typu zamenený a nadbytočný znak, pri pozične viazaných architektúrach, a až 73% presnosť pri oprave chýb všetkých typov, na úsekoch textu rôznej dĺžky, delených na hranici slov, pri využití CTC.

V práci by bolo vhodné pokračovať vykonaním ďalších experimentov nad už natrénovanými architektúrami, a optimalizáciou pamäťových nárokov trénovania. Tiež by som chcel implementovať a otestovať viacnásobný priechod navrhnutými architektúrami, a porovnať tieto výsledky s dosiahnutými výsledkami pri jedinom priechode. Vhodné by taktiež bolo pokúsiť sa skombinovať navrhnuté architektúry so slovníkom slov, z ktorého by sa na základe výstupu zo siete mohli navrhovať konkrétne slová ako návrhy na opravu.

Literatúra

[1] Basta, N. *The Differences between Sigmoid and Softmax Activation Functions* [online]. 2020 [cit. 2023-04-08]. Dostupné z: https://medium.com/arteos-ai/the-differences-between-sigmoid-and-softmax-activation-function-12adee8cf322.

[2] Bishop, C. M. Neural networks and their applications. *Review of Scientific Instruments*. 1994, zv. 65, č. 6, s. 1803–1832. DOI: 10.1063/1.1144830. Dostupné z: https://doi.org/10.1063/1.1144830.

[3] Brownlee, J. *How Do Convolutional Layers Work in Deep Learning Neural Networks?* [online]. 2020 [cit. 2023-04-08]. Dostupné z: https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/.

[4] Brownlee, J. *Difference Between Backpropagation and Stochastic Gradient Descent* [online]. 2021 [cit. 2023-04-28]. Dostupné z: https://machinelearningmastery.com/difference-between-backpropagation-and-stochastic-gradient-descent/.

[5] Chen, B. *7 popular activation functions you should know in Deep Learning and how to use them with Keras and TensorFlow 2* [online]. 2021 [cit. 2023-04-08]. Dostupné z: https://towardsdatascience.com/7-popular-activation-functions-you-should-know-in-deep-learning-and-how-to-use-them-with-keras-and-27b4d838dfe6.

[6] Chiron, G., Doucet, A., Coustaty, M. a Moreux, J.-P. ICDAR2017 Competition on Post-OCR Text Correction. In: *2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR)*. 2017, sv. 01, s. 1423–1428. DOI: 10.1109/ICDAR.2017.232.

[7] Devlin, J., Chang, M., Lee, K. a Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR*. 2018, abs/1810.04805. Dostupné z: http://arxiv.org/abs/1810.04805.

[8] D'hondt, E., Grouin, C. a Grau, B. Low-resource OCR error detection and correction in French Clinical Texts. In: *Proceedings of the Seventh International Workshop on Health Text Mining and Information Analysis*. Auxtin, TX: Association for Computational Linguistics, November 2016, s. 61–68. DOI: 10.18653/v1/W16-6108. Dostupné z: https://aclanthology.org/W16-6108.

[9] Dong, R. a Smith, D. Multi-Input Attention for Unsupervised OCR Correction. In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Melbourne, Australia: Association for Computational Linguistics, Júl 2018, s. 2363–2372. DOI: 10.18653/v1/P18-1220. Dostupné z: https://aclanthology.org/P18-1220.

[10] Goodfellow, I., Bengio, Y. a Courville, A. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[11] Graves, A. a Schmidhuber, J. Framewise phoneme classification with bidirectional LSTM networks. In: *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.* 2005, sv. 4, s. 2047–2052 vol. 4. DOI: 10.1109/IJCNN.2005.1556215.

[12] Graves, A., Fernández, S., Gomez, F. a Schmidhuber, J. Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks. In: *Proceedings of the 23rd International Conference on Machine Learning*. New York, NY, USA: Association for Computing Machinery, 2006, s. 369–376. ICML '06. DOI: 10.1145/1143844.1143891. ISBN 1595933832. Dostupné z: https://doi.org/10.1145/1143844.1143891.

[13] Gu, J., Wang, Z., Kuen, J., Ma, L., Shahroudy, A. et al. Recent advances in convolutional neural networks. *Pattern Recognition*. 2018, zv. 77, s. 354–377. DOI: https://doi.org/10.1016/j.patcog.2017.10.013. ISSN 0031-3203. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0031320317304120.

[14] Hochreiter, S. a Schmidhuber, J. Long Short-term Memory. *Neural computation*. December 1997, zv. 9, s. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[15] Jayanthi, S. M., Pruthi, D. a Neubig, G. NeuSpell: A Neural Spelling Correction Toolkit. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Október 2020, s. 158–164. DOI: 10.18653/v1/2020.emnlp-demos.21. Dostupné z: https://aclanthology.org/2020.emnlp-demos.21.

[16] Li, H., Wang, Y., Liu, X., Sheng, Z. a Wei, S. Spelling Error Correction Using a Nested RNN Model and Pseudo Training Data. *CoRR*. 2018, abs/1811.00238. Dostupné z: http://arxiv.org/abs/1811.00238.

[17] Mokhtar, K., Bukhari, S. S. a Dengel, A. OCR Error Correction: State-of-the-Art vs an NMT-based Approach. In: *2018 13th IAPR International Workshop on Document Analysis Systems (DAS)*. 2018, s. 429–434. DOI: 10.1109/DAS.2018.63.

[18] Nastase, V. a Hitschler, J. Correction of OCR Word Segmentation Errors in Articles from the ACL Collection through Neural Machine Translation Methods. In: *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*. Miyazaki, Japan: European Language Resources Association (ELRA), Máj 2018. Dostupné z: https://aclanthology.org/L18-1113.

[19] Nguyen, T. T. H., Jatowt, A., Coustaty, M. a Doucet, A. Survey of Post-OCR Processing Approaches. *ACM Comput. Surv.* New York, NY, USA: Association for Computing Machinery. jul 2021, zv. 54, č. 6. DOI: 10.1145/3453476. ISSN 0360-0300. Dostupné z: https://doi.org/10.1145/3453476.

[20] Pramoditha, R. *The Concept of Artificial Neurons (Perceptrons) in Neural Networks, Neural Networks and Deep Learning Course: Part 1* [online]. 2021 [cit.

2023-04-05]. Dostupné z: https://towardsdatascience.com/the-concept-of-artificial-neurons-perceptrons-in-neural-networks-fab22249cbfc.

[21] Pröve, P.-L. *An Introduction to different Types of Convolutions in Deep Learning* [online]. 2017 [cit. 2023-05-01]. Dostupné z: https://towardsdatascience.com/types-of-convolutions-in-deep-learning-717013397f4d.

[22] Rigaud, C., Doucet, A., Coustaty, M. a Moreux, J.-P. ICDAR 2019 Competition on Post-OCR Text Correction. In: *2019 International Conference on Document Analysis and Recognition (ICDAR)*. 2019, s. 1588–1593. DOI: 10.1109/ICDAR.2019.00255.

[23] Sakaguchi, K., Duh, K., Post, M. a Durme, B. V. Robsut Wrod Reocginiton via semi-Character Recurrent Neural Network. *CoRR*. 2016, abs/1608.02214. Dostupné z: http://arxiv.org/abs/1608.02214.

[24] Scheidl, H. *An Intuitive Explanation of Connectionist Temporal Classification* [online]. 2018 [cit. 2023-04-18]. Dostupné z: https://towardsdatascience.com/intuitively-understanding-connectionist-temporal-classification-3797e43a86c.

[25] Staudemeyer, R. C. a Morris, E. R. Understanding LSTM–a tutorial into long short-term memory recurrent neural networks. *ArXiv preprint arXiv:1909.09586*. 2019. Dostupné z: https://arxiv.org/pdf/1909.09586.pdf.

[26] Zarzycki, K. a Ławryńczuk, M. LSTM and GRU Neural Networks as Models of Dynamical Processes Used in Predictive Control: A Comparison of Models Developed for Two Chemical Reactors. *Sensors*. 2021, zv. 21, č. 16. DOI: 10.3390/s21165625. ISSN 1424-8220. Dostupné z: https://www.mdpi.com/1424-8220/21/16/5625.

Príloha A

Obsah priloženého pamäťového média

Priložené pamäťové médium (SD karta) obsahuje:

- Priečinok `src` obsahujúci všetky zdrojové kódy
- Priečinok `data` obsahujúci vytvorené trénovacie a validačné súbory
- Priečinok `models` obsahujúci natrénované architektúry, každú vo zvlášť priečinku, ktorý obsahuje model, výstup z trénovania toho modelu a súbor obsahujúci testovacie dáta a výstupy z natrénovanej architektúry, vhodný na počítanie štatistík.
- Priecinok `src_latex` s archívom obsahujúcim zdrojové LaTeX súbory k tomuto dokumentu
- Tento PDF súbor
- Video `video.mp4` prezentujúce túto prácu a dosiahnuté výsledky