

UNIVERZA V MARIBORU
FAKULTETA ZA ELEKTROTEHNIKO,
RAČUNALNIŠTVO IN INFORMATIKO

Tomaž Piko

**IZDELAVA POGOVORNEGA ROBOTA Z
REKURENTNO NEVRONSKO MREŽO
LSTM**

Diplomsko delo

Maribor, september 2020

UNIVERZA V MARIBORU
FAKULTETA ZA ELEKTROTEHNIKO,
RAČUNALNIŠTVO IN INFORMATIKO

Tomaž Piko

IZDELAVA POGOVORNEGA ROBOTA Z REKURENTNO NEVRONSKO MREŽO LSTM

Diplomsko delo

Maribor, september 2020

IZDELAVA POGOVORNEGA ROBOTA Z REKURENTNO NEVRONSKO MREŽO LSTM

Diplomsko delo

Študent: Tomaž Piko

Študijski program: Univerzitetni študijski program
računalništvo in informacijske tehnologije

Mentor: red. prof. dr. Aleš Holobar, univ. dipl. inž. rač. in inf.

Somentorja: asist. Mladen Borovič, mag. inž. rač. in inf. tehnol.
dr. Milan Ojsteršek, univ. dipl. inž. el.

ZAHVALA

Zahvaljujem se mentorju dr. Alešu Holobarju za prevzem mentorstva ter somentorjema dr. Milanu Ojsteršku in asist. Mladenu Boroviču za svetovanje in usmerjanje pri pisanju diplomskega dela.

Hvala tudi moji družini za neizmerno podporo pri študiju.

Izdelava pogovornega robota z rekurentno nevronske mrežo LSTM

Ključne besede: pogovorni roboti, rekurentne nevronske mreže, celica LSTM, obdelava naravnih jezikov

UDK: 004.934(043.2)

Povzetek

V diplomskem delu so v prvem delu najprej predstavljeni pogovorni roboti in njihovi tipi, nato rekurentne nevronske mreže ter delovanje različnih celic, ki jih pri njih najpogosteje srečujemo. V drugem delu pa je prikazan primer implementacije in učenja rekurentne nevronske mreže LSTM (Long Short-Term Memory) ter izdelava mobilne aplikacije, v kateri lahko pisno komuniciramo z izdelano mrežo oziroma našim pogovornim robotom v slovenskem ali angleškem jeziku.

Chatbot implementation using a LSTM recurrent neural network

Key words: chatbots, recurrent neural networks, LSTM cell, natural language processing

UDC: 004.934(043.2)

Abstract

The first part of this diploma thesis describes chatbots and their types. We then describe recurrent neural networks and how their most frequently used types of cells work. The second part shows an example of implementing and training a recurrent neural network LSTM (Long Short-Term Memory) and developing a mobile application in which we can communicate with the created neural network and our own chatbot via text in the Slovenian or English language.

KAZALO VSEBINE

1	UVOD	1
2	POGOVORNI ROBOTI	2
2.1	Turingov test	2
2.2	Osnove delovanja pogovornih robotov	3
2.2.1	Delovanje na osnovi pravil	3
2.2.2	Delovanje na osnovi umetne inteligence	3
2.3	Pogovorni robot ELIZA	4
2.4	Pogovorni roboti na pogovornih platformah	5
2.5	Virtualni pomočniki	6
3	REKURENTNE NEVRONSKE MREŽE	8
3.1	Celica LSTM (Long Short-Term Memory)	11
3.2	Celica GRU (Gated Recurrent Unit)	14
4	IMPLEMENTACIJA NEVRONSKE MREŽE	16
4.1	Priprava podatkovnih zbirk	17
4.1.1	Pridobivanje slovenskih podatkov	17
4.1.2	Pridobivanje angleških podatkov	19
4.2	Tokenizacija povedi	20
4.3	Besedne vložitve	22
4.3.1	Word2Vec	22
4.3.2	GloVe	24
4.4	Vektorizacijska plast nevronske mreže	24
4.5	Implementacija modela	25
4.6	Učenje modela	26

4.7	Uporaba modela	28
5	IZDELAVA APLIKACIJE.....	33
5.1	Izdelava z ogrodjem Angular	33
5.2	Pretvorba v mobilno aplikacijo	38
6	REZULTATI	39
7	ZAKLJUČEK.....	49
8	VIRI IN LITERATURA	50

KAZALO SLIK

Slika 2.1: Primer našega pogovora z <i>ELIZA</i> [7].	5
Slika 2.2: Primer našega pogovora s pogovornim robotom na osnovi pravil.	6
Slika 3.1: Razvita rekurentna nevronska mreža. Povzeto po [17].	8
Slika 3.2: Kombinacije vhodov in izhodov RNN. Povzeto po [18].	9
Slika 3.3: Ponavljajoča se celica preproste RNN. Povzeto po [17].	10
Slika 3.4: Ponavljajoča se celica LSTM. Povzeto po [17].	11
Slika 3.5: Opis notacije. Povzeto po [17].	11
Slika 3.6: Stanje celice LSTM. Povzeto po [17].	12
Slika 3.7: Vrata za pozabljanje. Povzeto po [17].	12
Slika 3.8: Vhodna vrata. Povzeto po [17].	13
Slika 3.9: Posodabljanje stanja celice. Povzeto po [17].	13
Slika 3.10: Izhodna vrata. Povzeto po [17].	14
Slika 3.11: Celica GRU. Povzeto po [17].	15
Slika 4.1: Shema arhitekture kodirnik-dekodirnik. Povzeto po [21].	16
Slika 4.2: Slovenski podatki pred obdelavo.	18
Slika 4.3: Slovenski podatki po obdelavi.	19
Slika 4.4: Angleški podatki pred obdelavo.	20
Slika 4.5: Programska koda tokenizacije vhodnih in izhodnih podatkov.	21
Slika 4.6: Primerjava delovanja obeh Word2Vec modelov. Povzeto po [26].	23
Slika 4.7: Programska koda primera uporabe Word2Vec.	23
Slika 4.8: Programska koda implementacije vektorizacijske plasti.	25
Slika 4.9: Programska koda implementacije kodirnik-dekodirnik plasti.	25
Slika 4.10: Programska koda generatorja učnih podatkov.	27
Slika 4.11: Programska koda funkcije za učenje modela.	27
Slika 4.12: Programska koda za razdelitev modela na kodirnik in dekodirnik.	28
Slika 4.13: Shema kodirnika.	29
Slika 4.14: Shema dekodirnika.	29
Slika 4.15: Shema naše implementacije arhitekture kodirnik-dekodirnik.	31
Slika 4.16: Programska koda za shranjevanje modelov.	32

Slika 5.1: Končni izgled aplikacije izdelane v ogrodju Angular.....	34
Slika 5.2: Uvoz knjižnic in datotek JSON v aplikacijo.	35
Slika 5.3: Programska koda za nalaganje modelov iz datotek JSON.	35
Slika 5.4: Programska koda za generiranje odziva pogovornega robota.	36
Slika 5.5: Shema delovnega toka v aplikaciji.....	37
Slika 6.1: Graf odstotkov povedi glede na dolžine povedi v slovenski podatkovni zbirki.	40
Slika 6.2: Graf odstotkov povedi glede na dolžine povedi v angleški podatkovni zbirki.	40
Slika 6.3: Grafa natančnosti in izgube prvega modela.	42
Slika 6.4: Prikaz učinka različnih velikosti paketa na potek učenja.....	43
Slika 6.5: Primeri slovenskih pogovorov z velikostjo paketa 32.	44
Slika 6.6: Primeri slovenskih pogovorov z velikostjo paketa 128.	44
Slika 6.7: Grafa primerjave slovenskega in angleškega modela.	45
Slika 6.8: Primeri angleških pogovorov z velikostjo paketa 32.	46
Slika 6.9: Primerjava angleških modelov glede na velikost podatkovne zbirke.	47
Slika 6.10: Primeri pogovorov modelov z različnim številom vhodnih podatkov.....	48

SEZNAM UPORABLJENIH KRATIC

UI – Umetna inteligenca (angl. Artificial Intelligence)

NLP – Obdelava naravnega jezika (angl. Natural Language Processing)

RNN – Rekurentna nevronska mreža (angl. Recurrent Neural Network)

Celica LSTM – Dolgo kratko-ročna spominska celica (angl. Long Short-Term Memory)

CNN – Konvolucijske nevronske mreže (angl. Convolutional Neural Network)

Celica GRU – Vratna rekurentna celica (angl. Gated Recurrent Unit)

Seq2seq – Zaporedje v zaporedje (angl. Sequence to sequence)

API – Aplikacijski programski vmesnik (angl. Application programming interface)

Word2vec – Word to vector

GloVe – Global Vectors for Word Representation

CBOW – Continuous Bag Of Words

HTML – Hyper Text Markup Language

CSS – Cascading Style Sheets

JSON – Objektna notacija za JavaScript (angl. JavaScript Object Notation)

SDK – razvijalski paket (angl. Software Development Kit)

JDK – Javansko razvojno okolje (angl. Java Development Kit)

1 UVOD

Umetna inteligenca (v nadaljevanju UI) postaja del našega vsakdana, saj veliko programskih rešitev na pametnih telefonih in napravah, ki jih uporabljamo v vsakdanjem življenju, uporablja njene algoritme. Glavna ideja raziskovalcev, ki razvijajo algoritme, temelječe na umetni inteligenci, je razviti računalniške programe, ki delujejo podobno kot človeški možgani. Ti programi so se na podlagi učnih podatkov sposobni nekaj novega naučiti, logično sklepati ali rešiti problem. UI uporabljamo tudi pri delovanju tako imenovanih pogovornih robotov, ki uporabljajo nevronske mreže, ki se s pomočjo učne množice besedil nauči razbrati pomen povedi in sestaviti smiselni odgovor. Pogovorni roboti so večinoma specializirani za komuniciranje znotraj določene teme (na primer odgovarjajo na vprašanja strank določene trgovine), zato tej temi prilagodimo njihovo delovanje in učne podatke.

Namen diplomskega dela je bil spoznati sestavo in delovanje rekurentnih nevronske mreže ter kako jih uporabiti pri izdelavi pogovornega robota. Cilj je bil s pridobljenim znanjem izdelati in naučiti nevronske mreže, ki jo lahko uporabimo v mobilni aplikaciji za smiselno pisno komunikacijo s pogovornim robotom.

V drugem poglavju smo opisali delovanje pogovornih robotov in podali primere nekaterih bolj znanih in širše uporabljenih robotov. V tretjem poglavju smo opisali rekurentne nevronske mreže in podrobneje predstavili vrsto le-teh, ki smo jo uporabili pri izdelavi pogovornega robota. V četrtem poglavju smo opisali tehnologije, uporabljene pri zasnovi nevronske mreže, in kako smo pridobili ter preuredili podatke za učenje. V petem poglavju smo na kratko opisali ogrodje, ki smo ga uporabili pri izdelavi aplikacije, in kako je izdelava potekala. V zadnjem poglavju smo prikazali rezultate pogovorov glede na različne podatke, uporabljene pri učenju, in opisali trajanje učenja ter izpostavili možnosti za morebitne izboljšave.

2 POGOVORNI ROBOTI

Pogovorni robot (angl. chatbot) je računalniški program, ki preko glasovne ali pisne komunikacije komunicira z uporabnikom. Začeli so jih razvijati v 60-tih letih prejšnjega stoletja, ko so preizkušali, če lahko računalniški program prelisiči končnega uporabnika, tako da ta ne ugotovi, da se ne pogovarja s pravim človekom. Če uporabnik ugotovi, da ne komunicira s pravim človekom, opravi pogovorni robot tako imenovani Turingov test [1].

Sprva je bila izdelava pogovornih robotov le vir zabave in so v vnosu uporabnika iskali ključne besede, na katere so imeli vnaprej pripravljene odgovore. Z napredkom razpoznave besedil, obdelave naravnih jezikov, dostopnosti do velikih količin besedil in umetne inteligence ter seveda računalniških zmogljivosti je prišlo do razvoja veliko bolj kompleksnih pogovornih robotov [2].

Danes postajajo pogovorni roboti vedno bolj popularni. So zanimiv in interaktivni medij za posredovanje informacij, zato jih večkrat srečujemo v pojavnih oknih na spletu. Podjetjem omogočajo, da rešijo večjo število povpraševanj strank in hkrati zmanjšajo potrebo po človeški interakciji. Z uporabo pogovornega robota lahko dobijo stranke takojšnji odgovor, saj ni potrebno čakati na prostega delavca. Prav tako so lahko na voljo izven delovnih ur podjetja (24 ur na dan) [3].

2.1 Turingov test

Turingov test je leta 1950 zasnoval Alan Turing. Gre za preizkus zmožnosti stroja izkazovati inteligentno obnašanje, ki je ekvivalentno oziroma nerazločljivo od človeškega. Test ni mišljen za preverjanje sposobnosti stroja, da pravilno odgovarja na vprašanja. Pomembno je le, da so ti odgovori karseda podobni človeškim odgovorom [4].

Turing je predlagal, da se človeški sodnik vključi v pogovor v naravnem jeziku med človekom in strojem, izdelanim, da vrača človeku podobne odgovore. Sodnik ve, da je eden od njegovih pogovornih partnerjev robot, vsi udeleženci pa so fizično ločeni. Prav

tako je pogovor omejen le na pisno komunikacijo preko monitorja in tipkovnice, tako da rezultati niso odvisni od morebitne zmožnosti stroja, da analizira govor. Če sodnik ne more z gotovostjo razločiti med človekom in strojem, potem je ta stroj prestal test [4].

2.2 Osnove delovanja pogovornih robotov

Glede na to, na kakšen način komuniciramo z njim in kako vrne odgovor, poznamo dva osnovna načina delovanja pogovornega robota.

2.2.1 Delovanje na osnovi pravil

Prvi tip so pogovorni roboti, ki temeljijo na pravilih (angl. rule-based chatbots) oziroma odločitvenih drevesih. Imajo vnaprej določena vprašanja, med katerimi lahko izbiramo, in na njih pripravljene odgovore. Ta postopek se ponavlja, dokler ne pridemo do končne rešitve. V komercialni uporabi nas, v kolikor od njih ne dobimo željenih informacij, napotijo naprej, v večini primerov do fizične osebe, odgovorne za storitve za stranke. Uporabljajo lahko zelo preprosta ali pa zapletena pravila, vendar niso sposobni odgovarjati na vprašanja izven definiranih pravil in se ne učijo sproti skozi interakcije z ljudmi. Njihove glavne prednosti so, da so enostavni in hitri za implementacijo, so zanesljivi in varni ter lahko v pogovoru vsebujejo tudi interaktivne vsebine (na primer slike, animacije, video) [5] [6].

2.2.2 Delovanje na osnovi umetne inteligence

Drugi tip so pogovorni roboti, ki temeljijo na uporabi algoritmov umetne inteligence (angl. artificial intelligence chatbots). So bolj dinamični in zahtevnejši za sprogramirati kot pogovorni roboti, ki delujejo na podlagi pravil, saj se ne zanašajo na vnaprej definirane vnose. Dodatno jih lahko delimo še v dve podkategoriji [5] [6].

Prva podkategorija se zanaša na obdelavo naravnega jezika (angl. natural language processing – NLP), ki iz uporabnikovega vnosa izlušči kontekst in namen vprašanja. Ljudje

se izražamo različno, uporabljamo narečne izraze, žargon in včasih pomotoma napačno črkujemo besede. Ta vrsta robotov poskuša razumeti človeški jezik in izbrati primeren odziv [6].

Druga podkategorija za učinkovito delovanje poleg koncepta obdelave naravnega jezika uporablja še strojno učenje. Ti pogovorni roboti uporabljajo kompleksne nevronske mreže, s pomočjo katerih se iz velikih količin podatkov naučijo uporabnikom nuditi smiselne odgovore. V primeru pogovornih robotov so učni podatki po navadi pogovori, kakršne hočemo z robotom pustvariti [6].

2.3 Pogovorni robot ELIZA

Leta 1966 je Joseph Weizenbaum razvil prvega pogovornega robota, ki naj bi prestal Turingov test [4]. Poimenoval ga je *ELIZA*. Prikazuje primer pogovornega robota, razvitega v času, ko so se izdelovali predvsem za zabavo. Deloval je na principu iskanja ključnih besed v uporabnikovem vnosu, na katere je imel vnaprej določene odzive. Če ključne besede ni našel, pa je ponovil enega izmed svojih prejšnjih odgovorov ali pa se je odgovoru izmikal. Na spletu lahko najdemo več spletnih različic tega pogovornega robota. Slika 2.1 prikazuje primer pogovora. Povrh vsega pa naj bi *ELIZA* predstavljala psihoterapevtkinjo, zaradi česar so njeni izmikajoči odgovori toliko bolj prepričljivi. S temi tehnikami je Weizenbaum-ov program uspel mnoge prelisičiti. Nekatere poskusne osebe naj bi bilo celo težko prepričati, da *ELIZA* v resnici ni človek [4] [7].

Talk to Eliza by typing your questions and answers in the input box.

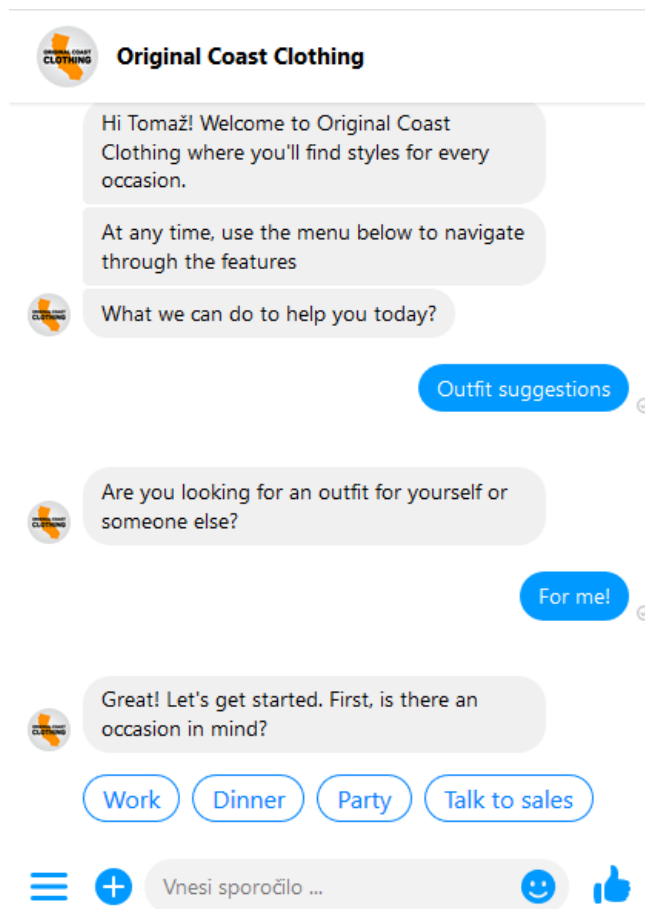
```
> Hello, I am Eliza. I'll be your therapist today.  
* Hello. I am Tomaž. I'll be your patient today.  
> Did you come to me because you are Tomaž. you'll be my patient today?  
* No i came to you because im experiencing problems.  
> We were discussing you, not me.  
* But we are discussing me?  
> I think you are blocking what you really want to say.  
* I am not.  
> Do you believe it is normal to be not?
```

TYPE HERE

Slika 2.1: Primer našega pogovora z *ELIZA* [7].

2.4 Pogovorni roboti na pogovornih platformah

Dandanes ljudje skoraj vsakodnevno uporabljamo različne pogovorne platforme, kot so Facebook messenger, Skype, Slack itd. Vse te platforme omogočajo tudi uporabo in razvoj pogovornih robotov. Osredotočili se bomo na Facebook, ki nudi dokumentacijo za razvijalce, ki želijo za njihovo aplikacijo Facebook messenger razviti svoje pogovorne robote. V dokumentaciji je tudi nekaj pogovornih robotov, namenjenih izključno za demonstracijo njihovega delovanja. Slika 2.2 prikazuje primer našega pogovora z enim od njihovih testnih pogovornih robotov, ki temelji na pravilih in nam na vsakem koraku ponudi nekaj gumbov, ki narekujejo temo pogovora [8].



Slika 2.2: Primer našega pogovora s pogovornim robotom na osnovi pravil.

2.5 Virtualni pomočniki

Lahko bi rekli, da so virtualni pomočniki (angl. virtual assistant) pogovorni roboti, ki so sposobni izvajati določene naloge oziroma dejavnosti za posameznika glede na ukaze ali vprašanja. Nekateri virtualni pomočniki so sposobni razpoznavati človeški govor in odgovarjati s pomočjo sinteze govora. Uporabniki lahko svoje pomočnike sprašujejo, preko njih upravljajo druge naprave (na primer glasbeni predvajalnik, televizor) ali pa izvajajo enostavne naloge kot je na primer upravljanje opomnikov na koledarju določene osebe [9].

Večino virtualnih pomočnikov zasledimo na pametnih telefonih, med najbolj poznanimi po svetu pa so Google-ov Google assistant [10], Apple-ova Siri [11], Amazon-ova Alexa

[12]. Seveda jih zasledimo tudi na osebnih računalnikih. Primer tega je Microsoft-ova Cortana [13].

Apple-ova Siri je virtualna pomočnica, ki je prinesla v ospredje koncept glasovno aktiviranih pomočnikov [11]. Sprva je bila leta 2010 razvita kot samostojna aplikacija, vendar je Apple hitro pograbil priložnost in jo integriral v svoje pametne telefone. Aktiviramo jo z gumbom ali pa glasovnim ukazom »Hey Siri«. Podpira osnovne naloge, kot so klicanje ljudi iz imenika, odgovarjanje na vprašanja, nastavljanje budilke, prikaz opomnikov in še več. Zelo znana je tudi po svojih šaljivih odgovorih [14].

Google je leta 2016 predstavil svojega virtualnega pomočnika imenovanega Google assistant [10]. Kljub temu, da je bil razvit pozneje od ostalih, mnoge raziskave Google assistant uvrščajo med najboljše kar se tiče razpoznavanja ukazov in nudenja pravih odgovorov [14].

Amazon-ova Alexa je kljub temu, da raziskave kažejo, da slabše razpoznava ukaze kot prej omenjena virtualna pomočnica, zelo popularna kot hišni pomočnik. To je predvsem zaradi njenega dobrega sodelovanja z ostalimi pametnimi napravami, ki jih imamo doma (na primer telefoni, televizorji in zvočniki) [14].

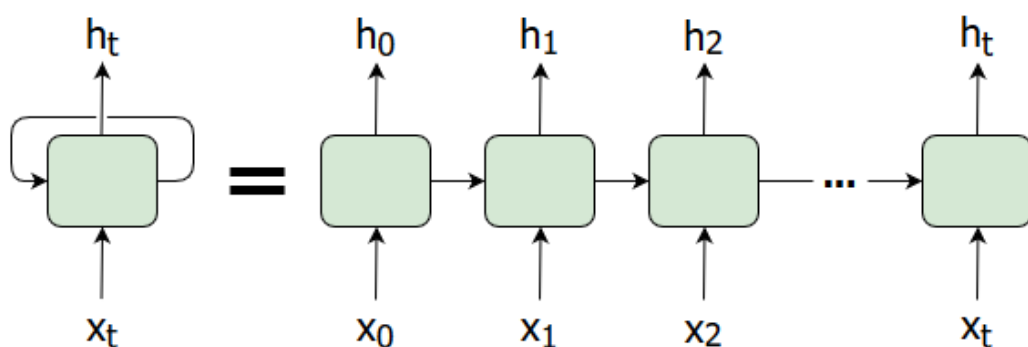
Ostane nam le še Cortana, ki jo najdemo na osebnih računalnikih, ki uporabljajo operacijski sistemi Windows [15]. Uporabna je predvsem skupaj z ostalimi Microsoft-ovimi programi, kot so Office 365 in Outlook [14].

Glavna prednost virtualnih pomočnikov je priročnost in njihova glasovna aktivacija, ki lahko zelo pomaga ljudem s posebnimi potrebami. Prav tako je glasovna izkušnja bolj zabavna kot tipkanje ukazov. Imajo pa seveda tudi svoje slabosti. Njihovo glasovno razpoznavanje lahko hitro zataji, če je veliko šuma v ozadju ali pa ima uporabnik poseben naglas. Eden glavnih razlogov za skrb je tudi varnost, saj naprava, ki neprestano posluša kaj se dogaja okoli nje, posega v našo zasebnost [14].

3 REKURENTNE NEVRONSKE MREŽE

Rekurentne nevronske mreže (angl. recurrent neural networks – RNN) so močna in robustna vrsta nevronskih mrež in spadajo med najbolj obetavne algoritme trenutno v uporabi. Kot veliko drugih algoritmov za globoko učenje (angl. deep learning) so bile rekurentne nevronske mreže razvite že v 1980-tih, a šele pretekla leta jih lahko srečujemo v njihovem polnem potencialu. V ospredje jih je pripeljalo povečanje zmogljivosti računalnikov ter dostopnost do velikih količin podatkov, ki jih lahko uporabljamo pri učenju, vključno z razvojem celice LSTM (Long Short-Term Memory) v 1990-tih. Uporabljajo se celo v Google prevajalniku in virtualnih pomočnikih kot je Apple-ova Siri. Zaradi njihovega notranjega spomina si lahko zapomnijo pomembne stvari o vhodih, ki so jih prejele, kar jim omogoča, da so zelo natančne v napovedovanju rezultatov. Zaradi tega so prva izbira za zaporedne podatke kot so govor, tekst, finančni podatki, zvok, video in še veliko več. Rekurentne nevronske mreže lahko tvorijo veliko globlje razumevanje zaporedja in njegovega pomena v primerjavi z drugimi algoritmi [16].

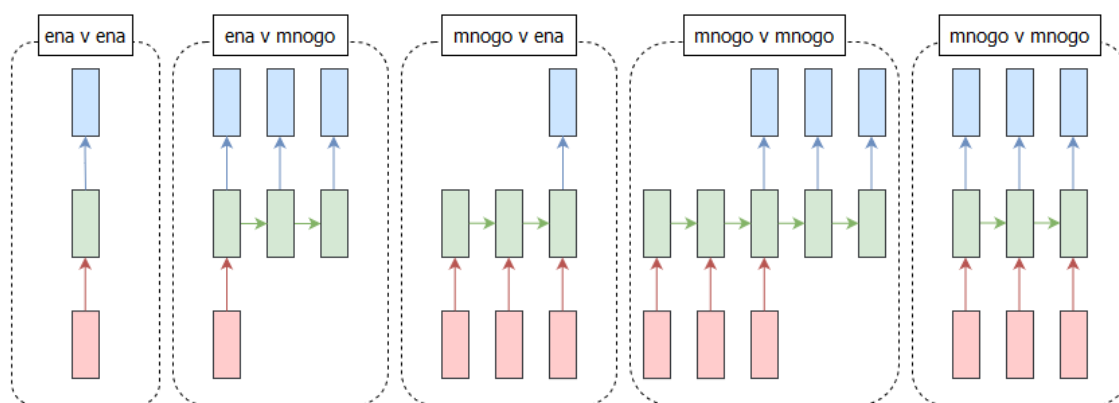
Rekurentne nevronske mreže so v osnovi zelo podobne navadnim, le da vsebujejo notranji spomin. To dosežejo z zanko tako, da po vsakem izračunu svoj izhod pošljejo nazaj v mrežo, kjer se uporabi kot dodaten vhod na naslednjem koraku in vpliva na izhod. Zaradi teh zank se lahko na prvi pogled zdijo rekurentne nevronske mreže zelo kompleksne. Za lažje razumevanje si jih lahko predstavljamo kot množico kopij iste mreže. Vsaka kopija prenaša sporočilo do svojega naslednika. Primer takšne predstavitve vidimo na sliki 3.1 [17].



Slika 3.1: Razvita rekurentna nevronska mreža. Povzeto po [17].

Verigi podobna struktura na sliki 3.1 razkrije, da so rekurentne nevronske mreže tesno povezane z zaporedji in seznamami [17].

Izrazite pomanjkljivosti običajnih in prav tako konvolucijskih nevronske mrež (angl. convolutional neural networks – CNN) so, da je njihovo delovanje zelo omejeno. Kot vhod prejmejo vektor fiksne velikosti (na primer slika) in na izhodu proizvedejo prav tako vektor fiksne velikosti. Poleg tega vse to dosežejo s fiksno količino računskih korakov, določeno s številom skritih plasti v mreži. Glavni razlog, zakaj vedno pogosteje srečujemo rekurentne nevronske mreže je, da nam omogočajo, da operiramo nad različno dolgiimi zaporedji vektorjev. V tem primeru je zaporedje vhod ali izhod, v večini primerov pa kar oboje. Slika 3.2 nam pomaga razumeti vse mogoče kombinacije, ki jih srečujemo v rekurentnih nevronske mrežah [18].



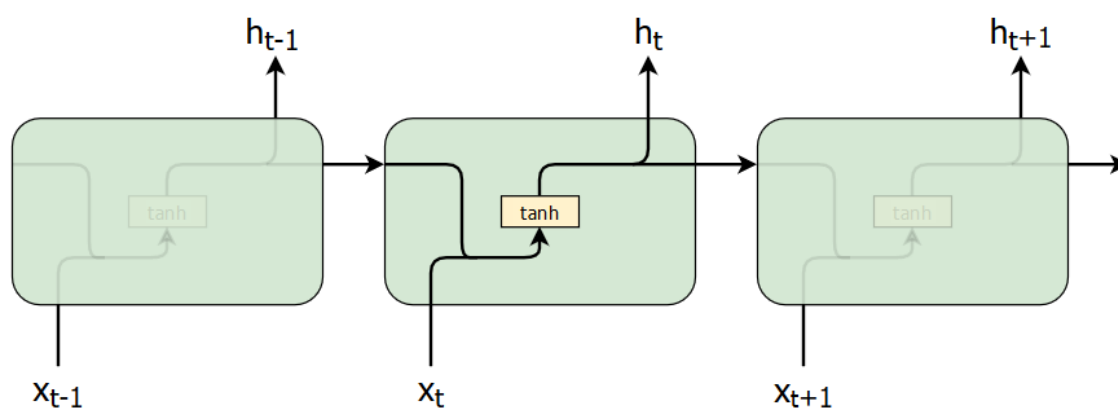
Slika 3.2: Kombinacije vhodov in izhodov RNN. Povzeto po [18].

Na sliki 3.2 vsak pisani pravokotnik predstavlja vektor, puščice pa funkcije. Rdeči kvadrati so vhodni vektorji, modri pa izhodni. Zeleni predstavljajo stanje celice. Več o tem in funkcijah bomo spoznali v nadaljevanju. Od leve proti desni prvi prikaz predstavlja en vhod in en izhod, ki ga srečujemo pri klasifikaciji slik. V drugem prikazu imamo en vhod (na primer slika), izhod pa je zaporedje vektorjev (na primer besede, ki opisujejo dano sliko). Pri tretjem prikazu je ravno obratno, zaporedje kot vhod in en izhod (na primer vnesemo poved in dobimo izhod ali poved izraža pozitivno ali negativno mnenje). Zadnja dva primera oba prikazujeta zaporedje vektorjev kot vhod in izhod. Četrty prikaz srečujemo pri prevajanju povedi. Zadnji primer se od četrtega razlikuje v tem, da so

vhodi in izhodi sinhronizirani, vendar vsak vhod vpliva na naslednjega. Primer tega je klasifikacija videa, kjer hočemo vsako sličico uvrstiti v določen razred [18].

Za potrebe opisa delovanja rekurentnih nevronske mreže bomo spoznali dve vrsti aktivacijskih funkcij. Prva je sigmoidna aktivacijska funkcija, ki vrednosti normalizira na interval $[0, 1]$. Druga pa je hiperbolični tangens oziroma funkcija \tanh , ki zavzame vrednosti na intervalu $[-1, 1]$ [19].

Vse različice rekurentnih nevronske mreže imajo obliko verige ponavljajočih se celic. V običajnih rekurentnih nevronske mrežah imajo celice zelo preprosto strukturo, na sliki 3.3 je prikazan primer celice, ki vsebuje le eno plast z aktivacijsko funkcijo \tanh . Takšne trpijo zaradi kratkoročnega spomina. Če je vneseno zaporedje dovolj dolgo, bodo imele probleme s prenašanjem pomembnih informacij skozi večje število časovnih korakov. Ko torej poizkušamo procesirati daljši odstavek besedila, lahko izpustijo ključne informacije iz začetka odstavka. Za reševanje problema kratkoročnega spomina sta bila razvita dva posebna tipa rekurentnih nevronske mreže, to sta mreža LSTM (Long Short-Term Memory) in mreža GRU (Gated Recurrent Unit). Obe v osnovi delujeta isto kot običajne rekurentne nevronske mreže, a sta se sposobni naučiti dolgoročne povezave s pomočjo mehanizma, ki ga imenujemo vrata (angl. gates) [17].

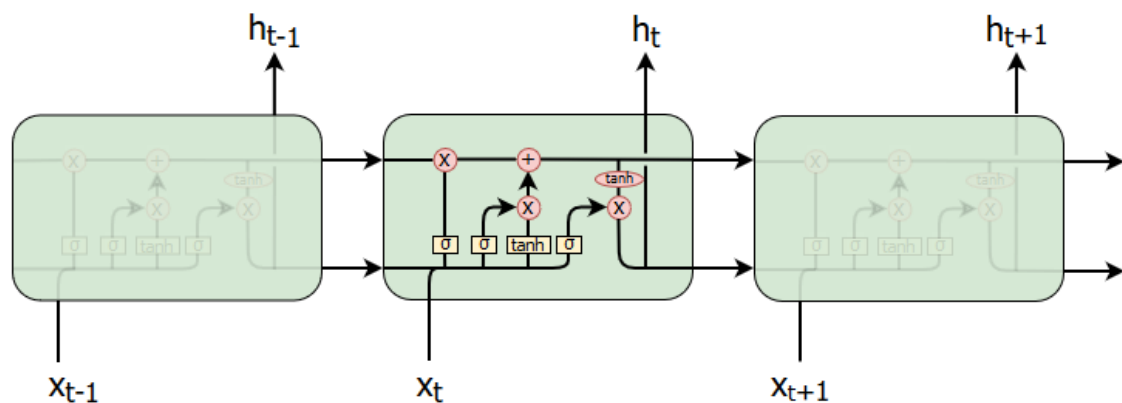


Slika 3.3: Ponavljajoča se celica preproste RNN. Povzeto po [17].

3.1 Celica LSTM (Long Short-Term Memory)

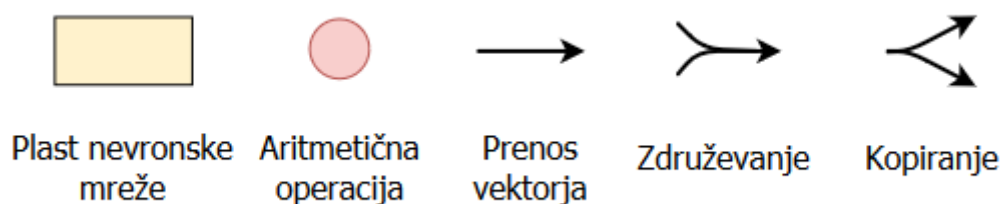
Nevronske mreže LSTM imajo prav tako verigi podobno sestavo, kar lahko vidimo na spodnji sliki. Vendar imajo posamezne celice bolj kompleksno strukturo. Namesto ene plasti ima celica LSTM kar štiri, ki med seboj komunicirajo na zelo poseben način [17].

Na sliki 3.4 vidimo kompleksnejšo sestavo celice LSTM v primerjavi z običajnimi celicami na sliki 3.3.



Slika 3.4: Ponavljajoča se celica LSTM. Povzeto po [17].

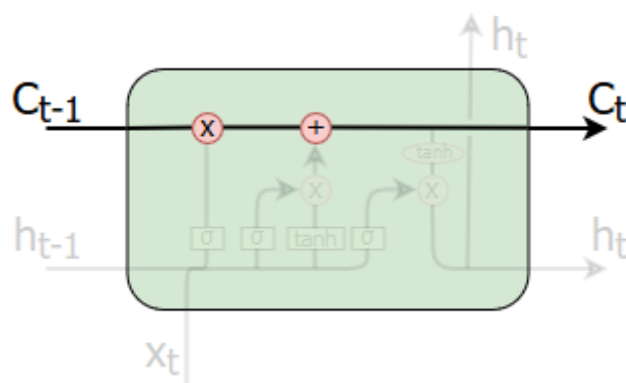
Slika 3.5 prikazuje notacijo, ki smo jo srečali že na slikah 3.3 in 3.4 in jo bomo srečevali še naprej, ko si bomo ogledali delovanje posameznih delov celice LSTM. Torej kvadrati predstavljajo plasti, krogi aritmetične operacije (seštevanje in množenje), črte s puščicami pa prenos vektorjev. Če se dve črti stikata pomeni, da se dva vektorja združita, če pa se črta razcepi pomeni, da se ustvari kopija vektorja.



Slika 3.5: Opis notacije. Povzeto po [17].

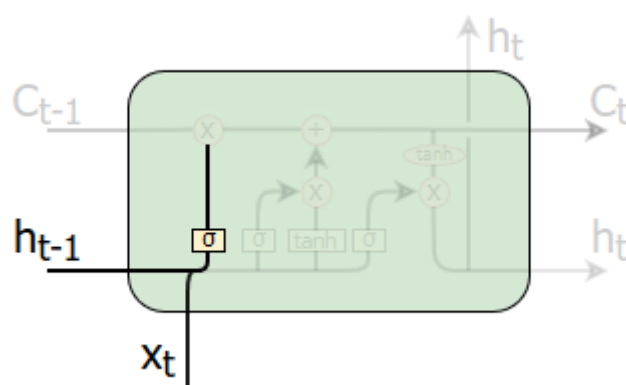
Posebnosti celice LSTM so stanje celice (angl. cell state) in različna vrata. Stanje celice služi prenosu pomembnih informacij med celicami od samega začetka do konca

procesiranja celotnega zaporedja. Lahko si ga predstavljamo kot spomin mreže, prikazuje pa ga slika 3.6. Vsaka celica ima sposobnost stanju celice dodati ali vzeti informacije, kar pa regulirajo strukture v sami celici, ki jim pravimo vrata [17].



Slika 3.6: Stanje celice LSTM. Povzeto po [17].

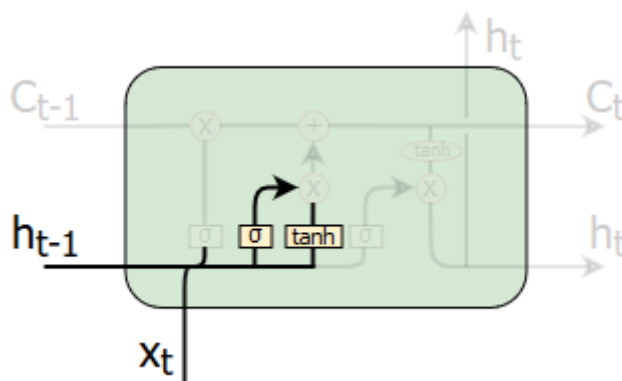
Na prvem koraku se moramo odločiti, katere informacije bomo zavrgli iz celičnega stanja. To določijo vrata za pozabljanje (angl. forget gate) s sigmoidno aktivacijsko funkcijo, ki jih vidimo na sliki 3.7. Glede na trenutni vhod in skrito stanje iz prejšnjega koraka vrnejo ta vrata vrednost med 0 in 1. Vrednost 1 pomeni, da informacijo v celoti obdržimo, vrednost 0 pa, da informacijo zavrnemo [17].



Slika 3.7: Vrata za pozabljanje. Povzeto po [17].

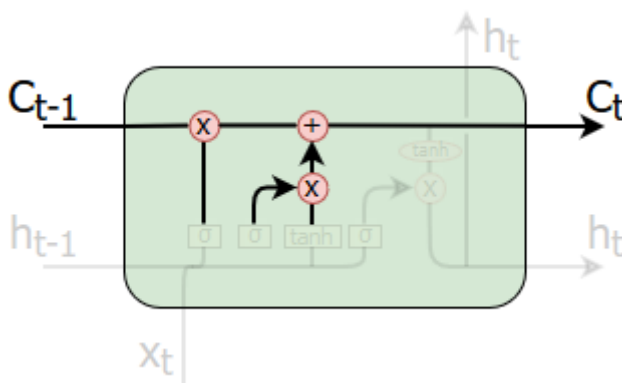
Na naslednjem koraku se moramo odločiti, katere nove informacije bomo shranili v stanje celice, to se zgodi v dveh delih. Vhodna vrata (angl. input gate), ki so prikazana na sliki 3.8, prav tako glede na trenutni vhod in skrito stanje (angl. hidden state) iz prejšnjega koraka najprej ponovno s sigmoidno funkcijo določijo, katere vrednosti bomo

posodobili. Nato s funkcijo \tanh ustvarijo vektor vrednosti na intervalu $[-1, 1]$, ki bi se lahko dodale v stanje celice. Izhodne vrednosti obeh funkcij med seboj pomnožimo, da dobimo skalirane vrednosti, ki jih uporabimo v posodabljanju stanja (Slika 3.9) [17].



Slika 3.8: Vhodna vrata. Povzeto po [17].

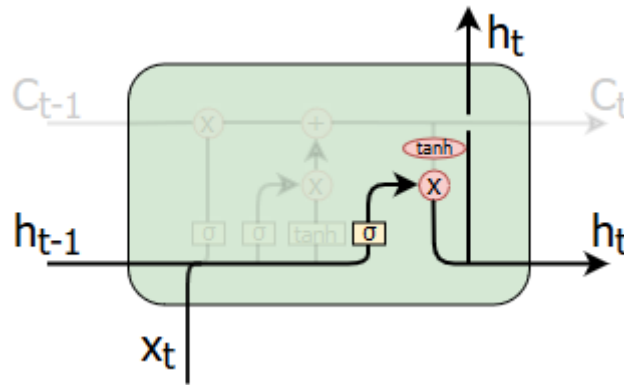
Sedaj imamo dovolj podatkov, da lahko stanje celice iz prejšnjega časovnega koraka posodobimo v trenutno stanje. Najprej staro stanje pomnožimo z izhodom vrat za pozabljanje, nato pa še prištejemo izhod vhodnih vrat in tako dobimo novo stanje, ki se prenese na naslednji časovni korak [17].



Slika 3.9: Posodabljanje stanja celice. Povzeto po [17].

Na koncu se moramo le še odločiti, kaj bo končni izhod celice. To storijo izhodna vrata (angl. output gate), prikazana na sliki 3.10. Izhodna vrata prejmejo ponovno enake vhodne podatke kot ostala vrata. S sigmoidno funkcijo najprej določijo, katere podatke celičnega stanja bomo poslali naprej. S funkcijo \tanh vrednosti stanja spravimo na

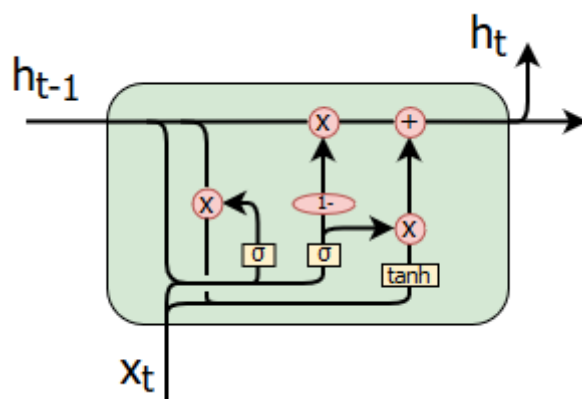
interval $[-1, 1]$ in jih pomnožimo z izhodom sigmoidne funkcije. Rezultat tega je novo skrito stanje, ki se prenese na naslednji časovni korak [17].



Slika 3.10: Izhodna vrata. Povzeto po [17].

3.2 Celica GRU (Gated Recurrent Unit)

Obstaja mnogo različic celic LSTM, vendar so razlike med njimi zelo majhne in ne bomo vseh posebej omenjali. Ogledali si bomo le LSTM-u zelo podobni, a veliko novejši tip celice, ki jo imenujemo celica GRU. Celotno strukturo celice GRU prikazuje slika 3.11. Vidimo, da struktura združuje prej omenjena vrata za pozabljanje in vhodna vrata v ena sama vrata za posodabljanje (angl. update gate). Tako ima celica GRU le dvojna vrata, vrata za ponastavljanje (angl. reset gate) in vrata za posodabljanje. Prav tako združi stanje celice in skrito stanje v eno skrito stanje, ki prenaša vse informacije [17] [20].

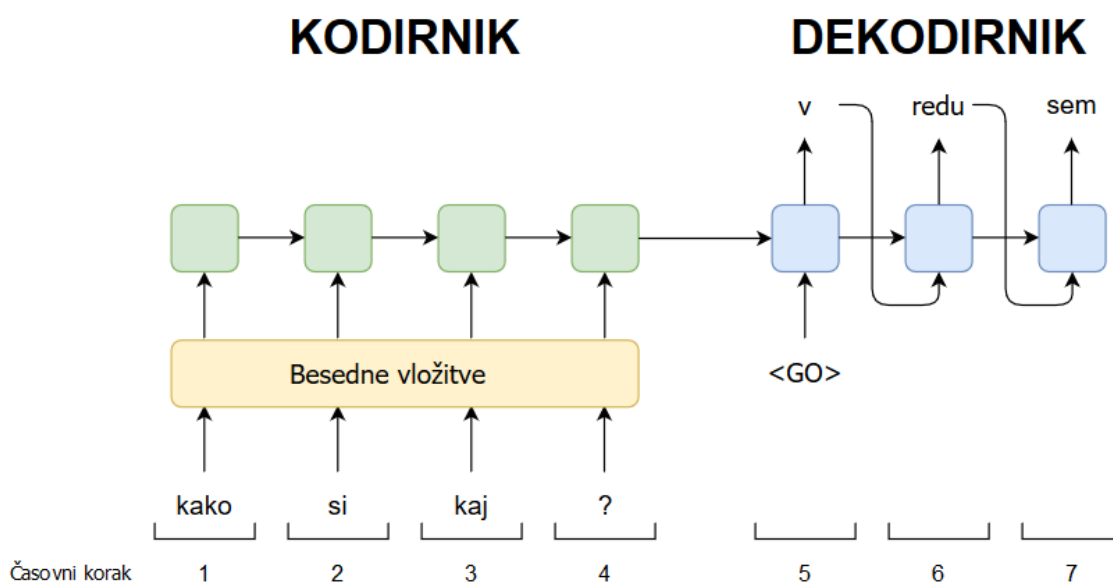


Slika 3.11: Celica GRU. Povzeto po [17].

Končni model celice je preprostejši kot LSTM in postaja vedno bolj popularen. Vsebuje manj računskih operacij zato jih lahko učimo hitreje, vendar pa ni jasnega zmagovalca med obema celicama. Strokovnjaki po navadi preizkusijo oba in ocenijo, kateri bolj ustreza njihovu problemu [20].

4 IMPLEMENTACIJA NEVRONSKE MREŽE

V poglavju o rekurentnih nevronske mrežah smo spoznali različne arhitekture mrež glede na vhode in izhode (Slika 3.2). Naš problem je v osnovi zelo podoben prevajanju povedi, ki smo ga omenili kot primer za model mnogo v mnogo (angl. many to many). V obeh primerih je vhod v mrežo poved in izhod prav tako poved. Prav za takšne probleme je zelo popularen in učinkovit model seq2seq (angl. sequence to sequence), ki temelji na arhitekturi kodirnik-dekodirnik (angl. encoder-decoder). Takšen model, prikazan na sliki 4.1, je sestavljen iz dveh rekurentnih nevronske mrež. Prva mreža imenovana kodirnik (angl. encoder) sprejme v našem primeru kot vhod zaporedje oziroma poved in na vsakem koraku procesira eno besedo. Njegova naloga je, da zaporedje besed spremeni v vektor fiksne velikosti, ki vsebuje le pomembne informacije o povedi. Kot smo spoznali v prejšnjem poglavju, rekurentne nevronske mreže prenašajo skrito stanje in stanje celice preko časovnih korakov. Izhod prve nevronske mreže je končno skrito stanje in si ga lahko predstavljamo kot povzetek celotne povedi. To stanje imenujemo miselni vektor (angl. thought vektor). Druga nevronska mreža, imenovana dekodirnik (angl. decoder), nam generira novo zaporedje besed, eno besedo na časovni korak, kjer besedo določi glede na miselni vektor in besedo, generirano v prejšnjem časovnem koraku [21].



Slika 4.1: Shema arhitekture kodirnik-dekodirnik. Povzeto po [21].

Na sliki 4.1 pri vходу v dekodirnik opazimo tudi poseben znak (angl. token) <GO>. To je poljubni znak, ki označuje začetek povedi in nam pride prav v fazi generiranja povedi. Omenili smo, da druga nevronska mreža novo besedo določi glede na miselni vektor in besedo, generirano v prejšnjem koraku. V prvem koraku takšne besede nimamo na voljo, zato vstavimo začetni znak <GO>. V našem modelu bomo uporabili začetni znak <START>, prav tako smo povedim dodali še končni znak <END>, ki pove, da smo s fazo generiranja povedi zaključili.

Nevronsko mrežo bomo implementirali in naučili v programskem jeziku Python [22] s pomočjo knjižnice Tensorflow [23] in njihove implementacije aplikacijskega programskega vmesnika Keras [24].

4.1 Priprava podatkovnih zbirk

Za učenje dialoga je potrebno veliko število pogovorov oziroma povedi. Odločili smo se, da bomo podatke pridobili iz podnapisov filmov. Povedi je potrebno procesirati tako, da bodo čim bolj enostavne za razumeti. Se pravi potrebno je odstraniti posebne znake in morebitne čudne besede. Podatkovne zbirke (angl. dataset) za učenje nevronske mreže smo pripravili v angleškem in slovenskem jeziku. Vse podatke smo predprocesirali v programskemu jeziku Python [22].

4.1.1 Pridobivanje slovenskih podatkov

S spleta (<https://www.podnapisi.net/>) smo naključno izbrali nekaj čez 30 modernejših filmov in pridobili njihove podnapise. Večinoma se zadnje čase podnapisi zapisujejo v datoteke s končnico ».srt«, ki pa jih lahko odpremo kot navadne tekstovne datoteke, njihova struktura, prikazana na sliki 4.2, je zelo enostavna za razumeti.

01.	94
02.	00:07:47,880 --> 00:07:51,520
03.	Spet motiš
04.	predavanje g. Morgana?
05.	
06.	95
07.	00:07:53,480 --> 00:07:55,760
08.	Povedala sem
09.	svoje mišljenje.

Slika 4.2: Slovenski podatki pred obdelavo.

Podnapisi, ki se nam prikažejo na zaslonu med gledanjem filma, so v datoteki v veliki večini primerov zapisani z več vrsticami. Prva prikazuje le zaporedno številko podnapisa, druga vrstica prikazuje časovno oznako, kdaj točno tekom filma naj se prikaže na zaslonu tekst in kdaj naj izgine. Zadnje vrstice pa prikazujejo dejansko besedilo (v večini primerov vidimo le eno ali dve vrstici hkrati, da ne prekrijemo preveč zaslona). Sledi še prazna vrstica, da je vse skupaj lažje berljivo, nato pa je naslednja skupina podatkov. Takšen zapis se ponovi več stokrat za celotni film.

Najprej smo ročno pregledali vsako izmed datotek in preverili, ali je struktura zapisa ustrezna. Hkrati smo izbrisali zapise, ki niso pogovori. Izbrisali smo podatke o igralcih v glavnih vlogah in ljudi, ki so pomagali pri snemanju filma. Ti podatki se pojavijo ali na začetku ali na koncu filma in so zapisani z velikimi tiskanimi črkami.

V programskem jeziku Python smo po vrsticah prebrali vsako datoteko s podnapisi. Vsako vrstico smo procesirali in zapisali v novo skupno datoteko, ki bo vsebovala našo podatkovno zbirko. Z vsako vrstico storimo naslednje:

- Nepotrebne vrstice, ki smo jih spoznali prej, in prazne vrstice odstranimo. To storimo tako, da v kodi preverimo, če vrstica vsebuje le eno številko, brez ločil ali nadaljnjih besed. Časovne oznake pa smo prepoznali po puščici med časovnima oznakama. Kadar smo prepoznali takšno vrstico, smo jo zavrgli.
- Posebno pozornost posvečamo končnim ločilom, da ločimo več povedi v eni vrstici in obratno.
- Iz vrstice odstranimo vse posebne znake in pretvorimo vse velike tiskane črke v majhne.

- Na koncu samo še odstranimo večkratne ponovitve presledkov in nepotrebne presledke na začetku ali koncu povedi.
- Vrstico pripnemo na konec datoteke.

Tako zapis podnapisov na sliki 4.2 pretvorimo v pogovorne izmenjave na sliki 4.3, kjer liho oštevilčene vrstice predstavljajo vprašanje, sode pa odgovor.

```
01. spet motiš predavanje g morgana
02. povedala sem svoje mišljenje
```

Slika 4.3: Slovenski podatki po obdelavi.

Na takšen način smo pridobili več kot petdeset tisoč povedi. Vse skupaj je bilo dokaj enostavno, a vseeno tukaj nastopi nekaj problemov. V učnih podatkih poskušamo simulirati dialoge, ki so tipa vprašanje-odgovor. Vendar je v kodi nemogoče vedeti kdaj, na primer, ena oseba zaporedoma pove dve povedi. Takšni primeri so slabi za učenje.

4.1.2 Pridobivanje angleških podatkov

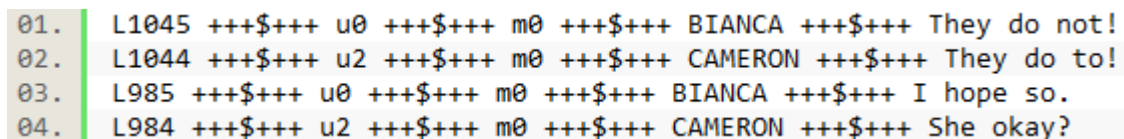
Zaradi večje razširjenosti angleškega jezika po svetu ni nič čudnega, da so nam na voljo boljši viri dialogov kot so podnapisi, ki jih lahko napiše in objavi kdorkoli. Za voljo doslednosti s slovenskimi podatki smo tudi tukaj pogovore pridobili iz filmov.

Na spletu je prosto dostopen korpus pogovorov, pridobljenih iz filmskih scenarijev. Vključuje preko dvesto dvajset tisoč pogovornih izmenjav iz kar 617 različnih filmov [25].

Prednost tega korpusa je, da vključuje dejanske pogovorne izmenjave med osebami iz filmov in tako se znebimo problema, na katerega smo naleteli pri slovenskih podatkih.

Ker se format zapisa korpusa, prikazanega na sliki 4.4, drastično razlikuje od zapisa podnapisov smo za predprocesiranje angleških podatkov uporabili drugačen pristop. Vsaka vrstica nam nudi pet podatkov, ločenih z oznako » +++\$+++ «. Za nas pa sta pravzaprav pomembna le prvi in zadnji podatek. Prva je številčna oznaka vrstice (črko L lahko odstranimo), zadnja pa poved. Številčna oznaka nam pomaga pri razdeljevanju vrstic v pogovorne izmenjave, če je razlika med dvema zaporednima oznakama enaka 1,

spadata v isti pogovor. Pogovorna izmenjava lahko vključuje več oseb in več kot le dve vrstici. Opazimo tudi, da ima vsaka naslednja vrstica nižjo oznako, kar nam da vedeti, da so povedi iz filmov v datoteko zapisane od spodaj navzgor.



```
01. L1045 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ They do not!  
02. L1044 +++$+++ u2 +++$+++ m0 +++$+++ CAMERON +++$+++ They do to!  
03. L985 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ I hope so.  
04. L984 +++$+++ u2 +++$+++ m0 +++$+++ CAMERON +++$+++ She okay?
```

Slika 4.4: Angleški podatki pred obdelavo.

Vrstice smo najprej grupirali v pogovorne izmenjave in obrnili vrstni red. Kadar je izmenjava vsebovala neparno število vrstic smo vrstico z najvišjo številčno oznako preprosto odstranili. Ko so bile vse vrstice smiselno urejene podobno kot pri slovenskih podatkih, smo povedi očistili. V vsaki povedi smo s pomočjo regularnih izrazov odpravili nekaj najpogostejših okrajšav besed v angleškem jeziku (na primer »I'm« razdelimo na »I am«) in odstranili vse posebne znake.

Na koncu smo vse velike tiskane črke spremenili v majhne in vse očiščene povedi zapisali v novo datoteko, kjer vsaka liha vrstica predstavlja vprašanja in naslednja sodo odgovor.

4.2 Tokenizacija povedi

V tem poglavju bomo spoznali, kako točno naše povedi spravimo v obliko, ki bo razumljiva za naš model, ki ga bomo zasnovali v nadaljevanju. Pripravljeni imamo dve zbirki podatkov, to so naše povedi v slovenskem in angleškem jeziku, seveda pa nevronske mreže operirajo nad števili, ne besedami. Tukaj pride v poštev tokenizacija (angl. tokenization) besed oziroma povedi. S tokenizacijo dosežemo, da je vsaka beseda predstavljena z edinstvenim indeksom. Tako lahko povedi zapišemo kot zaporedje celih števil.

Željeno število povedi najprej preberemo iz datoteke in jih razdelimo v dva seznama oziroma na vhode in izhode. Če začnemo šteti z ena, vsako liho vrstico pripnemo v seznam vhodov, vsako sodo pa v seznam izhodov. Hkrati pa vsaki lihi vrstici na začetek

povedi dodamo znak <START> in na konec znak <END>, saj sta ta znaka pomembna za delovanje dekodirnika.

Če želimo besede zapisati z indeksi, moramo ustvariti tako imenovani slovar besed, s pomočjo katerega bomo skozi delovanje programa lahko sledili, kateri indeks pripada kateri besedi in obratno. Izdelavo slovarja in zapis z indeksi smo izvedli s pomočjo razreda `Tokenizer` programskega orodja Keras (Slika 4.5).

```
from tensorflow.keras.preprocessing.text import Tokenizer

def create_vocab(tokenizer, max_vocab_size):
    word2idx = {} #Slovar.
    idx2word = {} #Obratni slovar.
    for k, v in tokenizer.word_index.items():
        if v < max_vocab_size:
            word2idx[k] = v #Primer: word2idx['ja'] = 1
            idx2word[v] = k #Primer: idx2word[1] = 'ja'
        else:
            break
    return word2idx, idx2word

VOCAB_SIZE = 15000 #Maksimalno število besed.
tokenizer = Tokenizer(num_words=VOCAB_SIZE) #Inicijalizacija razreda.
tokenizer.fit_on_texts(input_texts + output_texts) #Ustvari slovar.

#Vhode zapišemo z indeksi.
input_sequences = tokenizer.texts_to_sequences(input_texts)
#Izhode zapišemo z indeksi.
output_sequences = tokenizer.texts_to_sequences(output_texts)
```

Slika 4.5: Programska koda tokenizacije vhodnih in izhodnih podatkov.

Razred vsebuje dve metodi. Njuno uporabo vidimo na sliki 4.5. Pomembna je tudi spremenljivka `VOCAB_SIZE`, s katero omejujemo maksimalno število besed v slovarju. Prva metoda `fit_on_texts()` sprejme seznam povedi in iz njih izlušči vse različne besede, ki se pojavijo v povedih in vsaki dodeli edinstven indeks od 1 naprej. Indeks 0 je rezerviran za neznane besede in razširjanje povedi, kar bomo spoznali kasneje. Ko imamo generiran slovar, lahko kličemo naslednjo metodo `texts_to_sequences()`, ki ponovno vzame seznam povedi tipa črkovni niz (angl. string) in pretvori povedi v seznam celih števil (indeksov) s pomočjo prej generiranega slovarja. Pri inicializaciji razreda lahko podamo dodatni parameter (`num_words`), kar razred upošteva pri pretvorbi povedi v seznam celih števil, tako da uporabi le določeno število besed, ki se najpogosteje

pojavi v povedih. Z lastno funkcijo *create_vocab()* iz razreda *Tokenizer* izluščimo slovar z maksimalnim številom besed (*VOCAB_SIZE*) in ustvarimo obratni slovar. Obratni slovar potrebujemo, ker podatkovni tip slovarja (angl. *dictionary*) Python-a podpira le enosmerni dostop do shranjenih vrednosti.

4.3 Besedne vložitve

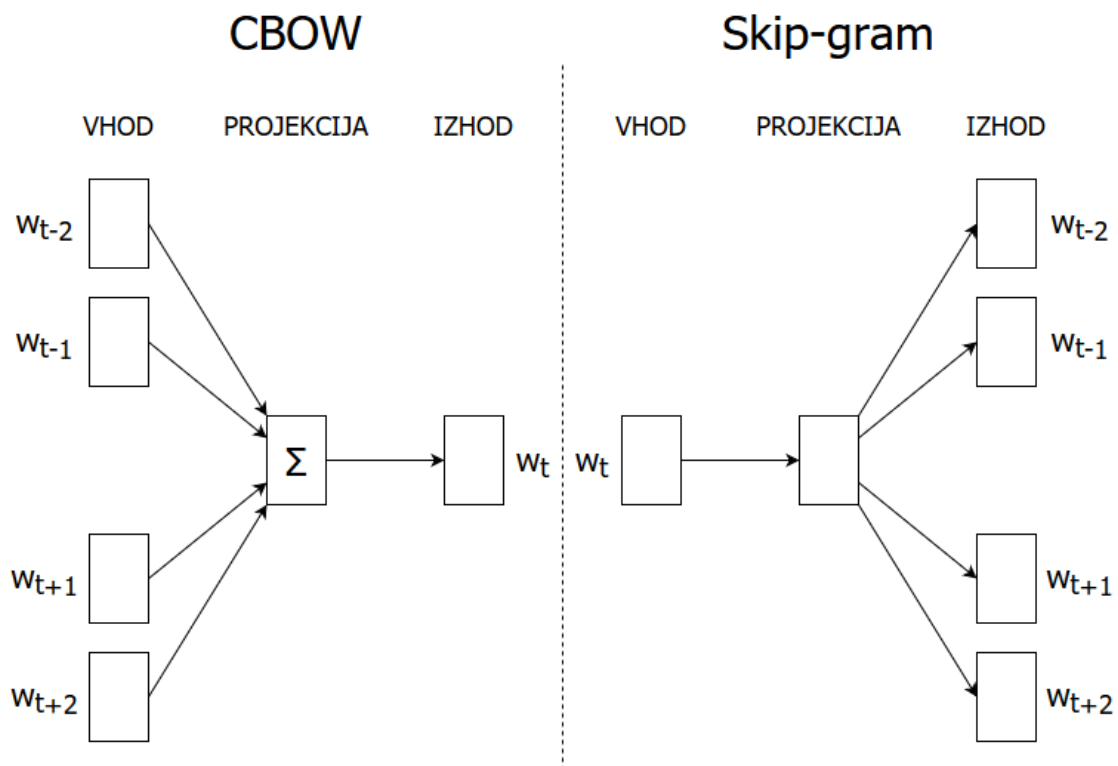
Besedne vložitve (angl. *word embeddings*) so razred tehnologij, ki se uporabljajo za zapis besed z vektorji realnih števil. Rezultat je *n*-dimenzionalni vektorski prostor, kjer vsak vektor predstavlja besedo na takšen način, da besede s sorodnim pomenom ali uporabo ležijo blizu druga drugi [26].

Pri učenju pogovornega robota smo za slovenske podatke uporabili knjižnico *Gensim Word2vec (word to vector)* [27] in z njo generirali svoj vektorski zapis besed. Za angleške podatke pa smo uporabili globalne vektorje za predstavo besed (angl. *Global Vectors for Word Representation – GloVe*) [28].

4.3.1 Word2Vec

Word2vec [29] je skupina dveh različnih modelov nevronske mreže, ki sta zelo učinkovita za numerično vektorizacijo besed iz velikega besednega korpusa. Razvili so jih pri podjetju Google leta 2013, da bi vektorizacijo na osnovi nevronske mreže naredili bolj učinkovito. Od takrat naprej so »de facto« standard za razvoj vnaprej treniranih besednih vložitev [26].

Oba modela delujeta na principu drsečega okenca poljubne velikosti. Slika 4.6 prikazuje razliko med njunim delovanjem. Prvi model CBOW (*Continuous Bag of Words*) se uči tako, da predvideva trenutno besedo glede na ostale besede, zajete v okencu. Drugi model imenovan Skip-Gram pa glede na trenutno besedo predvideva, katere besede jo obdajajo [26].



Slika 4.6: Primerjava delovanja obeh Word2Vec modelov. Povzeto po [26].

Na sliki 4.7 je prikazano učenje besednih vložitev z Word2Vec. Je zelo enostavno in potrebuje le par vrstic kode. V model vstavimo seznam povedi, kar v našem primeru preberemo kar iz datoteke, ki smo jo opisali v Poglavju 4.1.1. Izmed ostalih parametrov sta za nas pomembna še »size«, s katerim podamo velikost dimenzije vektorjev in parameter »window«, ki predstavlja velikost drsečega okna. Učenje modela je zelo hitro. Po učenju modela naučene besedne vektorje shranimo v tekstovno datoteko.

```
from gensim.models import Word2Vec

fileR = open('./DataSLO/Final.txt', 'r', encoding='utf-8')
lines = fileR.readlines() #Preberemo povedi iz podatkovne zbirke.
model = Word2Vec(lines,
                  min_count=3,
                  size=100,
                  workers=8,
                  window=3,
                  iter=15)
model.wv.save_word2vec_format('./DataSLO/embedding_SLO_100.txt',
                              binary=False)
fileR.close()
```

Slika 4.7: Programska koda primera uporabe Word2Vec.

4.3.2 GloVe

Algoritem GloVe (Global Vectors for Word Representation) [28] je nadgradnja Word2Vec za učenje besednih vektorjev. Razvili so ga na ameriški univerzi Stanford. Namesto drsečega okna za definicijo lokalnega pomena oziroma konteksta GloVe sestavi matriko besed in pomena z uporabo statistike celotnega korpusa. Rezultat tega je model, ki se lahko zelo dobro nauči vektorizacije besed [26].

Na spletu je prosto dostopnih več vnaprej treniranih besednih vložitev GloVe, shranjenih v obliki tekstovnih datotek. Za učenje pogovornega robota bomo uporabili datoteke, ki vsebuje čez štiristo tisoč besed, zapisanih z vektorjem dimenzije med 50 in 200 [28].

4.4 Vektorizacijska plast nevronske mreže

V našem modelu gredo vhodi in izhodi skozi vektorizacijsko plast (angl. embedding layer), ki zaporedje pozitivnih indeksov spremeni v vektorje fiksne velikosti. Plast lahko inicializiramo prazno in se uteži izračunajo tekom učenja, lahko pa uporabimo vnaprej naučene besedne vektorje in pospešimo učenje. Za uporabo Word2Vec ali GloVe besednih vložitev moramo ustvariti matriko vektorjev, ki jih preberemo iz tekstovne datoteke. To storimo tako, da za vsako besedo v slovarju poiščemo ustrezni vektor in ga zapišemo v matriko v vrstico, ki ustreza indeksu besede.

Ker uporabljamo isti slovar in iste besedne vložitve za vhode in izhode, lahko definiramo skupno vektorizacijsko plast, ki jo bomo v modelu uporabili dvakrat. Na sliki 4.8 vidimo implementacijo plasti, kjer so parametri:

- **Input_dim:** Število različnih indeksov, ki lahko pridejo v vektorizacijsko plast, to je število besed v slovarju + 1 (indeks 0).
- **Output_dim:** Dimenzija vektorjev na izhodu iz plasti.
- **Weights:** Matrika vektorjev, ki predstavlja začetne uteži plasti.
- **Trainable:** S tem parametrom povemo, ali se lahko uteži tekom učenja posodablajo ali ne.

```
embedding_layer = Embedding(input_dim=VOCAB_SIZE,
                             output_dim=embedd_size,
                             weights=[embedding_matrix],
                             trainable=True)
```

Slika 4.8: Programska koda implementacije vektorizacijske plasti.

4.5 Implementacija modela

Izdelava osnovnega modela kodirnik-dekodirnik seq2seq je dokaj preprosta, saj je celotna arhitektura že zelo izpopolnjena (Slika 4.9). Struktura modela, ki se uporablja za učenje, ni primerna za rekurzivne klice, zato ga najprej definiramo kot en model in ga po učenju razdelimo na dva sklepalna modela (angl. inference models), pri tem pa uporabimo naučene uteži iz prvotnega modela.

```
LATENT_DIM = 128 #Število skritih enot obeh mrež LSTM.
encoder_inputs = Input(shape=(None, ), dtype='int32',
                          name='encoder_inputs')
encoder_embedding = embedding_layer(encoder_inputs)
encoder_outputs, state_h, state_c = LSTM(units=LATENT_DIM,
                                          return_state=True,
                                          name='encoder_LSTM')
encoder_states = [state_h, state_c]

decoder_inputs = Input(shape=(None, ), dtype='int32',
                         name='decoder_inputs')
decoder_embedding = embedding_layer(decoder_inputs)
decoder_LSTM = LSTM(units=LATENT_DIM,
                    return_state=True,
                    return_sequences=True,
                    name='decoder_LSTM')
decoder_outputs, _, _ = decoder_LSTM(decoder_embedding,
                                     initial_state=encoder_states)
decoder_dense = Dense(VOCAB_SIZE, activation='softmax',
                      name='decoder_dense')
outputs = decoder_dense(decoder_outputs)

model = Model([encoder_inputs, decoder_inputs], outputs)
model.compile(loss='categorical_crossentropy',
              optimizer=Adam(),
              metrics=['accuracy'])
```

Slika 4.9: Programska koda implementacije kodirnik-dekodirnik plasti.

Slika 4.9 prikazuje implementacijo modela. Vsebuje dve vhodni plasti (eno za naša vprašanja in drugo za odgovore), eno skupno vektorizacijsko plast, dve plasti s celicami LSTM in na koncu še gosto povezano plast (angl. densely connected layer) z aktivacijsko

funkcijo softmax. Zadnja plast vsebuje toliko nevronov, kolikor imamo različnih besed v slovarju. S pomočjo aktivacijske funkcije softmax je izhod vsakega nevrona vrednost na intervalu [0, 1]. Seštevek vseh izhodnih vrednosti je enak 1, tako da lahko vrednosti interpretiramo kot verjetnosti za posamezne besede iz slovarja.

4.6 Učenje modela

Preden lahko začnemo učiti model, moramo naše podatke še nekoliko popraviti. Upoštevati moramo, da učne podatke v model pošiljamo v paketih (angl. batches), za pravilno delovanje mreže LSTM pa morajo biti vsi podatki iste dolžine. V našem primeru so povedi različnih dolžin. Seveda tukaj že govorimo o povedih, zapisanih kot zaporedje indeksov iz slovarja. To rešimo tako, da vse povedi razširimo do dolžine najdaljše povedi z večkratno uporabo indeksa 0. Za to v programskem orodju Keras uporabimo funkcijo *pad_sequences()*, proces pa imenujemo »sequence padding«.

Našo drugo mrežo (dekodirnik) poizkušamo naučiti, da glede na prejšnjo napove novo besedo, zato pripravimo še tretjo množico podatkov. Vsak odgovor v skupini zapišemo z bitno predstavitvijo (angl. one-hot encoded), kjer je vsaka beseda zapisana z binarnim vektorjem dolžine, ki je enaka velikosti našega slovarja. Ključnega pomena je, da vsak vektor zamaknemo za en časovni korak.

Zaradi velike količine podatkov je lahko poraba pomnilnika zelo visoka in lahko zelo hitro prekoračimo količine pomnilnika, ki ga imamo na voljo na svoji napravi. Da se temu izognemo, nam programski jezik Python in Keras API omogočata uporabo generatorjev. Generator je funkcija, ki tekom učenja podatke sproti dovaja v mrežo. Na sliki 4.10 je prikazana naša implementacija generatorja, ki povedi s prej omenjenimi metodami pripravi za učenje. Pred tem še vse podatke razdelimo na učno in testno množico. Testna množica vsebuje 20 odstotkov vseh podatkov. Velikost paketa (angl. batch size), to je število povedi v posamezni skupini, predstavlja spremenljivka `BATCH_SIZE`.

```

def generate_inputs(enc_sequences, dec_sequences):
    num_batches = len(enc_sequences) // BATCH_SIZE
    while True:
        for i in range(0, num_batches):
            start = i * BATCH_SIZE
            end = (i+1) * BATCH_SIZE
            enc_inp_data = pad_sequences(enc_sequences[start:end:1],
                                         maxlen=MAX_LEN,
                                         dtype='int32',
                                         padding='post',
                                         truncating='post')
            dec_inp_data = pad_sequences(dec_sequences[start:end:1],
                                         maxlen=MAX_LEN + 2,
                                         dtype='int32',
                                         padding='post',
                                         truncating='post')
            dec_out_data = np.zeros((BATCH_SIZE,
                                     MAX_LEN + 2,
                                     VOCAB_SIZE), dtype='float32')

            #Zanka za bitno predstavitev zaporedja.
            for j, seqs in enumerate(dec_inp_data):
                for k, seq in enumerate(seqs):
                    #Zamik za en časovni korak.
                    dec_out_data[j][k - 1][seq] = 1
            yield [enc_inp_data, dec_inp_data], dec_out_data

Xtrain, Xtest, Ytrain, Ytest = train_test_split(input_sequences,
                                                output_sequences,
                                                test_size=0.2)

train_gen = generate_inputs(Xtrain, Ytrain)
test_gen = generate_inputs(Xtest, Ytest)
train_gen_batches = len(Xtrain) // BATCH_SIZE
test_gen_batches = len(Xtest) // BATCH_SIZE

```

Slika 4.10: Programska koda generatorja učnih podatkov.

Učenje mreže poženemo s funkcijo, prikazano na sliki 4.11. V novejših različicah orodja Keras funkcija sama prepozna, kdaj so učni podatki podani v obliki generatorja. Funkcija prav tako na vsakem koraku (angl. epoch) beleži različne metrike, ki jih lahko podamo med implementacijo modela. S pomočjo teh metrik lahko preverjamo učinkovitost učenja modela.

```

history = model.fit(x=train_gen,
                    steps_per_epoch=train_gen_batches,
                    epochs=EPOCHS,
                    verbose=1,
                    validation_data=test_gen,
                    validation_steps=test_gen_batches)

```

Slika 4.11: Programska koda funkcije za učenje modela.

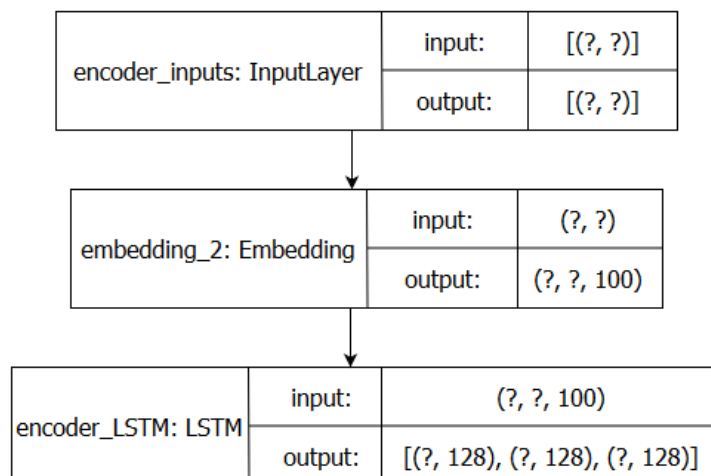
4.7 Uporaba modela

Naučeni model moramo za proces generiranja povedi razdeliti na kodirnik in dekodirnik. Slika 4.12 prikazuje, kako razdelimo prvotni model. Pri tem ponovno uporabimo iste spremenljivke, ki predstavljajo plasti, ki smo jih uporabili že pri implementaciji na sliki 4.9, tako da strukturo modelov spremenimo, uteži pa ostanejo enake.

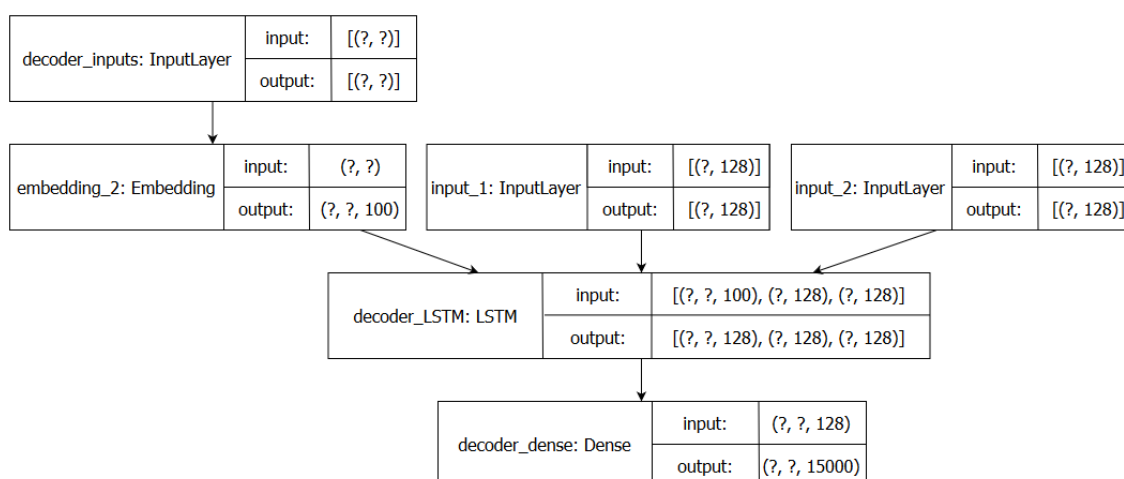
```
#Priprava kodirnika.  
encoder_model = Model(encoder_inputs, encoder_states)  
  
#Priprava dekodirnika.  
decoder_state_input_h = Input(shape=(LATENT_DIM, ))  
decoder_state_input_c = Input(shape=(LATENT_DIM, ))  
decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]  
decoder_outputs, state_h, state_c = decoder_LSTM(decoder_embedding,  
                                                  initial_state=decoder_states_inputs)  
  
decoder_states = [state_h, state_c]  
decoder_outputs = decoder_dense(decoder_outputs)  
decoder_model = Model([decoder_inputs] + decoder_states_inputs,  
                      [decoder_outputs] + decoder_states)
```

Slika 4.12: Programska koda za razdelitev modela na kodirnik in dekodirnik.

Slika 4.13 in slika 4.14 prikazujeta sheme obeh sklepalnih modelov. Sheme so generirane s pomočjo vmesnika Keras API in vključujejo tudi dimenzije vhodov in izhodov vseh plasti. Vprašaji predstavljajo v programskem jeziku Python rezervirano besedo »None«, ki smo jo uporabili na mestih, kjer dimenzije niso fiksne. Programsko orodje Keras jih tekom učenja samodejno priredi velikosti paketov in dolžini vhodov. Med fazo generiranja povedi pa nam to omogoča, da lahko v dekodirnik vstavimo le eno besedo naenkrat.



Slika 4.13: Shema kodirnika.



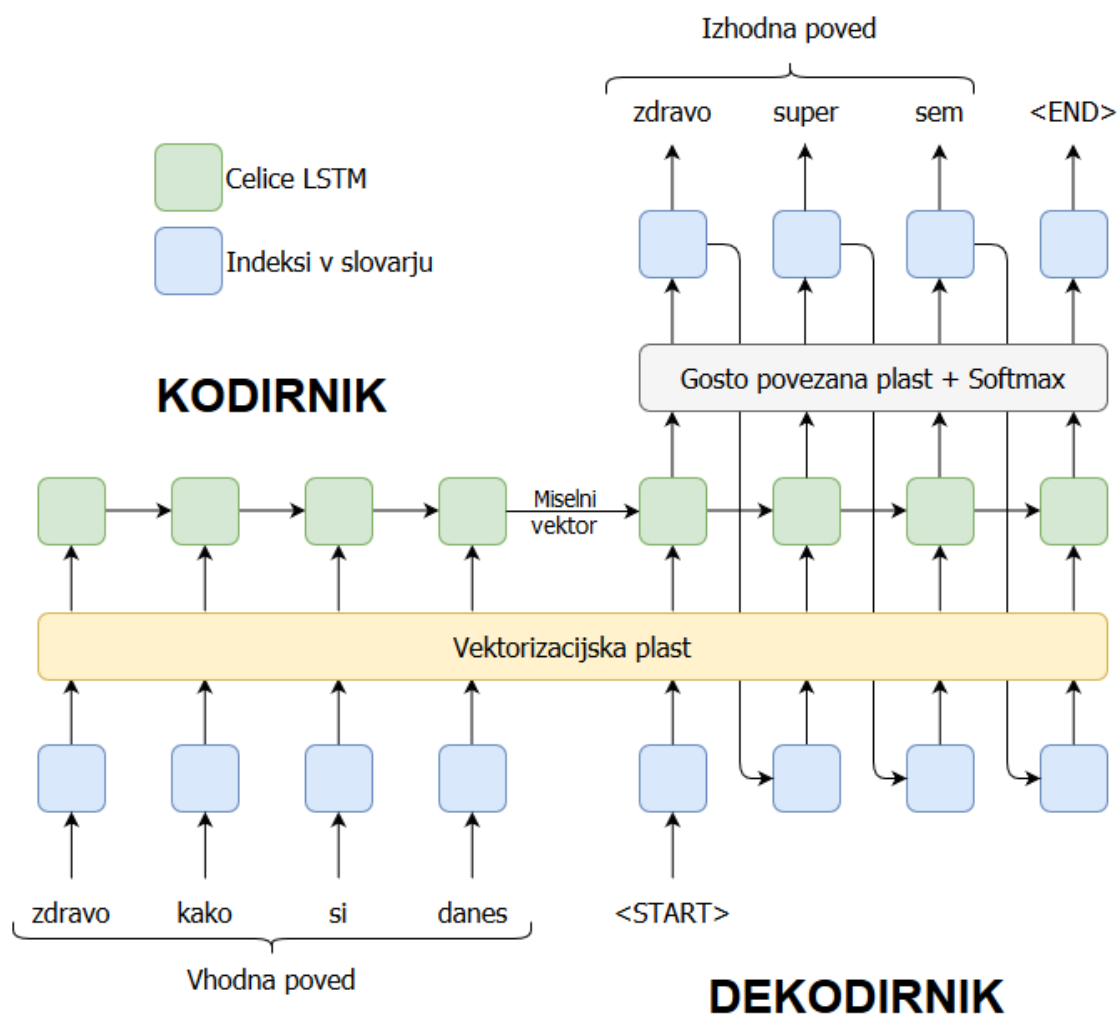
Slika 4.14: Shema dekodirnika.

Princip delovanja modela kodirnik-dekodirnik smo že spoznali. Sedaj ko imamo modela pripravljena, pa si lahko pogledamo postopek, kako z uporabo obeh sklepalnih modelov tvorimo povedi.

1. Poved najprej zapišemo kot zaporedje indeksov tako, da vsako besedo zapišemo z ustreznim indeksom iz slovarja. Pri tem pazimo na dolžino povedi, ki mora biti enaka kot v fazi učenja, zato po potrebi povedi razširimo (»sequence padding«).
2. Uporabimo kodirnik, da procesira poved in vrne miselni vektor oziroma povzetek povedi.

3. Dekodirnik ima tri vhode, dva od teh predstavljata miselni vektor kodirnika. Tretji vhod pa je trenutna beseda, s pomočjo katere napove naslednjo poved. Ker trenutno na prvem koraku še nimamo nobene besede, vstavimo v dekodirnik indeks začetnega znaka <START>.
4. Dekodirnik vrne seznam verjetnosti v velikosti slovarja. Izmed verjetnosti najdemo najvišjo in njen indeks povežemo z ustrezno besedo, ki jo dodamo na konec naše izhodne povedi.
5. Kot vhod v dekodirnik zdaj podamo trenutno besedo in njegov lastni miselni vektor.
6. Korake 3, 4 in 5 ponavljamo dokler ne prekoračimo omejene dolžine povedi ali pa je vrnjena beseda dekodirnika končni znak <END>.

Slika 4.15 prikazuje shemo naše implementacije arhitekture kodirnik-dekodirnik, ki jo bomo uporabili pri generiranju odziva pogovornega robota.



Slika 4.15: Shema naše implementacije arhitekture kodirnik-dekodirnik.

Slika 4.16 prikazuje, kako modele shranimo, da jih lahko kasneje že naučene uporabljamo v različnih ukaznih datotekah ali programih programskega jezika Python. Za pravilno delovanje modela moramo pri ponovni uporabi uporabljati isto dolžino povedi in iste slovarje, ki smo jih uporabili pri učenju, zato shranimo tudi te. Poleg dolžine povedi med podatke (datoteka *data.json*) shranimo trajanje učenja in podobne informacije, ki nam pomagajo ločiti modele med sabo.

```

#Shranjevanje slovarjev in informacij o modelu v JSON datoteke.
with open('./' + save_folder_path + '/data.json', 'w') as fp:
    json.dump(saveData, fp)
with open('./' + save_folder_path + '/dictionary.json', 'w') as fp:
    json.dump(word2idx, fp)
with open('./' + save_folder_path + '/revDictionary.json', 'w') as fp:
    json.dump(idx2word, fp)

#Shranjevanje Keras modelov.
encoder_model.save('./' + save_folder_path + '/encoder.h5')
decoder_model.save('./' + save_folder_path + '/decoder.h5')

```

Slika 4.16: Programska koda za shranjevanje modelov.

5 IZDELAVA APLIKACIJE

Cilj diplomskega dela je bil tudi izdelati mobilno aplikacijo, v kateri lahko uporabimo prej naučeno nevronske mreže in preizkusimo delovanje našega pogovornega robota. Sama aplikacija ni preveč kompleksna in imitira izgled tipičnih pogovornih platform, kot je Facebook messenger. Ključni deli takšne aplikacije so vnosno polje, kamor lahko uporabnik vnese svoje vprašanje in ločeni pogovorni oblaki (angl. speech bubble) za uporabnika in pogovornega robota.

5.1 Izdelava z ogrodjem Angular

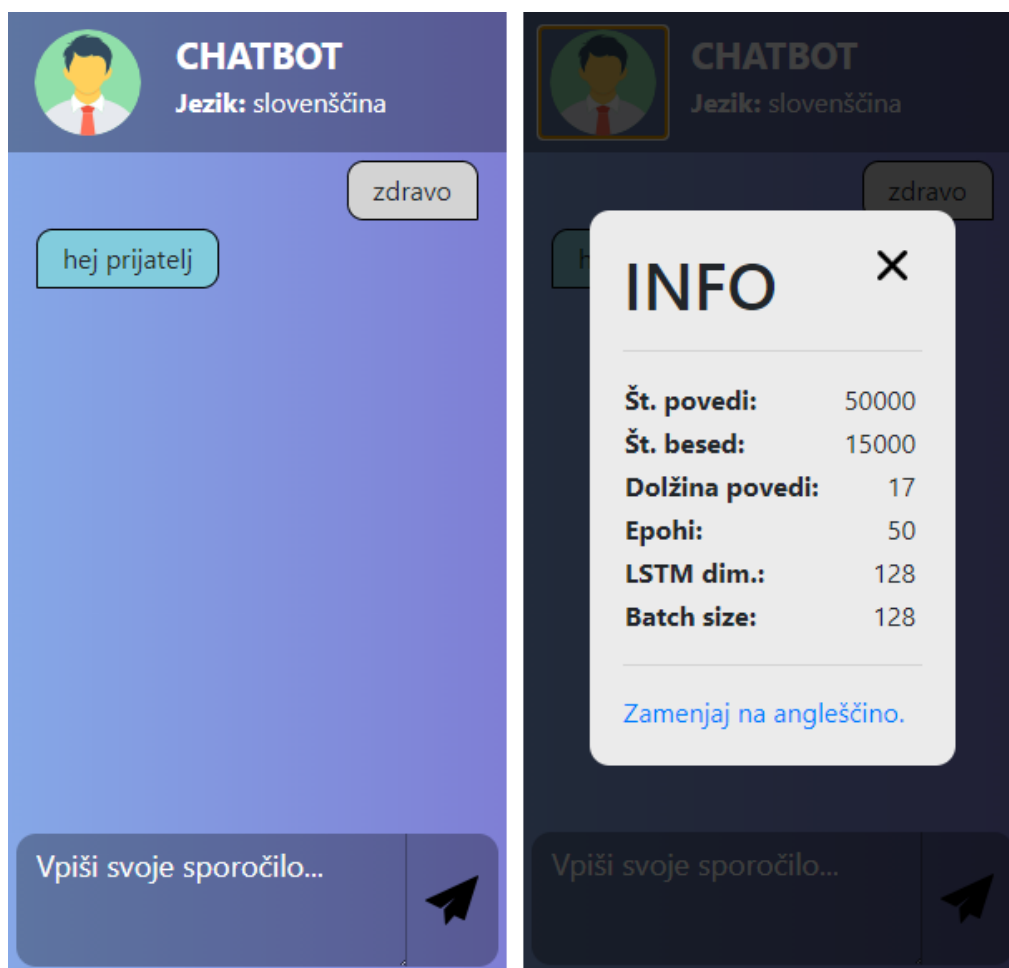
Za izdelavo mobilne aplikacije smo uporabili ogrodje Angular [30], ki je odprtokodno ogrodje za izdelavo mobilnih in spletnih aplikacij. Temelji na programskem jeziku TypeScript, za izgled aplikacije pa poskrbita jezik HTML (Hyper Text Markup Language) in v večini primerov shema CSS (Cascading Style Sheets), vendar omogoča tudi uporabo drugih stilov [30] [31].

Osnovni gradniki aplikacij Angular so moduli, tako imenovani NgModuli (angl. NgModules). Vsaka aplikacija ima vsaj en takšen modul, imenujemo ga korenski modul, ki poskrbi za zagon aplikacije. Moduli so sestavljeni iz komponent. Vsak modul ima vsaj korensko komponento. Komponente in storitve pa niso nič drugega kot razredi, označeni z dekoratorji, ki Angular-ju povedo, kako jih uporabiti. Moduli se lahko uporabljajo tudi znotraj drugih modulov. To, kar ponavadi imenujemo Angular knjižnice, so pravzaprav NgModuli [31].

Komponente predstavljajo to, kar vidimo na zaslonu. Medtem ko se uporabnik premika skozi aplikacijo, Angular sproti ustvarja, posodablja in uničuje komponente. Logika komponente je definirana v razredu, njen izgled pa v predlogi HTML. Za klepet z robotom smo za vsak jezik definirali svojo komponento, »chat-box« za slovenski in »chat-box-eng« za angleški. Komponenti sta si pravzaprav identični, le modele nevronske mreže in podatke nalagata iz različnih imenikov. Takšen pristop smo izbrali, ker omogoči zelo enostavno preklapljanje med modeli nevronske mreže [31].

Storitve so preprosti razredi, ki jih lahko kličemo iz več komponent. Njihov namen je, da nam pomagajo aplikacijo razdeliti na manjše, različne logične enote, ki jih večkrat ponovno uporabimo. V naši aplikaciji smo dodali storitev v obliki modalnega okna, ki smo ga uporabili v obeh komponentah za prikaz podatkov o modelu ter preklapljanje med obema jezikoma [31].

Na sliki 5.1 je prikazan končni izgled aplikacije. Vsebuje vnosno polje, kamor vnesemo svoje sporočilo, ki se potem prikaže v sivem oblačku na desni strani pogovora. Odgovor pogovornega robota je prikazan v svetlo modrem oblačku na levi strani. Ob kliku na sličico v levem zgornjem kotu aplikacije se odpre modalno okno s prikazanimi informacijami o modelu.



Slika 5.1: Končni izgled aplikacije izdelane v ogrodju Angular.

Datoteke JSON (JavaScript Object Notation), ki smo jih generirali in shranili v jeziku Python enostavno kopiramo v imenik našega projekta Angular in jih uvozimo v TypeScript, kot vidimo na sliki 5.2. Za upravljanje z modeli smo uporabili različico knjižnice Tensorflow, namenjene za programski jezik JavaScript in TypeScript, imenovane TensorflowJS. Za nalaganje modelov nam TensorFlowJS ponuja poseben pretvornik in funkcije, ki modele preberejo iz datoteke JSON v objekte, s katerimi lahko upravljamo zelo podobno kot v jeziku Python.

```
import { Component, OnInit } from '@angular/core';
import { NgbModule, NgbModal, NgbActiveModal } from '@ng-bootstrap/ng-bootstrap';
import data from '../../assets/Models/SLO/data.json';
import dict from '../../assets/Models/SLO/dictionary.json';
import revDict from '../../assets/Models/SLO/revDictionary.json';
import * as tf from '@tensorflow/tfjs';
import { fetch as fetchPolyfill } from 'whatwg-fetch';
import { ModalService } from '../_modal';
```

Slika 5.2: Uvoz knjižnic in datotek JSON v aplikacijo.

TensorflowJS pretvori modele v datoteke JSON, ki vsebujejo le opis arhitekture in plasti modela, uteži plasti pa so shranjene v več binarnih datotek, ki se morajo nahajati v istem imeniku kot datoteka JSON. Zaradi tega modelov ne moremo tako enostavno uvoziti kot slovarje. To smo rešili z modulom »*whatwg-fetch*«, ki nam omogoča, da v ogrodju Angular uporabimo nalaganje datotek preko relativnih poti znotraj aplikacije. Uvoz modula vidimo na sliki 5.2, uporabo modula in nalaganje modelov pa na sliki 5.3.

```
async loadModel() {
  window.fetch = fetchPolyfill;
  this.encoder = await tf.loadLayersModel('assets/Models/SLO/Encoder_tf/model.json');
  console.log('Encoder loaded...');
  this.decoder = await tf.loadLayersModel('assets/Models/SLO/Decoder_tf/model.json');
  console.log('Decoder loaded...');
}
```

Slika 5.3: Programska koda za nalaganje modelov iz datotek JSON.

Slika 5.4 prikazuje programsko kodo v TypeScript za generiranje povedi oziroma odziva pogovornega robota z uporabo obeh sklepalnih modelov. Postopek smo opisali v Poglavju 4.7.

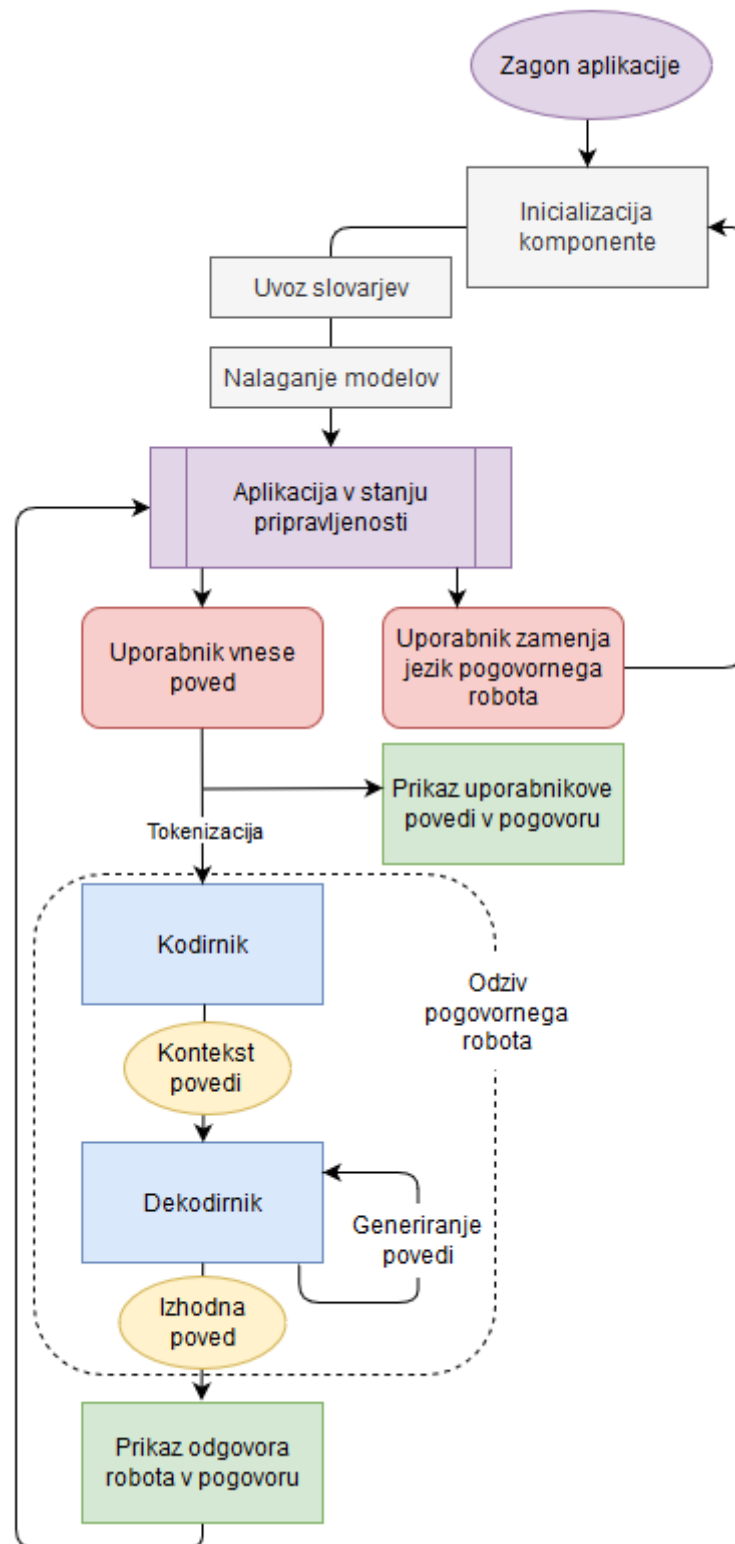
```

botReply(input_seq) {
  let states_value = this.encoder.predict(input_seq) as tf.Tensor[]; //Uporaba Encoderja.
  let target_seq = tf.buffer<tf.Rank.R2>([1, 1]); //Trenutna beseda.
  target_seq.set(dict['<START>'], 0, 0); //Na prvem koraku znak <START>
  let stop_condition = false;
  let decoded_sentence = '';
  let word_count = 1;
  while (!stop_condition) {
    const [output_tokens, h, c] = this.decoder.predict([target_seq.toTensor(), ...states_value]) as
    [tf.Tensor<tf.Rank.R2>, //Uporaba Decoderja
     tf.Tensor<tf.Rank.R2>, //Metoda predict enako kot v Python-u
     tf.Tensor<tf.Rank.R2>];
    const sampledTokenIndex = //Najdemo indeks najvišje verjetnosti.
      output_tokens.squeeze().argMax(-1).arraySync() as number;
    let sampledWord = revDict[sampledTokenIndex];
    if (sampledWord != '<START>' && sampledWord != '<END>') {
      decoded_sentence += sampledWord + ' ';
      word_count++;
    }
    if (sampledWord == '<END>' || word_count > data['dec_max_length']) {
      stop_condition = true;
    }
    target_seq = tf.buffer<tf.Rank.R2>([1, 1]);
    target_seq.set(sampledTokenIndex, 0, 0); //Posodobimo trenutno besedo.
    states_value = [h, c] //Posodobimo miselni vektor.
  }
  return decoded_sentence
}

```

Slika 5.4: Programska koda za generiranje odziva pogovornega robota.

Slika 5.5 prikazuje shemo delovnega toka (angl. workflow) v naši aplikaciji. Na shemi opazimo, da se programska koda za uvoz slovarjev (Slika 5.2) in nalaganje modelov (Slika 5.3) uporabi ob zagonu aplikacije in vsakič, kadar uporabnik zamenja jezik modela. Pri tem pride tudi do izbrisa trenutnega pogovora. Shema prikazuje tudi posplošen prikaz uporabe kodirnika in dekodirnika za generiranje odziva pogovornega robota, prikazanega na sliki 5.4, ki pa se ponovno uporabi po vsakem novem vnosu uporabnika.



Slika 5.5: Shema delovnega toka v aplikaciji.

5.2 Pretvorba v mobilno aplikacijo

Aplikacijo, izdelano v ogrodju Angular, poganjamo v brskalniškem okolju. V kolikor želimo, jo lahko zelo preprosto izvozimo kot mobilno aplikacijo s pomočjo orodja Apache Cordove [32], ki je odprto kodno ogrodje za izdelavo mobilnih aplikacij. Je tudi zelo priročno orodje za razvijalce, ki želijo svojo že izdelano aplikacijo razširiti preko več platform.

Za izvoz aplikacije potrebujemo poleg orodja Apache Cordove še razvijalski paket (angl. Software Development Kit - SDK) za platformo, na katero želimo izvoziti aplikacijo. V našem primeru je to Android. Android SDK za svoje delovanje potrebuje še Javansko razvojno okolje (angl. Java Development Kit - JDK). Postopek izvoza je zelo enostaven in hiter, večino časa porabimo za samo namestitev ustrezne programske opreme.

Po namestitvi potrebne programske opreme smo pripravljeni na izvoz aplikacije. To storimo tako, da v naš projekt v orodju Angular dodamo elemente projekta Cordova. Najprej ustvarimo nov prazen projekt Cordova in mu z ukazom »*cordova add platform Android*« dodamo podporo za platformo, na katero bomo namestili aplikacijo. Novo ustvarjeni projekt Cordova združimo z našim projektom v orodju Angular. To storimo tako, da v orodju Angular v projekt prenesemo vse datoteke razen datotek *package.json*, *package-lock.json* in imenika *node_modules*. Datoteki *package.json* združimo. Priporočljivo je, da to storimo ročno in prepisujemo vrstico po vrstico. Tukaj lahko tudi spremenimo nekaj parametrov, na primer ime avtorja in ime aplikacije. Angular privzeto zgradi projekt v imeniku »*dist*«, Cordova pa v »*www*«. To lahko po želji tudi poenotimo v konfiguracijskih datotekah. Za enkratni izvoz pa datoteke orodja Angular enostavno prekopiramo v imenik »*www*«.

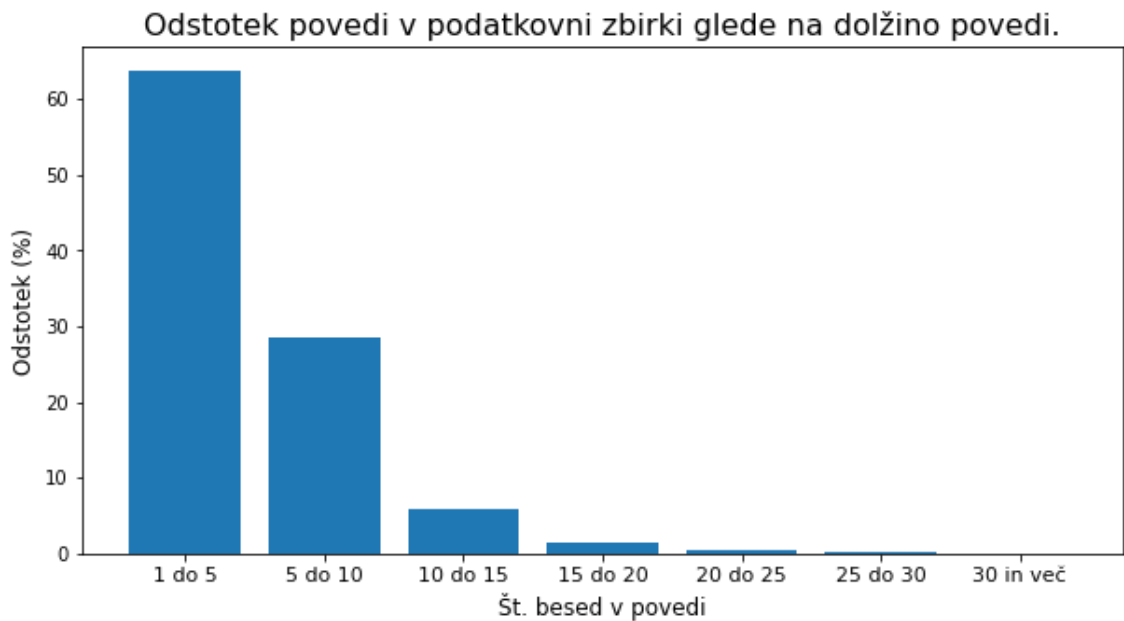
Aplikacijo sedaj le še zgradimo z ukazom »*cordova build Android*«. V kolikor imamo priključen mobilni telefon, ki ima omogočen način za razvijalce, lahko z ukazom »*cordova run android --device*« prenesemo aplikacijo na mobilni telefon.

6 REZULTATI

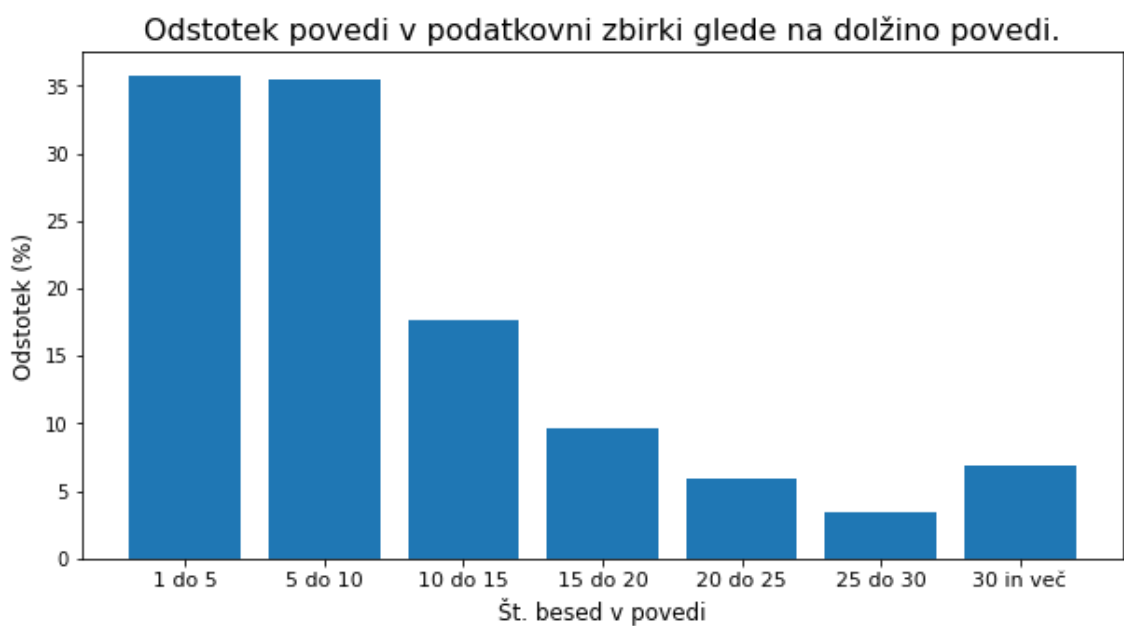
Testiranja so potekala na centralni procesni enoti AMD Ryzen 3700x, grafični procesni enoti Nvidia GeForce RTX 2070 Super in na računalniku z 16 GB pomnilniškega prostora.

V Poglavju 4.6 smo spoznali, da za učenje nevronske mreže LSTM potrebujemo povedi iste dolžine, kar smo dosegli s tako imenovanim »sequence paddingom«. Vse povedi smo z dodajanjem indeksa 0 priredili na dolžino najdaljše povedi, kar pa ni najbolj optimalen pristop. Angleška zbirka je zelo obsežna in vsebuje preko dvesto sedemdeset tisoč povedi. Zato smo jih za statistiko uporabili le prvih petdeset tisoč, kar je približno enako število povedi kot v slovenski zbirki. Dolžine povedi smo dodatno preučili in ugotovili, da najdaljša poved v slovenski zbirki podatkov vsebuje 61 besed, v angleški pa kar 277 besed. V povprečju slovenske povedi vsebujejo 4 besede, angleške pa 11. Prirejanje povedi na maksimalno dolžino bi tako pomenilo zelo slab izkoristek pomnilniškega prostora, ki nam je na voljo.

Grafa na slikah 6.1 in 6.2 predstavljata odstotne deleže povedi različnih dolžin. Razvidno je, da slovenska zbirka vsebuje zelo kratke povedi, zato bi večina povedi vsebovala ogromno ničelnih indeksov. Angleška zbirka sicer vsebuje precej enakomerno porazdelitev povedi, vendar vsebuje veliko večje odstopanje najdaljše povedi od povprečja, zato je prav tako potrebno omejiti dolžino povedi.



Slika 6.1: Graf odstotkov povedi glede na dolžine povedi v slovenski podatkovni zbirki.



Slika 6.2: Graf odstotkov povedi glede na dolžine povedi v angleški podatkovni zbirki.

Mreža LSTM se tekom učenja nauči pomena indeksa 0 in znaka <END> na koncu povedi, zato omejevanje dolžine povedi nima vpliva na končno natančnost mreže. Večja maksimalna dolžina prinese le večjo porabo pomnilnika in počasnejše učenje. Da smo se izognili prekoračitvi pomnilnika ter povečali hitrosti učenja, smo pri učenju modelov

omejili maksimalno dolžino povedi za slovenski jezik na 15 besed za angleški jezik pa na 30 besed.

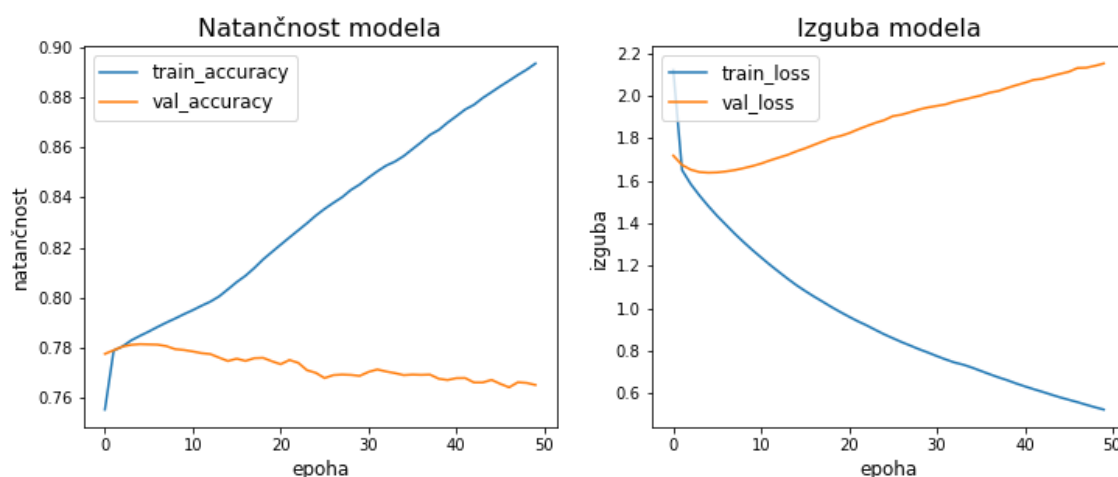
Porabo pomnilnika smo zelo omejili z uporabo generatorja, opisanega v Poglavju 4.6 in optimizacijo programske kode, zato nas pravzaprav poraba ne ovira. Večji problem, ki ga prinesejo daljše povedi, je trajanje učenja. Tega smo pohitrili z uporabo grafične procesne enote za učenje nevronske mreže. V povprečju je bilo učenje z grafično procesno enoto kar petkrat hitrejše kot s centralno procesno enoto.

Učenje pogovornega robota je lahko zelo nepredvidljivo, saj metrike natančnosti med učenjem modela niso dober pokazatelj smiselnosti odgovorov robota. Nekatera področja, ki uporabljajo podobno arhitekturo modela (na primer strojno prevajanje povedi), lahko vseeno primerjajo, kako uspešno je model prevedel poved [21]. V primeru pogovornih robotov, zlasti takšnih, ki so naučeni na velikih količinah podatkov, pa je smiselnost odgovora subjektivno mnenje posameznika. Zato smo tudi mi po učenju vsak model preizkusili na primeru pogovora. Kljub temu smo bili pri učenju modela pozorni na običajne metrike natančnosti, ki se merijo na učni in testni množici med učenjem. Funkcija s slike 4.11 nam je med učenjem vračala naslednje podatke:

- **Train_accuracy:** natančnost napovedovanja na učni množici, predstavljena z vrednostjo na intervalu [0, 1]. Višja kot je vrednost, bolj natančen je model.
- **Train_loss:** izguba pri napovedovanju učne množice, predstavljena z vrednostjo višjo od 0. Nižja kot je vrednost, manjša je izguba.
- **Val_accuracy:** natančnost napovedovanja na testni množici, predstavljena z vrednostjo na intervalu [0, 1]. Višja kot je vrednost, bolj natančen je model.
- **Val_loss:** izguba pri napovedovanju testne množice, predstavljena z vrednostjo višjo od 0. Nižja kot je vrednost, manjša je izguba.

V nadaljevanju si bomo ogledali, kako različna velikost paketa in količina podatkovne zbirke vplivata na rezultate obeh modelov. Prvi model smo naučili na slovenskih podatkih. Učenje je trajalo 50 epoh z velikostjo paketa 32. Za optimizacijo stopnje učenja (angl. learning rate) smo uporabili Kerasovo implementacijo algoritma Adam [33], ki je zelo popularen pri učenju modelov s podobno arhitekturo.

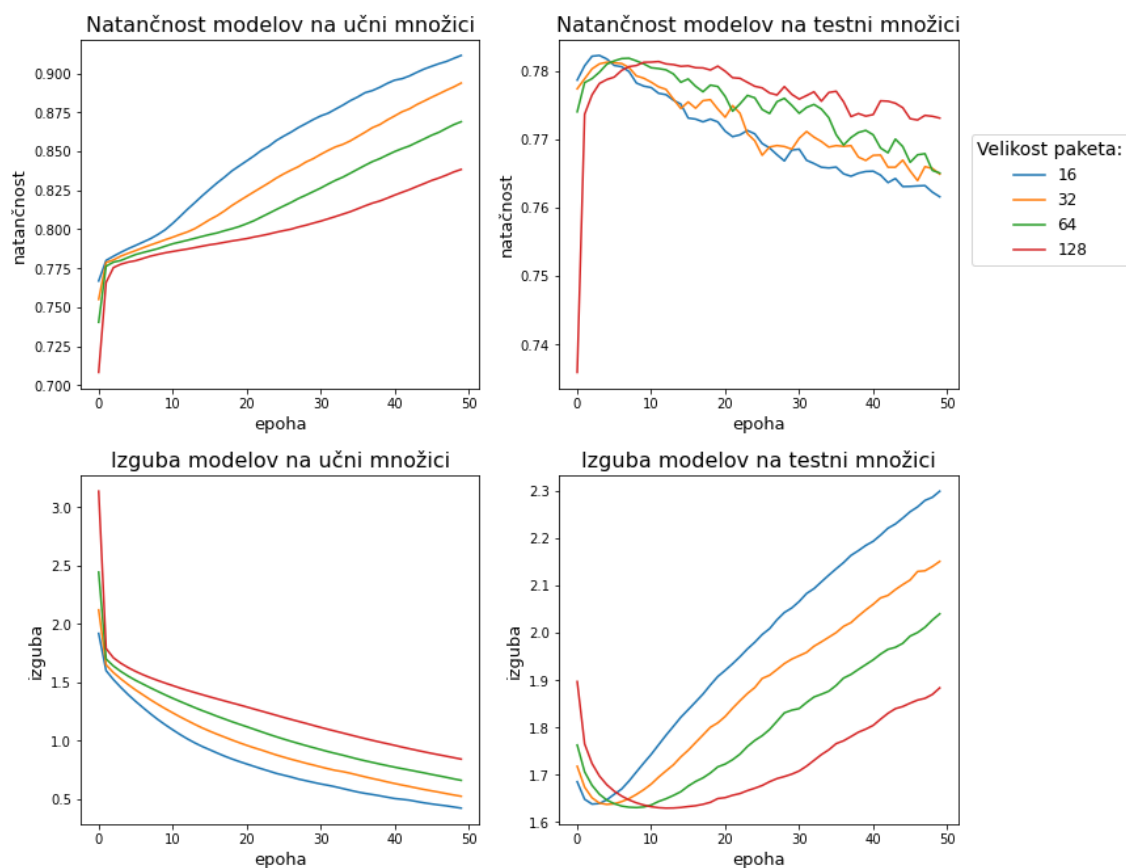
Pri ogledu grafov na sliki 6.3 opazimo, da pri učenju modela prihaja do tako imenovanega prekomernega prileganja (angl. overfitting), ki ga prepoznamo po naraščanju izgube pri napovedovanju na testni množici (metrika val_loss), izrazitemu naraščanju natančnosti napovedovanja na učni množici (metrika train_accuracy) in istočasnemu padanju natančnosti napovedovanja na testni množici (metrika val_accuracy).



Slika 6.3: Grafa natančnosti in izgube prvega modela.

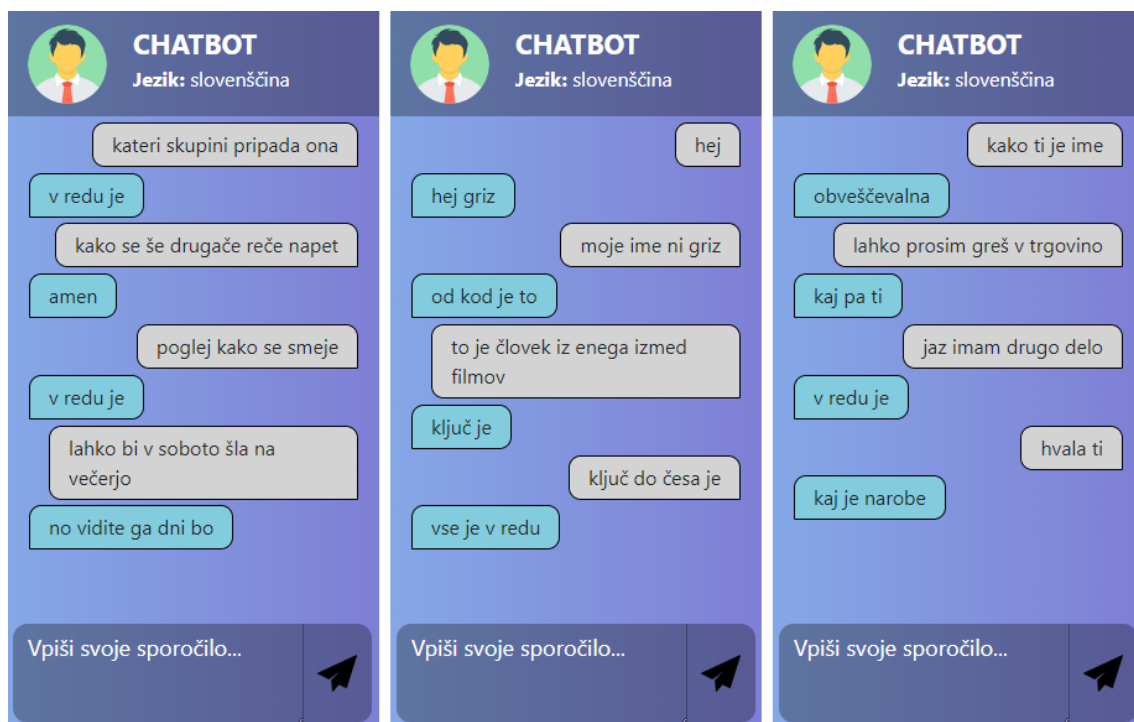
Prekomerno prileganje pomeni, da si model shrani učne podatke in se hkrati zelo slabo odziva na nove oziroma testne podatke. K zmanjševanju prekomernega prileganja pripomore uporaba večje velikosti paketa. Večja kot je velikost paketa, manjkrat se v eni epohi posodobijo uteži modela in manjkrat se izračunata natančnost in izguba. Učenje zato poteka počasneje, a bolj enakomerno, hkrati pa tako v testno množico zajamemo več podatkov. Posledično je manjša verjetnost, da množica vsebuje več slabih povedi.

Grafi na sliki 6.4 prikazujejo kako večja velikost paketa zakasni pojavitev prekomernega prileganja, seveda pa z naraščanjem velikosti skupin narašča tudi količina zasedenega pomnilnika. Prav tako opazimo, da bi morali pri večji velikosti paketa učiti z več epohami, da bi dosegli isti odstotek natančnosti na učni množici.

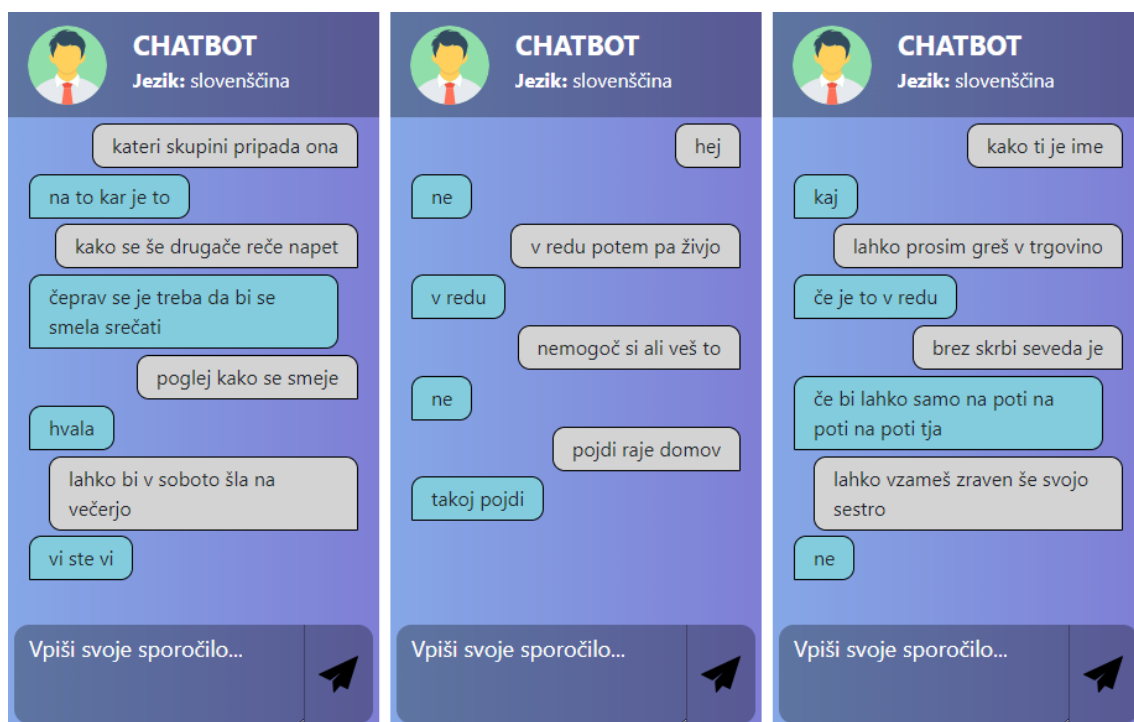


Slika 6.4: Prikaz učinka različnih velikosti paketa na potek učenja.

Slika 6.5 in 6.6 prikazujeta primere pogovorov z modelom, naučenim na slovenskih podatkih z uporabo različnih velikosti paketa (32 in 128). Njune krivulje natančnosti in izgube prikazujeta grafa na sliki 6.4. Glede na metrike naj bi se model z velikostjo paketa 32 bolje naučil primerov v učni množici, kar se pokaže pri odgovoru na pozdrav »hej«. Drugi model pogosteje odgovarja s krajšimi posplošenimi odgovori, razen tega pa ni opaznih sprememb.

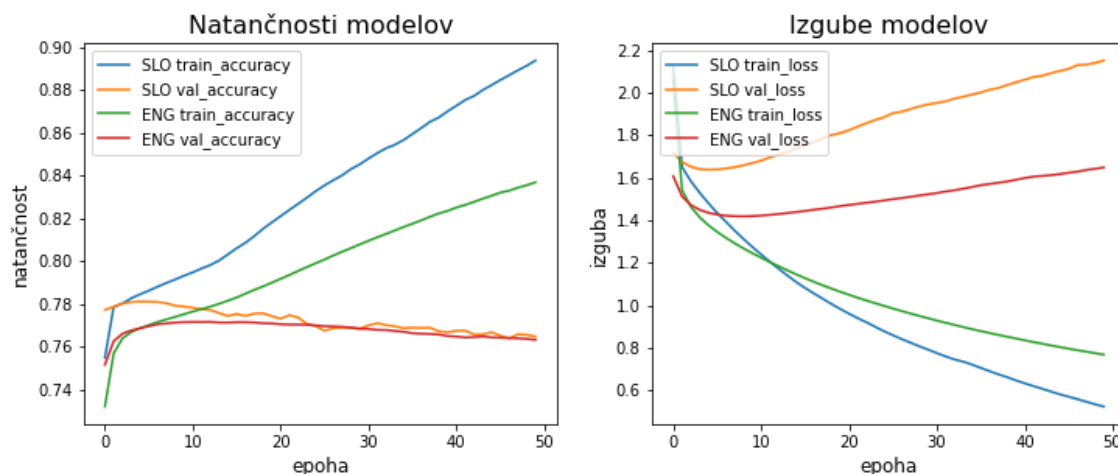


Slika 6.5: Primeri slovenskih pogovorov z velikostjo paketa 32.



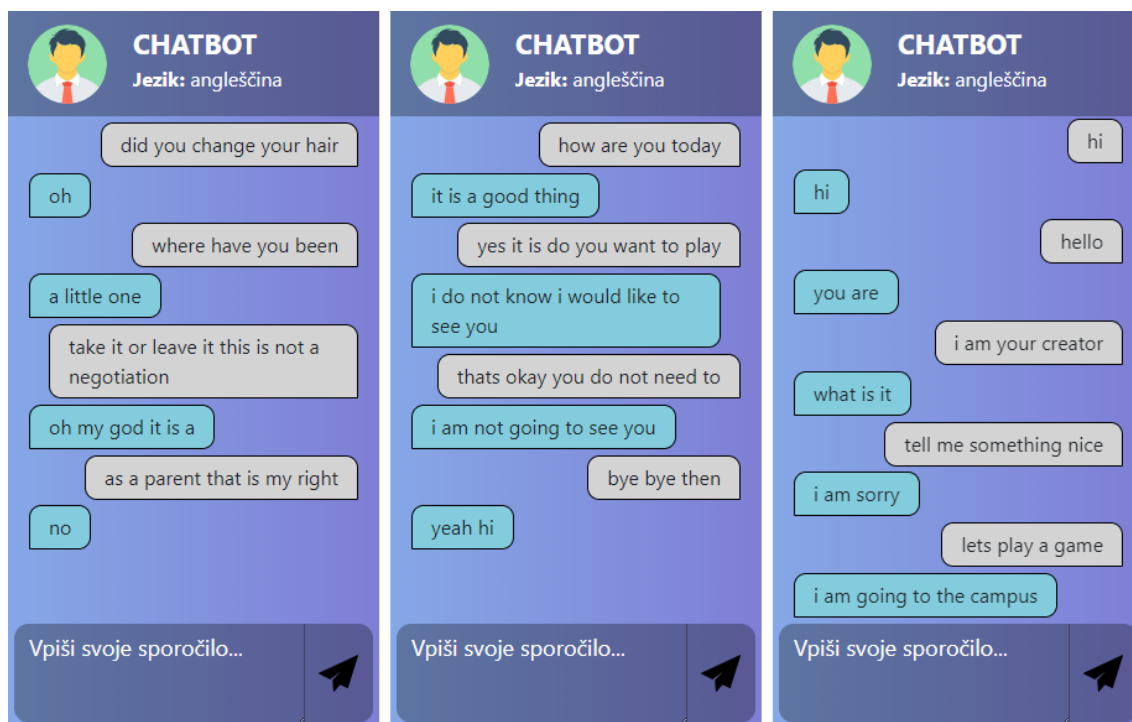
Slika 6.6: Primeri slovenskih pogovorov z velikostjo paketa 128.

Grafa na sliki 6.7 prikazujeta razliko metrik med modeloma obeh različnih jezikov. Opazimo manj šuma pri testni natančnosti učenja na angleških podatkih in hkrati manjša odstopanja med obema metrikama natančnosti in izgube, kar je posledica daljših in urejenih pogovorov v podatkovni zbirki in uporabe boljše naučenih besednih vložitev.



Slika 6.7: Grafa primerjave slovenskega in angleškega modela.

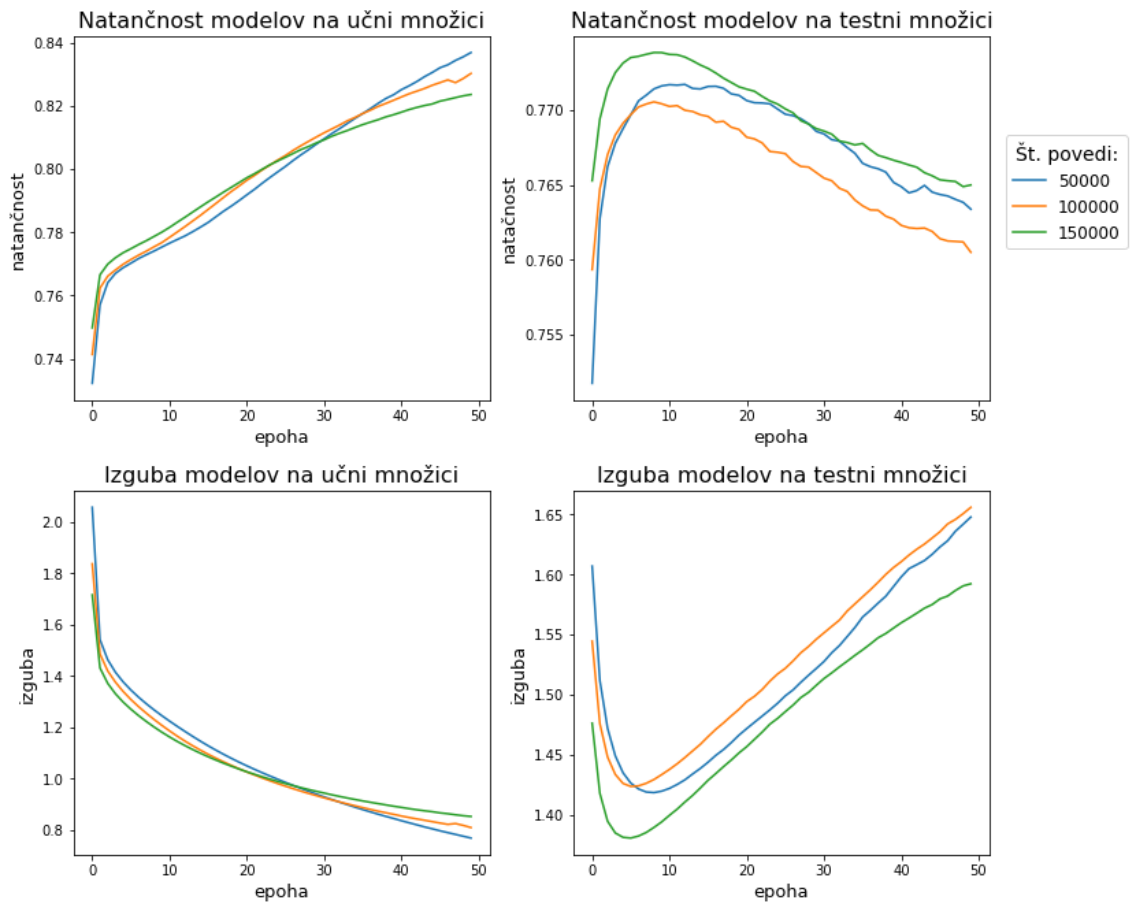
Slika 6.8 prikazuje primere pogovorov z angleškim modelom iz grafa na sliki 6.7. Rezultati kažejo na boljše naučeni model. Čeprav so odgovori še vedno dokaj nesmiselni, so besede zelo smiselno povezane v posplošene povedi, ki niso le memorizirane povedi iz podatkovne zbirke.



Slika 6.8: Primeri angleških pogovorov z velikostjo paketa 32.

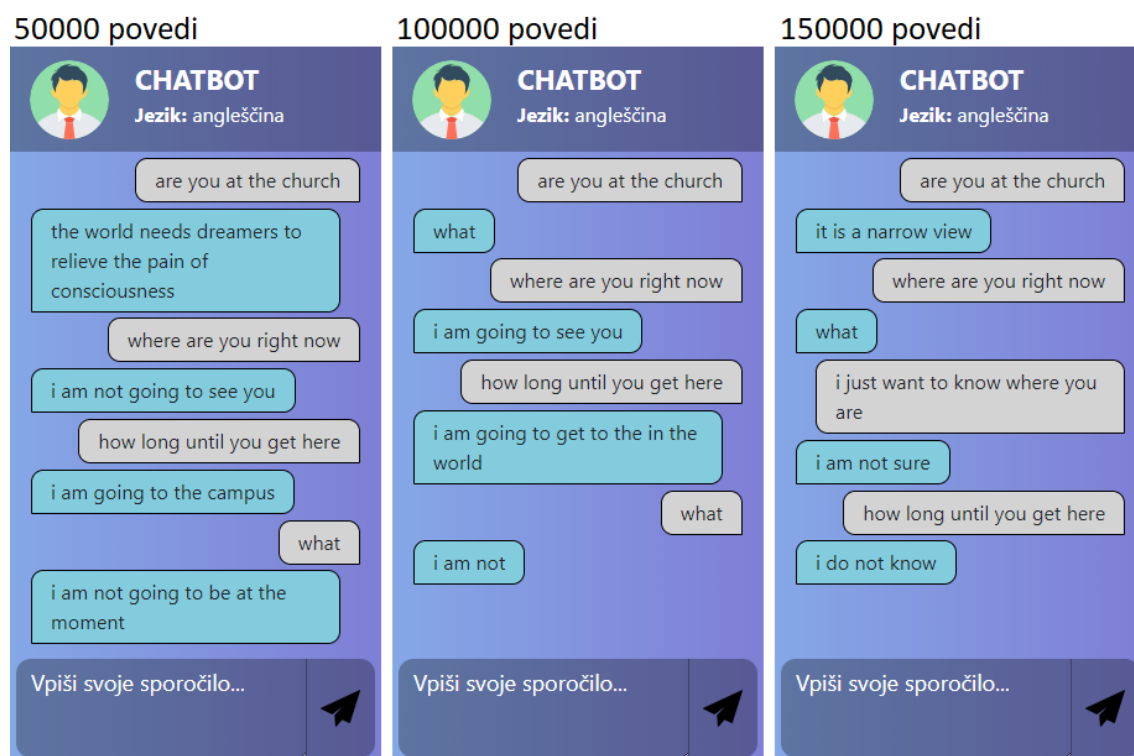
Drugi način, kako lahko modelu preprečimo, da se predobro nauči učnih podatkov je, da uporabimo večjo podatkovno zbirko. Naša angleška podatkovna zbirka vsebuje preko dvesto sedemdeset tisoč povedi, vendar smo za primerjavo s slovenskimi podatki uporabljali le prvih petdeset tisoč povedi. Za naslednji test smo število vhodnih podatkov povečali. S tem se je občutno podaljšalo tudi trajanje učenja. Za model s petdeset tisoč povedmi je učenje trajalo 43 minut, za sto petdeset tisoč povedi pa kar 5 ur in 16 minut, zato smo se pri tej količini povedi tudi ustavili.

Grafi na sliki 6.9 prikazujejo, kako na meritve vpliva število podatkov, uporabljenih za učenje modela. Opazimo, da so si vsi trije modeli glede natančnosti zelo blizu saj eden od drugega odstopajo za manj kot dva odstotka. Glede izgube najboljše rezultate prikaže model z največjim številom povedi, saj so pri njem izgube na testni množici najnižje. Model, naučen na sto tisoč povedih, glede na izgube na testni množici kaže najslabše rezultate, kar pa je lahko tudi posledica nesrečno izbranih povedi za testno množico, saj glede na izgube na učni množici kaže druge najboljše rezultate.



Slika 6.9: Primerjava angleških modelov glede na velikost podatkovne zbirke.

Slika 6.10 prikazuje primere pogovorov z modeli, ki smo jih učili na različnih količinah podatkov. V vseh primerih so uporabljeni zelo podobni dialogi. Vidimo, kako modela, naučena na več podatkih, uporabljata manj točno naučenih povedi in več posplošenih povedi v obliki povratnih vprašanj in zanikanj.



Slika 6.10: Primeri pogovorov modelov z različnim številom vhodnih podatkov.

Pokazali smo, kako na učenje naših modelov in kvaliteto pogovorov vplivata velikost paketa in velikost podatkovne zbirke. Pri učenju prihaja do prekomernega prilaganja, ki ga nismo uspeli odpraviti, vendar nam je uspelo, da je naš pogovorni robot sposoben pravilno tvoriti povedi, čeprav te povedi velikokrat niso najbolj smiselne glede na naše vprašanje oziroma vhod. To pomanjkljivost lahko pripišemo precej osnovni arhitekturi našega modela in sestavi nevronske mreže LSTM, pri katerih se izgublja referenca na kontekst iz preteklosti. Novejše, bolj dovršene arhitekture, so z uporabo dvosmernega kodirnika LSTM (angl. bi-directional LSTM encoder) [34] in mehanizma pozornosti (angl. attention mechanism) [35] sposobne dosegati veliko boljše rezultate, a so težje za implementirati in so računsko zahtevnejše.

7 ZAKLJUČEK

V diplomskem delu smo najprej spoznali, kaj so pogovorni roboti ter kako delujejo rekurentne nevronske mreže. Nato smo z uporabo rekurentnih nevronskih mrež LSTM in s sodobnimi tehnologijami strojnega učenja izdelali dva modela tako imenovanega pogovornega robota, s katerim lahko komuniciramo v izdelani mobilni aplikaciji.

Izdelava aplikacije je bila zelo preprosta, saj so tehnologije, ki jih uporablja ogrodje Angular zelo razširjene. Prav tako smo aplikacijo zasnovali tako, da uporablja minimalno število komponent in imitira izgled ostalih pogovornih platform.

Prvi model pogovornega robota smo naučili v slovenskem jeziku, kjer smo vse podatke za učenje pridobili iz podnapisov štiriintridesetih različnih filmov. Drugi model smo naučili v angleškem jeziku, kjer smo podatke za učenje pridobili iz korpusa urejenih dialogov iz kar 617 različnih filmov in uporabili bolj uveljavljene, vnaprej naučene besedne vložitve (GloVe). Tekom testiranja delovanja obeh modelov smo naučili tudi več testnih modelov z uporabo različnih velikosti paketov in z različnim številom vhodnih podatkov. Razlike med modeli smo prikazali z grafi in primeri pogovorov. Arhitektura naših modelov je zelo osnovna, zato je kvaliteta pogovorov tudi temu primerna in odgovori pogovornega robota niso najbolj smiselni glede na naša vprašanja. Z uporabo novejših arhitektur in večjo podatkovno zbirko bi rezultate znatno izboljšali, vendar nas nekoliko tudi omejuje zmogljivost osebnega računalnika, ker je takšno učenje tako časovno kot računsko zelo zahtevno. Kljub temu smo dosegli zadani cilj diplomskega dela, saj naša aplikacija deluje in lahko z uporabo modelov, ki vsebujejo rekurentne nevronske mreže LSTM, karseda smiselno komuniciramo.

8 VIRI IN LITERATURA

- [1] Wikipedia, „Chatbot,“ [Elektronski]. Dostopno na: <https://en.wikipedia.org/wiki/Chatbot>. [Poskus dostopa 22. 7. 2020].
- [2] S. Patel, „What is Chatbot? Why are Chatbots Important?,“ [Elektronski]. Dostopno na: <https://www.revechat.com/blog/what-is-a-chatbot>. [Poskus dostopa 22. 7. 2020].
- [3] B. A. Shawar in E. Atwell, „Chatbots: Are they Really Useful?,“ 2007. [Elektronski]. Dostopno na: https://jcl.org/content/2-allissues/20-Heft1-2007/Bayan_Abu-Shawar_and_Eric_Atwell.pdf. [Poskus dostopa 22. 7. 2020].
- [4] Wikipedia, „Turing test,“ [Elektronski]. Dostopno na: https://en.wikipedia.org/wiki/Turing_test. [Poskus dostopa 22. 7. 2020].
- [5] Hubtype, „Rule-Based Chatbots vs. AI Chatbots: Key Differences,“ [Elektronski]. Dostopno na: <https://www.hubtype.com/blog/rule-based-chatbots-vs-ai-chatbots/>. [Poskus dostopa 22. 7. 2020].
- [6] D. Pickell, „What Is a Chatbot? The Full Guide to Chatbots in 2020,“ [Elektronski]. Dostopno na: <https://learn.g2.com/chatbot>. [Poskus dostopa 22. 7. 2020].
- [7] M. Wallace in G. Dunlop, „ELIZA: a very basic Rogerian psychotherapist chatbot,“ [Elektronski]. Dostopno na: <https://web.njit.edu/~ronkowitz/eliza.html>. [Poskus dostopa 22. 7. 2020].
- [8] Facebook, „Facebook for Developers,“ [Elektronski]. Dostopno na: <https://developers.facebook.com/docs/messenger-platform/getting-started/sample-apps/>. [Poskus dostopa 22. 7. 2020].
- [9] Wikipedia, „Virtual assistant,“ [Elektronski]. Dostopno na: https://en.wikipedia.org/wiki/Virtual_assistant. [Poskus dostopa 22. 7. 2020].
- [10] Google, „Google assistant, your own personal Google,“ [Elektronski]. Dostopno na: <https://assistant.google.com/>. [Poskus dostopa 28. 8. 2020].
- [11] Apple, „Siri,“ [Elektronski]. Dostopno na: <https://www.apple.com/siri/>. [Poskus dostopa 28. 8. 2020].
- [12] Amazon, „Amazon Alexa Official Site: What is Alexa?,“ [Elektronski]. Dostopno na: <https://developer.amazon.com/en-GB/alexa>. [Poskus dostopa 28. 8. 2020].

- [13] Microsoft, „Cortana - Your personal productivity assistant,“ [Elektronski]. Dostopno na: <https://www.microsoft.com/en-us/cortana>. [Poskus dostopa 28. 8. 2020].
- [14] UneeQ, „A Siri-ous Guide To Voice Assistants: Benefits of Virtual Assistant AI in 2020,“ [Elektronski]. Dostopno na: <https://digitalhumans.com/blog/what-are-virtual-assistants/>. [Poskus dostopa 22. 7. 2020].
- [15] Microsoft, „Explore Windows 10 OS, Computers, Apps & More,“ [Elektronski]. Dostopno na: <https://www.microsoft.com/en-us/windows/>. [Poskus dostopa 28. 8. 2020].
- [16] N. Donges, „Recurrent neural networks 101: Understanding the basics of RNNs and LSTM,“ [Elektronski]. Dostopno na: <https://builtin.com/data-science/recurrent-neural-networks-and-lstm>. [Poskus dostopa 22. 7. 2020].
- [17] C. Olah, „Understanding LSTM Networks,“ [Elektronski]. Dostopno na: <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>. [Poskus dostopa 2. 8. 2020].
- [18] A. Karpathy, „The Unreasonable Effectiveness of Recurrent Neural Networks,“ [Elektronski]. Dostopno na: <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>. [Poskus dostopa 2. 8. 2020].
- [19] N. Guid in D. Strnad, Umetna inteligenca, 1. izdaja, Maribor: Fakulteta za elektrotehniko, računalništvo in informatiko, 2015.
- [20] M. Phi, „Illustrated Guide to LSTM's and GRU's: A step by step explanation,“ [Elektronski]. Dostopno na: <https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21>. [Poskus dostopa 3. 8. 2020].
- [21] A. Ali in M. Z. Amin, „Conversational AI Chatbot Based on Encoder-Decoder,“ [Elektronski]. Dostopno na: https://www.researchgate.net/publication/338100972_Conversational_AI_Chatbot_Based_on_Encoder-Decoder_Architectures_with_Attention_Mechanism. [Poskus dostopa 3. 8. 2020].
- [22] Python, „Welcome to Python.org,“ [Elektronski]. Dostopno na: <https://www.python.org/>. [Poskus dostopa 28. 8. 2020].
- [23] TensorFlow, „TensorFlow,“ [Elektronski]. Dostopno na: <https://www.tensorflow.org/>. [Poskus dostopa 28. 8. 2020].

- [24] Keras, „Keras: the Python deep learning API,“ [Elektronski]. Dostopno na: <https://keras.io/>. [Poskus dostopa 28. 8. 2020].
- [25] Cornell CIS, „Cornell Movie--Dialogs Corpus,“ [Elektronski]. Dostopno na: https://www.cs.cornell.edu/~cristian/Cornell_Movie-Dialogs_Corpus.html. [Poskus dostopa 23. 7. 2020].
- [26] J. Brownlee, „What Are Word Embeddings for Text?,“ [Elektronski]. Dostopno na: <https://machinelearningmastery.com/what-are-word-embeddings/>. [Poskus dostopa 3. 8. 2020].
- [27] Gensim, „gensim: models.word2vec - Word2vec embeddings,“ [Elektronski]. Dostopno na: <https://radimrehurek.com/gensim/models/word2vec.html>. [Poskus dostopa 28. 8. 2020].
- [28] J. Pennington, R. Socher in C. Manning, „GloVe: Global Vectors for Word Representation,“ [Elektronski]. Dostopno na: <https://nlp.stanford.edu/projects/glove/>. [Poskus dostopa 3. 8. 2020].
- [29] T. Mikolov, K. Chen, G. Corrado in J. Dean, „Efficient Estimation of Word Representations in Vector Space,“ [Elektronski]. Dostopno na: <https://arxiv.org/abs/1301.3781>. [Poskus dostopa 28. 8. 2020].
- [30] Angular, „Angular,“ [Elektronski]. Dostopno na: <https://angular.io/>. [Poskus dostopa 28. 8. 2020].
- [31] M. L. Bors, „An overview of Angular,“ [Elektronski]. Dostopno na: <https://medium.com/@mlbors/an-overview-of-angular-3ccd2950648e>. [Poskus dostopa 3. 8. 2020].
- [32] Apache, „Apache Cordova,“ [Elektronski]. Dostopno na: <https://cordova.apache.org/>. [Poskus dostopa 28. 8. 2020].
- [33] D. P. Kingma in J. Ba, „Adam: A method for stochastic optimization,“ [Elektronski]. Dostopno na: <https://arxiv.org/abs/1412.6980>. [Poskus dostopa 28. 8. 2020].
- [34] A. Padmanabhan, „Devopedia - Bidirectional RNN,“ [Elektronski]. Dostopno na: <https://devopedia.org/bidirectional-rnn>. [Poskus dostopa 28. 8. 2020].
- [35] A. Padmanabhan, „Devopedia - Attention Mechanism in Neural Networks,“ [Elektronski]. Dostopno na: <https://devopedia.org/attention-mechanism-in-neural-networks>. [Poskus dostopa 28. 8. 2020].