Analiza sentimenta

Buday, Lovro

Sveučilište J.J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni preddiplomski studij matematike

Lovro Buday

Analiza sentimenta

Završni rad

Osijek, 2020.

Sveučilište J.J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni preddiplomski studij matematike

Lovro Buday

Analiza sentimenta

Završni rad

Mentor: doc. dr. sc. Domagoj Ševerdija

Osijek, 2020.

Sažetak

U ovome radu bavit ćemo se problemom analize sentimenta. Problem analize sentimenta je problem izvlačenja emocionalnog tona iz određenog teksta. Problem ćemo rješavati strojnim učenjem, proći ćemo modele strojnog učenja koje koristimo i zašto. Rad detaljno prolazi način na koji reprezentiramo tekst modelu strojnog učenja. Učinkovitost metoda će biti potkrijepljena formulama ili citiranjem prijašnjih radova o zadanim temama.

Ključne riječi

analiza sentimenta, neuronske mreže, n-grami, vektorizacija riječi, word2Vec, RNN, LSTM

Sentiment analysis

Summary

In this paper we are going to go over the problem of sentiment analysis. That is the problem of extracting the emotional tone from a given text. We are going to solve it using machine learning, the paper goes over the models of machine learning we are going to use and it goes over why we use them. It includes a detailed look at how we represent the text to our machine learning model. The effectiveness of the methods is going to be backed up by formulas or by quoting previous papers done on the subject.

Key words

sentiment analysis, neural network, n-gram, word embedding, word2Vec, RNN, LSTM

Sadržaj

Uvod

Problem analize sentimenta je problem izvlačenja emocionalnog tona iz određenog teksta. Pod time mislimo na pozitivan ili negativan ton, a u nekim primjenama i neutralan.

Najčešće se koristi za nadziranje društvenih mreža, za izvlačenje boljeg pregleda javnog mišljenja o određenim temama. Iz toga proizlazi drugo ime Opinion Mining. Uporaba seže sve od izvlačenja mišljenja o markama sportske odjeće do izvlačenja mišljenja o politici

Analiza se radi preko modela strojnog učenja. Velika količina podataka sa unaprijed označenim emocionalnim tonom se obrađuje alatima strojnog učenja i dobiva se model koji može predvidjeti ton teksta sa određenom točnošću. Najvažniji faktori kod strojnog učenja su preciznost i brzina učenja. Brzinu učenja i preciznost modela poboljšavamo na više načina, rješavajući probleme koji su česti u primjenama Natural Language Processinga. Tim problemima ćemo se baviti u ovom radu, oni su navedeni u slijedećem stavku.

U svrhu učinkovite i precizne analize sentimenta morat ćemo riješiti probleme koji nas najviše sputavaju u tom zadatku.

1. **Stvaranje vokabulara i reprezentacija teksta.** Kako bi znatno smanjili količinu računanja kroz koju neuronska mreža mora proći riječi se ne predaju modelu slovo po slovo nego kao reprezentacija riječi u obliku vektora ili cijelih brojeva.

2. **Modeli strojnog učenja u analizi sentimenta.** Proći ćemo koji su to najčešći modeli kod obrade teksta i zašto.

3. **Dodatna obrada teksta koja pomaže u analizi.** Spomenuti ćemo način na koji možemo izmijeniti tekst prije nego ga predamo stroju. I kako takav tekst pomaže kod treniranja modela. Bavit ćemo se temom n-grama.

1 Stvaranje vokabulara i reprezentacija teksta

Kao što smo spomenuli modelu strojnog učenja predajemo reprezentaciju riječi a ne individualne riječi slovo po slovo. Zato stvaramo vokabular, ili drugim riječima stvaramo tablicu u kojoj svaka jedinstvena riječ ima svoj indeks.

1.1 Stvaranje vokabulara

Vokabular je skup jedinstvenih riječi iz našeg skupa podataka. Svaka riječ ima svoj indeks a svaki indeks reprezentira jedan one-hot vektor tj. vektor koji iščezava na svim poljima osim na polju koje odgovara indeksu riječi na kojem je vrijednost jednaka 1. Mana ovog sustava je memorijska zahtjevnost. Naš one-hot vektor biti će veličine našeg vokabulara npr. ako vokabular ima 100,000 riječi imati će 100,000 polja. Način na koji smanjujemo ovaj problem je tako da riječi koje se rijetko koriste stavimo pod jednaki indeks, te tako možemo ograničiti veličinu vokabulara a s time i veličinu one-hot vektora. Prvo izračunamo frekvenciju svih riječi u skupu podataka i poredamo ih po redu.

Ako definiramo funkciju:

$freq(x) =$ redni broj pozicije *riječi* x u poredanim frekvencijama

onda određujemo indeks od *riječi* x sa

$$f(x) = \begin{cases} freq(x), & \text{ako } \frac{freq(x)}{V_{size}-1} \leq 1 \\ V_{size}, & \text{inače} \end{cases}$$

gdje je V_{size} željena veličina vokabulara. Znači sve riječi koje su po redu frekvencija veće od veličine vokabulara manje jedan biti će reprezentirani jednim indeksom. Kada indeks ne pripada točno jednoj riječi ili ne pripada riječi uopće nazivamo ga token.

1.2 Tokenizacija

Kao što smo spomenuli skupina riječi čija $freq()$ vrijednost je veća od veličine vokabulara označena je tokenom <unknown> ili <UNK> tj. nepoznato. Taka tokenizacija pomaže kod Out of Vocabulary(OOV) riječi. Dakle u slučaju da postoje riječi u primjeni koje nisu bile prisutne kod stvaranja vokabulara biti će klasificirane kao <unknown> i model neće zastati. Mana je što gubimo ikakav kontekst te riječi. Iako postoji mnogo načina na koje se riječi mogu tokenizirati i razne svrhe u koje se takve tokenizacije koriste, mi ćemo promatrati samo onaj način koji nam pomaže kod analize sentimenta. To je u većini slučajeva token riječi je reprezentacija riječi, u nekim slučajevima kao što smo vidjeli token je <unknown>. Dodatni korisni tokeni su: token za početak skupina rečenica(nazovimo ga <CLS>), token za razdvajanje rečenica(nazovimo ga <SEP>) i padding token(tj. token za popunjavanje praznina [nazovimo ga <PAD>]).

<PAD> token koristimo za popunjavanje teksta do veličine koju naš model strojnog učenja uzima.

<CLS> i <SEP> dobri su za kontekst, može doći do zabune ako ne znamo gdje jedna rečenica počinje a druga završava. Također je neophodno kod modela za višeklasnu analizu sentimenta. U njoj je potrebno svaku rečenicu uz analizu i klasificirati. Za to je neophodno znati odvojiti rečenice točno.

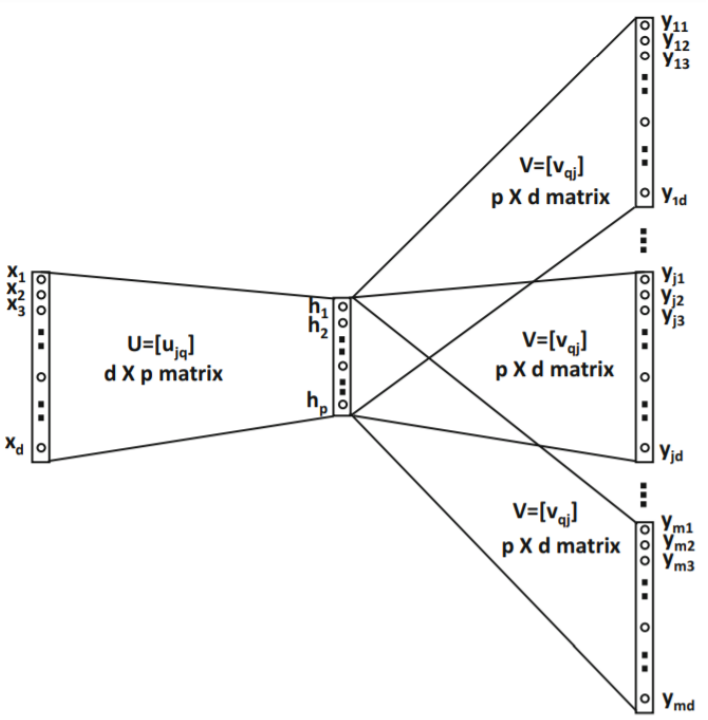
Tokenizacija nam pomaže kod analize i u nekim slučajevima je neophodna(<PAD> token osobito). Ali da bi bolje na trenirali model, bilo bi poželjno da tokeni riječi imaju odnos s obzirom na semantičko značenje. Pretvaranje riječi u nisko dimenzionalne vektore s obzirom na semantičko značenje nazivamo Vektorizacija riječi(eng. Word Embedding).

1.3 Vektorizacija riječi

Vektore za vektorizaciju riječi dobijemo obradom velikog korpusa određenim metodama. Dvije najpoznatije metode su CBOW(Continuous bag of words) i skip-gram. Obje metode se baziraju na odnosu centralne riječi i n riječi koje je okružuju te riječi nazivamo kontekst centralne riječi(kao što je pokazano u [2]). Model CBOW pokušava predvidjeti centralnu riječ ako unesemo kontekst, a skip-gram pokušava predvidjeti kontekst ako unesemo centralnu riječ. Taj postupak se ponavlja na više mjesta u korpusu. Na taj način svaka riječ je definirana svojom sposobnošću da predvidi kontekst u kojem se nalazi. Tako je traženje vektora za vektorizaciju savršen zadatak za nenadzirano učenje, iako je proces intenzivan rezultat(tj. dobivene vektore za vektorizaciju) možemo iskoristiti u više različitih primjena bez da ponovo obrađujemo podatke.

Za naš zadatak proći ćemo skip-gram metodu jer nam ona daje bolju reprezentaciju riječi koje se rjeđe pojavljuju u vokabularu, što je i poželjno.

Kao što je rečeno skip-gram uzima traženu riječ w kao ulaz i predviđa kontekst riječi $w_1...w_m$. Cilj nam je dobiti $P(w_1, w_2, ...w_m|w)$. Prvo riječi reprezentiramo kao one-hot vektore, tada će skip-gram model primat d(jednak veličini vokabulara) binarnih ulaza prikazanih sa $x_1...x_d$, a izlaz za svaku instancu je dimenzije $m \times d$. Svaki element izlaza skip-gram modela $y_{ij} \in \{0, 1\}$ govori jel i-ta riječ konteksta poprima j-tu moguću vrijednost. Vjerojatnosti u izlaznoj sloju za fiksan i i varirajući j imaju sumu jednaku 1. Skriveni sloj u našem modelu ima p članova(p je jednak dimenziji vektorizacije), izlazi su označeni sa $h_1...h_p$. Svaki x_j je spojen sa skrivenim čvorovima pomoću matrice U dimenzije $d \times p$. Nadalje p skrivenih čvorova je također spojeno sa m grupa d izlaznih čvorova sa istim skupom zajedničkih težina definiranih sa maticom V dimenzije $p \times d$.



Slika 1: Skip-gram model

Izlaz skrivenog sloja možemo izračunat koristeći težine u matrici $U = [u_{jq}]$ na sljedeći način

$$h_q = \sum_{j=1}^{d} u_{jq} x_j \ \forall q \in \{1...p\}$$

Pošto su riječi one-hot enkodirane x_j će biti jednak nuli na svim mjestima osim jednog. Tako je interpretacija ove jednadžbe u slučaju da je w r-ta riječ onda jednostavno kopiramo u_{rq} za svaki $q \in \{1...p\}$. Tj. r-ti red $\bar{u}_r$ od U je kopiran u skriveni sloj. Kao što je ranije navedeno skriveno sloj je povezan sa m grupa od d izlaza povezanih matricom $V = [v_{qj}]$. Pošto istu matricu V sve grupe dijele, neuronska mreža predviđa istu multinomijalnu distribuciju za svaku riječ konteksta. Pa imamo:

$$\widehat{y}_{ij} = P(y_{ij} = 1|w) = \frac{exp(\sum_{q=1}^{p} h_q v_{qj})}{\sum_{k=1}^{d} exp(\sum_{q=1}^{p} h_q v_{qk})} \ \forall i \in \{1...m\}$$

Uočavamo da $\widehat{y}_{ij}$ je jednak za varirajući i a fiksan j. Funkcija greške za povratnu propagaciju je:

$$L = -\sum_{i=1}^{m} \sum_{j=1}^{d} y_{ij} log(\widehat{y}_{ij})$$

Jednadžba za ažuriranje vrijednosti sa α stopom učenja je:

$$\bar{u}_i \Leftarrow \bar{u}_i - \alpha \frac{\partial L}{\partial \bar{u}_i} \ \forall i$$
$$\bar{v}_j \Leftarrow \bar{v}_j - \alpha \frac{\partial L}{\partial \bar{v}_j} \ \forall j$$

gdje je $\bar{v}_j$ j-ti stupac matrice V. Za našu vektorizaciju koristimo kao vektore p-dimenzionalne retke matrice U.

1.3.1 Brzina računanja

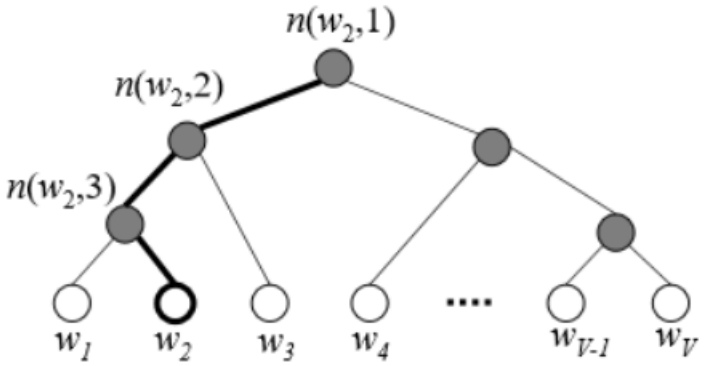
Ako promotrimo funkciju za softmax,

$\frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{V_{size}} \theta_j^T e_c}$, gdje je θ_t parametar asociran sa traženim izlazom(tj. outputom t, našoj traženoj kontekst riječi)

Ako obratimo pozornost na nazivnik ovog razlomka primjećujemo da je on jednak sumi koja ide od jedan sve do veličine vokabulara. Ako uzmemo u obzir da će naš vokabular biti jako velik(u većini slučajeva $>$ 50000 riječi) računati ovakvu sumu je previše intenzivan zadatak za svaku iteraciju metode. Da bi model bili u mogućnosti istrenirati moramo drastično smanjiti računsku cijenu zadatka. Metoda kojom ćemo to napraviti zove se hijerarhijski softmax.

Ocjena brzine za softmax je $O(n)$, a za hijerarhijski softmax je $O(log(n))$. Kao što smo vidjeli u skip-gramu vjerojatnost jednog izlaza ovisi o zbroju svih izlaza.

Hijerarhijski softmax koristi binarno stablo da reprezentira sve riječi u vokabularu. Svaki list stabla je čvor našeg grafa, i postoji jedinstveni put od korijena do lista. Svaki čvor između eksplicitno reprezentira relativne vjerojatnosti svoje djece(tj. čvorova ispod njega). Tako su ti čvorovi asocirani sa različitim vektorima koje će naš model naučiti. [citirano sa [3]]

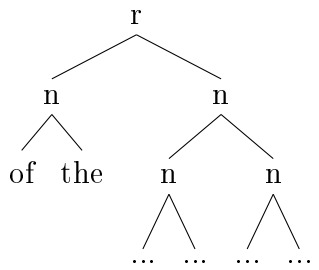


Slika 2: hierarhycal-softmax

Označimo vjerojatnosti kretanja lijevo na čvoru n kao $p(n, lijevo)$ i vjerojatnost da idemo desno kao $p(n, desno)$, tada definiramo vjerojatnost da ćemo doći do riječi w_2 na sljedeći način:

$$P(w_2|w_i) = p(n(w_2, 1), lijevo) * p(n(w2, 2), lijevo) * p(n(w_2, 3), desno)$$

Tj. ako slijedimo put kao na priloženoj slici i pratimo put od korijena do lista w_2. Vjerojatnosti su umnožak vjerojatnosti da skrenemo u određenom smjeru na priloženom čvoru. Navedeni proces implicira da će cijena računanja procesa biti proporcionalna broju čvorova na putu između korijena i željenog lista.
Da bi još dodatno poboljšali tj. smanjili cijenu računanja, stablo nećemo u potpunosti balansirati nego ćemo listove frekventnih riječi staviti bliže vrha tj. na putu sa manje međučvorova, a jako rijetke riječi ćemo staviti što dublje u stablu.



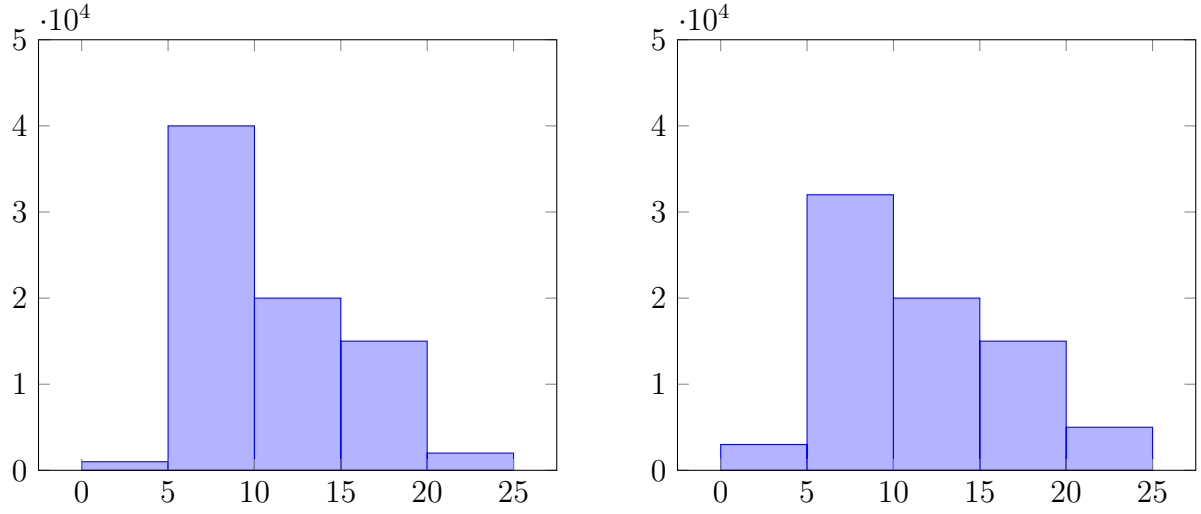
Slika 3: primjer ne balansiranog stabla

Ako je r korijen, a n neka su čvorovi, htjeli bi da su česte riječi kao "of" i "the" pri vrhu kao na slici.

1.3.2 Biranje riječi

Rekli smo da riječi koje predviđamo pomoću vektorizirane riječi biramo kao jednu od n riječi prije ili poslije centralne riječi. Ali da poboljšamo naše vektore za vektoriziranje riječi, riječi moramo birati na određeni način. Pogledajmo histograme frekvencija određenih riječi.

Ako usporedimo ne normalizirane i normalizirane podatke primjećujemo da su podatci niske frekvencije povećali frekvenciju, a podatci visoke frekvencije su smanjili frekvenciju. Bolje vektore za vektorizaciju dobivamo ako oni mogu bolje predvidjeti pojavljivanje nisko frekventnih riječi ih vokabulara. U tu svrhu ćemo birati takve riječi za treniranje modela malo češće nego se pojavljuju, a one više frekventne ćemo birati malo rjeđe.



(a) Prije normalizacije

(b) Nakon normalizacije

Slika 4: Prikaz prije i poslije normalizacije

1.3.3 Ocjena kvalitete

Da bi izračunali kvalitetu naše vektorizirane riječi moramo imati način da testiramo odnos između dva vektora. Npr. ako želimo dobiti vektor za "manji" možemo ga poistovjetiti sa vektorom za "mali". Sada ako pogledamo odnos između "velik" i "veći" on bi trebao biti isti kao odnos za vektore "mali" i "manji". Tako da dobijemo "manji" imamo formulu X = vector(”veći”)− vector(”velik”) + vector(”mali”) gdje bi X trebao biti jednak vector("manji"). Problem je što vektor nećemo sa točnom vrijednošću nego ćemo dobiti vektor koji se najvjerojatnije ne mapira na ni jedan vektor u vokabularu. Tada gledamo vektor koji mu je najbliži koristeći kosinus sličnost. Tj. sa formulom:

$$\frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}} \rightarrow min$$

Te uzmemo tablicu unaprijed određenih odnosa i testiramo koliko je često dobivenom vektoru najbliži traženi vektor(najbliži po priloženoj formuli). Odnosi kao u ovoj tablici.

Zaključak o vektorizaciji riječi

Izračunati vektore koji reprezentiraju riječi je jako intenzivan zadatak ali se mora odraditi samo jednom i znatno pomaže u procesiranju prirodnog jezika(u NLP primjenama). Te uvijek možemo preskočiti cijeli proces i preuzeti već izrađene vektore za vektorizaciju sa open source izvora.

Type of relationship	Word Pair 1		Word Pair 2	
Common capital city	Athens	Greece	Oslo	Norway
All capital cities	Astana	Kazakhstan	Harare	Zimbabwe
Currency	Angola	kwanza	Iran	rial
City-in-state	Chicago	Illinois	Stockton	California
Man-Woman	brother	sister	grandson	granddaughter
Adjective to adverb	apparent	apparently	rapid	rapidly
Opposite	possibly	impossibly	ethical	unethical
Comparative	great	greater	tough	tougher
Superlative	easy	easiest	lucky	luckiest
Present Participle	think	thinking	read	reading
Nationality adjective	Switzerland	Swiss	Cambodia	Cambodian
Past tense	walking	walked	swimming	swam
Plural nouns	mouse	mice	dollar	dollars
Plural verbs	work	works	speak	speaks

Slika 5: tablica parova vektorizacije

2 Modeli strojnog učenja u analizi sentimenta

Analiza sentimenta iz teksta ne bavi se samo traženjem riječi koje vuku negativnu ili pozitivnu konotaciju, nego i u vaganju te konotacije. Ako moramo analizirat tekst u kojem neko opisuje iskustvo i dio tog iskustva opiše riječima "jako grozno" i jedan dio opiše sa "zadovoljavajuće" za taj bi tekst rekli da ima negativan emocionalan ton iako je imao pozitivnu komponentu. Sastaviti običan program sa if-else grananjima za računanje sentimenta bilo bi skoro nemoguće ako ne i nemoguće. Zato moramo na trenirati neuronsku mrežu koja će sama naučiti težine određenih riječi i fraza i moći to primijeniti sa određenom sigurnošću na do tada neviđeni tekst. Jedan od modela koji se pokazao obećavajućim za klasifikacije teksta naziva se RNN(Recurrent neural network) model. Ako proučimo njegove karakteristike shvatit ćemo zašto.

2.1 RNN

Povratna neuronska mreža(recurrent neural network) je tip neuronske mreže koja je specijalizirana za obradu slijedova. Jedan od problema sa klasičnim neuronskim mrežama je da primaju ulaze fiksne veličine i daju izlaze fiksne veličine. Velika prednost RNN mreža je što mogu imati ulazi i izlaze varirane veličine(kao što je argumentirano u [6]).

One rade tako da iterativnim procesom ažuriraju skriveno stanje(eng. hidden state) označimo ga sa h_t(skriveno stanje koraka t). Naša RNN mreža prima ulaz i prijašnje h_{t-1}, ako je prva iteracija po redu onda prima inicijalno skriveno stanje h_0 koje je u većini slučajeva jednako 0. Dobivanje sljedećeg skrivenog stanja definirali bi rekurzivnom funkcijom:

$h_t = f_W(h_{t-1}, x_t)$, gdje je x_t ulaz na koraku t, a f_W funkcija ovisna o matrici težina unutar

mreže. Ta f_W funkcija najčešće prima oblik:

$$f_W(h_{t-1}, x_t) = h_t = tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

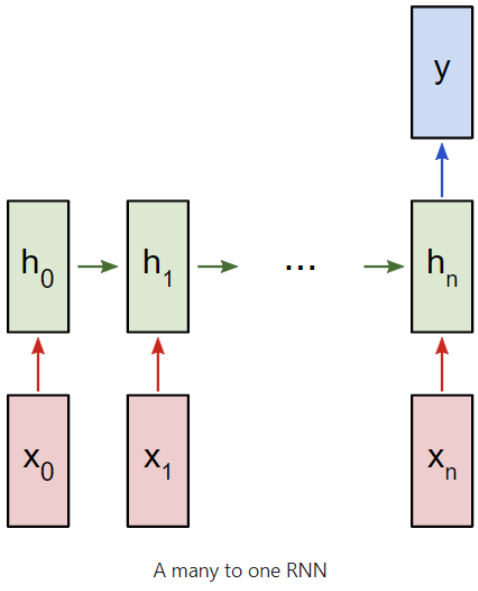
Izlaz tada ovisi o trenutno izračunatom skrivenom stanju i vlastitoj težinskoj matrici tj. $y_t = W_{hy}h_t$.

Dodatno u te jednadžbe možemo dodati bias(pristranost) koji na željeni način usmjerava podatke.

$$f_W(h_{t-1}, x_t) = h_t = tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$
$$y_t = W_{hy}h_t + b_y$$

U ovom primjeru koristimo tanh kao aktivacijsku funkciju ali možemo koristiti i sigmoid.

Za naš zadatak potreban nam je "many-to-one" tip RNN-a. To je tip koji prima više ulaza i koristi samo zadnje skriveno stanje za proizvest izlaz. Slikovni prikaz:



Slika 6: više naprema jedan RNN mreža

Težine u našoj težišnoj matrici se ažuriraju povratnom propagacijom. Algoritam za povratnu propagaciju radi tako da računa gradijent funkcije greške(koju ćemo poslije navest) s obzirom na svaku težinu uz pomoć lančanog pravila(formula za računanje derivacija kompozitne funkcije), iterirajući od posljednjeg sloja mreže.

Neka je f^L aktivacijska funkcija sloja L(u našem slučaju $tanh(W_{hh}h_{L-1} + W_{xh}x_L)$), $W^L = (w^L_{jk})$ je težišna matrica između slojeva $L-1$ i L, x je trenutni ulaz, a y željeni izlaz.

Cijela mreža je kombinacija kompozitnih funkcija i matričnih umnožaka tj.
$g(x) := f^L(W^L f^{L-1}(W^{L-1}...f^1(W^1 x)...))$
Našu funkciju greške definiramo sa $E(y_i, g(x_i))$ gdje je (x_i, y_i) uređeni par jednog ulaza i izlaza. $\widehat{y}$ je dobivena distribucija vjerojatnosti tj. predviđanja koja je model dao za ulaz x. Totalni error je suma greške za svaki vremenski korak(za korak gdje se računa aktivacija tj. onaj u kojem dobijemo skriveno stanje). Pogledajmo gradijent naše E funkcije s obzirom na W. Za primjer uzmimo instancu sa 3 rekurzivne iteracije naše RNN mreže. E_3 je error(greška) na trećem sloju pa imamo:

$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \widehat{y}} \frac{\partial \widehat{y}}{\partial f_3} \frac{\partial f_3}{\partial f_k} \frac{\partial f_k}{\partial f_W}$

Vidimo da je definiramo lančano. Jer pošto se error računa za svaki korak, za posljednji korak dobijemo:

$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \widehat{y}} \frac{\partial \widehat{y}}{\partial f_3} \frac{\partial f_3}{\partial f_W}$

A f_3 se dalje raspisuje kao $f_3 = tanh(W_{hh}h_2 + W_{xh}x_L)$ gdje je h_2 zapravo $f_2(h_1, x_1)$. Tako da sve te vrijednosti moramo rekurzivno uključit u derivaciju. Model ažurira podatke tako da:

$W \leftarrow W - \alpha \frac{\partial E_3}{\partial W}$, gdje je α veličina koraka.

2.1.1 Problem nestajućeg gradijenta(eng. Vanishing gradient problem)

Ako zamislimo da su naše W matrice samo skalari. Pošto lančano množimo njihove derivacije ako su one manje od 1 završni produkt kretat će se prema nuli. Kada se na taj način gradijent kreče prema nuli i dođe do broja toliko malog da skoro nema nikakvog utjecaja, taj problem nazivamo problem nestajućeg gradijenta(kao što je pokazano u [4]). Ista stvar može se dogoditi i elementima naše W matrice. Obrnuto se može dogoditi ako je vrijednost dovoljno velika da nadjača derivaciju tanh funkcije(koja je manja od 1), onda to nazivamo problem eksplozivnog gradijenta problem u kojem raste prema ∞. Ovakav problem nastaje i u dubokim neuronskim mrežama ali je puno gori u RNN mreži jer su matrice W jednake za svaki korak, a u dubokim mrežama su različite.

2.2 LSTM

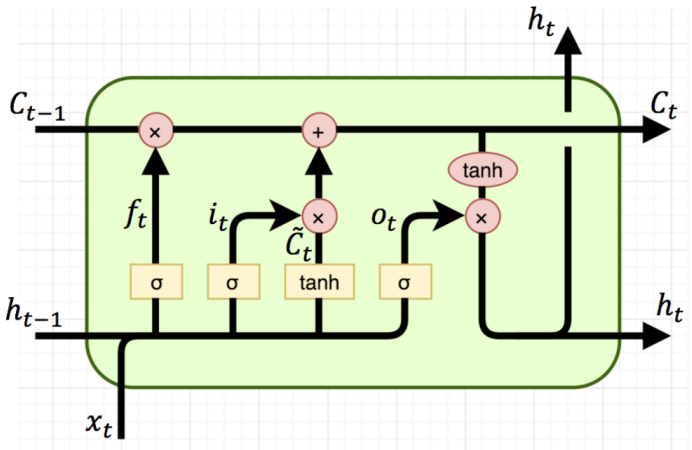
LSTM mreža je posebna vrsta RNN mreže koja uz skriveno stanje prima i šalje stanje ćelije(eng. cell state)(kao što je prikazano u [5]). To stanje kao i skriveno stanje ponovno šaljemo mreži. Unošenje i brisanje vrijednosti iz stanja ćelije radi se pomoću sklopki(eng.

gates).

- sklopka zaboravljanja(eng. forget gate)
- sklopka unosa(eng. input gate)
- sklopka izlaza(eng. output gate)

Sklopka zaboravljanja odlučuje koliko će podataka biti zaboravljeno u našem stanju ćelije, a sklopka unosa koliko će podataka biti uneseno u stanje ćelije. Sklopka izlaza odlučuje koliko će stanja ćelije biti otkriveno u računanju skrivenog stanja. Svo troje imaju sigmoid aktivaciju i daju broj između 0 i 1. U slučaju sklopke zaboravljanja 1 znači "u potpunosti zapamti", a 0 znači "u potpunosti zaboravi".

Pošto sklopka unosa kontrolira koliko će podataka biti uneseno potrebni su nam podatci koje planiramo unijeti. Kandidate za te podatke računamo pomoću tanh sloja. Označimo na koraku t, f_t je sklopka zaboravljanja, i_t sklopka unosa, $\tilde{C}_t$ je spomenuti tanh sloj. Dodatno neka su C_{t-1} i C_t stanja ćelija koraka $t-1$ i t, a o_t sklopka izlaza.



Slika 7: Slikovni prikaz LSTM mreže

Svaka sklopka ima svoju težišnu matricu, njihove težišne matrice su: W_f, W_i, W_C, W_o. tada i ih računamo formulama. ($*$ ćemo koristiti kao oznaku za množenje po elementima)

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$
$$\tilde{C}_t = tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

$h_t = o_t * tanh(C_t)$
gdje su b_f b_i b_C b_o, pristranost pripadajućih aktivacija. Radi pojednostavljenja formula ćemo ih nadalje zanemariti. Neka nam je funkcija greške ponovno E. Kao i kod RNN-a gradijent je dan sa:

$\frac{\partial E}{\partial W} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial W}$, gdje je E posljednji gradijent, E_t gradijent u koraku t, a T broj koraka.

Da bi se riješili problema nestajućeg gradijenta moramo spriječiti konvergiranje izraza $\sum_{t=1}^{T} \frac{\partial E_t}{\partial W}$ prema nuli. Ako zamislimo taj izraz kao red funkcija onda po definiciji on konvergira prema 0 ako mu niz parcijalnih suma konvergira prema nuli tj. $(S_1, S_2, S_3, S_4, ...)$ gdje $S_n = \sum_{t=1}^{n} \frac{\partial E_t}{\partial W}$ konvergira prema nuli.

Ako želimo da naš gradijent ne nestane, naša mreža treba povećati šansu da barem jedan od ovih parcijalnih gradijenta neće nestat [1]. Dakle moramo se pobrinuti da jedan od parcijalnih gradijenta neće konvergirat prema nuli. Raspišimo dalje naš gradijent u koraku t

$\frac{\partial E_t}{\partial W} = \frac{\partial E_t}{\partial h_t}\frac{\partial h_t}{\partial C_t}...\frac{\partial C_2}{\partial C_1}\frac{\partial C_1}{\partial W} = \frac{\partial E_t}{\partial h_t}\frac{\partial h_t}{\partial C_t}(\prod_{j=2}^{t}\frac{\partial C_j}{\partial C_{j-1}})\frac{\partial C_1}{\partial W}$

U običnoj RNN mreži produkt $\prod_{j=2}^{t}\frac{\partial C_j}{\partial C_{j-1}}$ bi uzrokovao nestajući gradijent. U LSTM ako raspišemo:

$\frac{\partial C_j}{\partial C_{j-1}} = \frac{\partial}{\partial C_{j-1}}[C_{j-1} * f_j + \tilde{C}_j * i_j] = \frac{\partial f_j}{\partial C_{j-1}} \cdot C_{j-1} + \frac{\partial C_{j-1}}{\partial C_{j-1}} \cdot f_j + \frac{\partial i_j}{\partial C_{j-1}} \cdot \tilde{C}_j + \frac{\partial \tilde{C}_j}{\partial C_{j-1}} \cdot i_j =$
$= \sigma'(W_f \cdot [h_{j-1}, x_j]) \cdot W_f \cdot o_{j-1} * tanh'(C_{j-1}) \cdot C_{j-1}$
$+ f_j$
$+ \sigma'(W_i \cdot [h_{j-1}, x_j]) \cdot W_i \cdot o_{j-1} * tanh'(C_{j-1}) \cdot \tilde{C}_j$
$+ \sigma'(W_C \cdot [h_{j-1}, x_j]) \cdot W_C \cdot o_{j-1} * tanh'(C_{j-1}) \cdot i_j$

Označimo posebno elemente koje zbrajamo:

$A_j = \sigma'(W_f \cdot [h_{j-1}, x_j]) \cdot W_f \cdot o_{j-1} * tanh'(C_{j-1}) \cdot C_{j-1}$
$B_j = f_j$
$C_j = \sigma'(W_i \cdot [h_{j-1}, x_j]) \cdot W_i \cdot o_{j-1} * tanh'(C_{j-1}) \cdot \tilde{C}_j$
$D_j = \sigma'(W_C \cdot [h_{j-1}, x_j]) \cdot W_C \cdot o_{j-1} * tanh'(C_{j-1}) \cdot i_j$

Ako uvrstimo u gradijent:

$\frac{\partial E_t}{\partial W} = \frac{\partial E_t}{\partial h_t}\frac{\partial h_t}{\partial C_t}(\prod_{j=2}^{t}[A_j + B_j + C_j + D_j])\frac{\partial C_1}{\partial W}$

Uočavamo da je naš produkt unutar gradijenta napravljen od sume 4 elementa koji imaju različita ponašanja. Kao takvi mogu se puno bolje balansirati i tako spriječiti nestajući gradijent, dok u običnoj RNN mreži se sastoji od izraza sličnih ponašanja koji su svi između 0 i 1.

primjer: $\prod_{j=2}^{t}[A_j + B_j + C_j + D_j] \approx \prod_{j=2}^{t}[0.2 + 0.6 + 0.05 + 0.15] \approx \prod_{j=2}^{t} 1 \not\to 0$

Ne konvergira prema nuli. Nestajući gradijent se još uvijek može pojavit ali je to jako rijetko u LSTM mrežama.

2.3 Cross Entropy loss

Funkcija greške koju koristimo je tzv. Cross Entropy loss ili log-loss. Njena formula je:

$E(\widehat{y}, y) = -\frac{1}{n}\sum_{i=1}^{n} y_i \times log\ \widehat{y}_i$, y je tražena distribucija vjerojatnosti, a $\widehat{y}$ dobivena distribucija nakon evaluacije našom mrežom. Ako u traženoj distribuciji y tražimo da je samo prvi element jednak jedan, a u $\widehat{y}$ neki vektor $[x_1, x_2, ..., x_n]$ raspisom $E(\widehat{y}, y) = -\frac{1}{n}(1 \times log(x_1) + 0 \times log(x_2)...0 \times log(x_n)) = -\frac{1}{n}log(x_1)$ vidimo da ovisi samo o x_1. Generalno naša error funkcija ovisi samo o vrijednosti vjerojatnosti elementa na indeksu jedinice u traženom izlazu. To je glavna prednost Cross Entropy error funkcije.

3 Dodatna obrada riječi

Iako naš model radi sa velikom preciznošću i trenira se dovoljno brzo, postoji način da još ubrzamo treniranje. Možemo prije unosa u mrežu dodatno obraditi tekst. Obrađujemo ga tako da izračunamo n-grame i dodamo ih na ulaz. N-gram je skup n riječi koje se pojavljuju zaredom. Da bi shvatili zašto je to korisno pogledajmo primjer teksta "bilo je jako loše na početku, a dobro na kraju" da bi stroj izvukao emocionalan ton pravilno mora shvatit da "jako loše" nosi veću emocionalnu težinu od samo "dobro". Da bi stroj sam shvatio da riječi "jako" + [riječ emocije] se pojave zajedno nose veću težinu trebat će mu dosta vremena treniranja. Ako stvorimo bi-grame(n-gram od dvije riječi) on prima dvije riječi kao jedan ulaz i lakše zaključuje. Takva obrada je pokazala dobre rezultate.

4 Eksperimenti

Za pisanje koda koristit ćemo pytorch paket, radit ćemo analizu sentimenta filmskih recenzija. Pokušat iterativno dobiti što bolje meta-parametre za model.

Postavljanje SEED-a tako da nam nasumične permutacije različitih lista daju iste permutacije.

```
import torch
from torchtext import data, datasets
import torch.optim as optim
import torch.nn as nn
```

```
SEED = 1423

torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True

TEKST = data.Field(tokenize = 'spacy', include_lengths = True)
OZNAKE = data.LabelField(dtype = torch.float)
```

Preuzimanje skupa podataka recenzija i podjela podataka na podatke za testiranje i podatke za treniranje.

```
[ ]: from torchtext import datasets

trening, testiranje = datasets.IMDB.splits(TEKST, OZNAKE)
```

Dodatna podjela podataka za treniranje na podatke za validaciju.

```
[ ]: import random

trening, validacija = trening.split(random_state = random.seed(SEED))
```

Stvaranje vokabulara, stavljamo maksimalnu veličinu vokabulara od 25,000 jedinstvenih riječi. U vokabular inicijaliziramo Glove vektorizacija dimenzije 100.

```
[ ]: TEKST.build_vocab(trening,
                      max_size = 25_000,
                      vectors = "glove.6B.100d",
                      unk_init = torch.Tensor.normal_)

OZNAKE.build_vocab(trening)
```

Odabiremo uređaj na kojem želimo da se treniranje obavi, GPU ako je slobodan u ostalom CPU. Uzimamo iteratore sa veličinama grupa od 64 recenzije.

```
[ ]: device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

treniranje_it, validacija_it, testiranje_it = data.BucketIterator.splits(
    (trening, validacija, testiranje),
    batch_size = 64,
    sort_within_batch = True,
    device = device)
```

RNN klasa koja prima meta-parametre kojima inicijaliziramo našu neuronsku mrežu. Prvi sloj mreže je vektorizacija kojoj šaljemo padding indeks, tj. indeks vektora na kojem su nalazi vektorska vrijednost padding tokena. Sljedeći sloj je RNN sloj, tj. ponavljačka neuronska mreža. Kao što smo ranije spomenuli za ovakav zadatak najbolja je LSTM struktura. Koristimo LSTM mrežu iz pytorch "torch.nn" paketa. Posljednji sloj je linearan tj. sloj koji aktivira naše skrivene slojeve u LSTM sloju. Tako da je njegova ulazna dimenzija jednaka skrivenoj dimenziji puta 2, a izlazna dimenzija je 1. Tj. 0 za "Negativnu" a 1 za

"Pozitivnu" recenziju. U incijalizaciji koristimo regularizacijsku metodu zvanu dropout, ona služi za sprječavanje prenatreniranosti(kao što je argumentirano u [8].

U metodi "forward" koristimo regularizacijsku metoda na vektorizacijskom sloju, pakiramo vektoriziranu recenziju u "torch.nn.utils.rnn.PackedSequence" format više na [7] koji može primiti svaka RNN mreža. Puštamo tu sekvencu kroz RNN sloj dobijemo izlaz u istom formatu i izlaz skrivenog sloja i stanje ćelije. Raspakiramo sekvencu sa "pad_packed_sequence" funkcijom koja nam daje paddane tekste i njihove duljine. Spojimo posljednje skrivene slojeve (skriveno[-2,:,:]) i prve početne skrivene slojeve (skriveno[-2,:,:]).

[ ]:
```
class RNN(nn.Module):
    def __init__(self, vocab_size, vektorizacija_dim, skrivena_dim,␣
↪izlazna_dim, n_slojeva,
                 bidirectional, dropout, pad_idx):

        super().__init__()

        self.embedding = nn.Embedding(vocab_size, vektorizacija_dim,␣
↪padding_idx = pad_idx)

        self.rnn = nn.LSTM(vektorizacija_dim,
                           skrivena_dim,
                           num_layers=n_slojeva,
                           bidirectional=bidirectional,
                           dropout=dropout)

        self.lin = nn.Linear(skrivena_dim * 2, izlazna_dim)

        self.dropout = nn.Dropout(dropout)

    def forward(self, tekst, tekst_duljine):

        embedded = self.dropout(self.embedding(tekst))

        packed_vektorizirano = nn.utils.rnn.
↪pack_padded_sequence(embedded, tekst_duljine)

        packed_izlaz, (skriveno, celija) = self.rnn(packed_vektorizirano)

        izlaz, izlaz_duljine = nn.utils.rnn.
↪pad_packed_sequence(packed_izlaz)

        skriveno = self.dropout(torch.cat((skriveno[-2,:,:], skriveno[-1,:
↪,:]), dim = 1))

        return self.lin(skriveno)
```

Definiramo funkciju za preciznost koja uspoređuje predviđeni y i pravi y. I vrati postotak preciznosti.

[ ]:
```
def preciznost(preds, y):
    #zakruzeno na najblizi cijeli broj
    zaokruzena_pred = torch.round(torch.sigmoid(preds))
    tocno = (zaokruzena_pred == y).float()

    prec = tocno.sum() / len(tocno)
    return prec
```

Zatim definiramo funkciju za treniranje koja uzima iterator i iterira skupinu po skupinu recenzija koje model obrađuje. Modelu šaljemo tekstove(recenzije) i njihove duljine, dobijemo predviđanje te njega šaljemo ranije definiranoj funkciji koja računa preciznost. Zatim radimo povratnu propagaciju sa izračunatom greškom, te napravimo korak optimizatora. Pamtimo vrijednost greške i preciznosti. Funkcija vraća ukupnu grešku i preciznost.

[ ]:
```
def treniraj(model, iterator, optimizator, fja_greske):

    epoch_greska = 0
    epoch_prec = 0

    model.train()

    for batch in iterator:

        optimizator.zero_grad()

        tekst, tekst_duljine = batch.text

        pred = model(tekst, tekst_duljine).squeeze(1)

        greska = fja_greske(pred, batch.label)

        prec = preciznost(pred, batch.label)

        greska.backward()

        optimizator.step()

        epoch_greska += greska.item()
        epoch_prec += prec.item()

    it_dulj = len(iterator)

    return epoch_greska / it_dulj, epoch_prec / it_dulj
```

Funkcija za evaluaciju je slična funkciji za treniranje. Razlika je da model stavimo u stanje za evaluaciju tako da nam se parametri ne bi mijenjali. Kao i kod treniranja modelu

šaljemo tekstove sa njihovim duljinama dobijemo predviđanje. Izračunamo preciznost te pamtimo vrijednost greške i preciznosti. Povratna propagacija se ovdje ne radi. Vračamo ukupnu grešku i preciznost.

[ ]:
```
def evaluiraj(model, iterator, fja_greske):

    epoch_greska = 0
    epoch_prec = 0

    model.eval()

    with torch.no_grad():

        for batch in iterator:

            tekst, tekst_duljine = batch.text

            pred = model(tekst, tekst_duljine).squeeze(1)

            greska = fja_greske(pred, batch.label)

            prec = preciznost(pred, batch.label)

            epoch_greska += greska.item()
            epoch_prec += prec.item()

    it_dulj = len(iterator)

    return epoch_greska / it_dulj, epoch_prec / it_dulj
```

Definiramo jednostavnu funkciju koja iz vrijednosti za vrijeme računa i vrača protekle minute i sekunde.

[ ]:
```
import time

def epoch_vrijeme(pocetak, kraj):
    proteklo_vrj = kraj - pocetak
    protekle_min = int(proteklo_vrj / 60)
    protekle_sec = int(proteklo_vrj - (protekle_min * 60))
    return protekle_min, protekle_sec
```

Funkcija “odradi_treniranje” uzima model i u 5 epocha ga trenira. Funkcijama za treniranje i validaciju šalje primjerene iteratore, naš trenutni model i funkciju greške. Funkciji za treniranje također šalje optimizator. Računa proteklo vrijeme između početka treniranja i evaluacije i njihovog kraja. Pamti sve vrijednosti koje su vratile funkcije za treniranje i evaluaciju i sprema ih u polje objekata “model_obj”. Funkcija vrača “model_obj”.

[ ]:
```
def odradi_treniranje(model, optimizator, greska, treniranje_it,␣
 ↪validacija_it):
```

```
    najbolja_greska = float('inf')

    model_obj = { "tr_greska" : [], "tr_prec" : [], \
                  "val_greska" : [], "val_prec": [], \
                  "epoch_mins" : [], "epoch_secs": []}

    for epoch in range(5):

        print(epoch)
        pocetak = time.time()

        tr_greska, tr_prec = treniraj(model, treniranje_it, optimizator,␣
↪greska)
        val_greska, val_prec = evaluiraj(model, validacija_it, greska)

        kraj = time.time()

        epoch_mins, epoch_secs = epoch_vrijeme(pocetak, kraj)

        model_obj["tr_greska"].append(tr_greska)
        model_obj["tr_prec"].append(tr_prec)
        model_obj["val_greska"].append(val_greska)
        model_obj["val_prec"].append(val_prec)
        model_obj["epoch_mins"].append(epoch_mins)
        model_obj["epoch_secs"].append(epoch_secs)

    return model_obj
```

Sljedeći dio koda stvara tri modela. Modelima mjenjamo broj slojeva da provjerimo razliku između preciznosti i greške. Na samom početku petlje definirani su meta-parametri modela. Može se jednostavno vidjeti koje su njihove vrijednosti. Padding indeks dobijamo funkcijom "stoi"(string to index) koja vrača cijeli broj koji pripada određenom stringu, a string za padding token je zapamćen unutar objekta "TEKST".

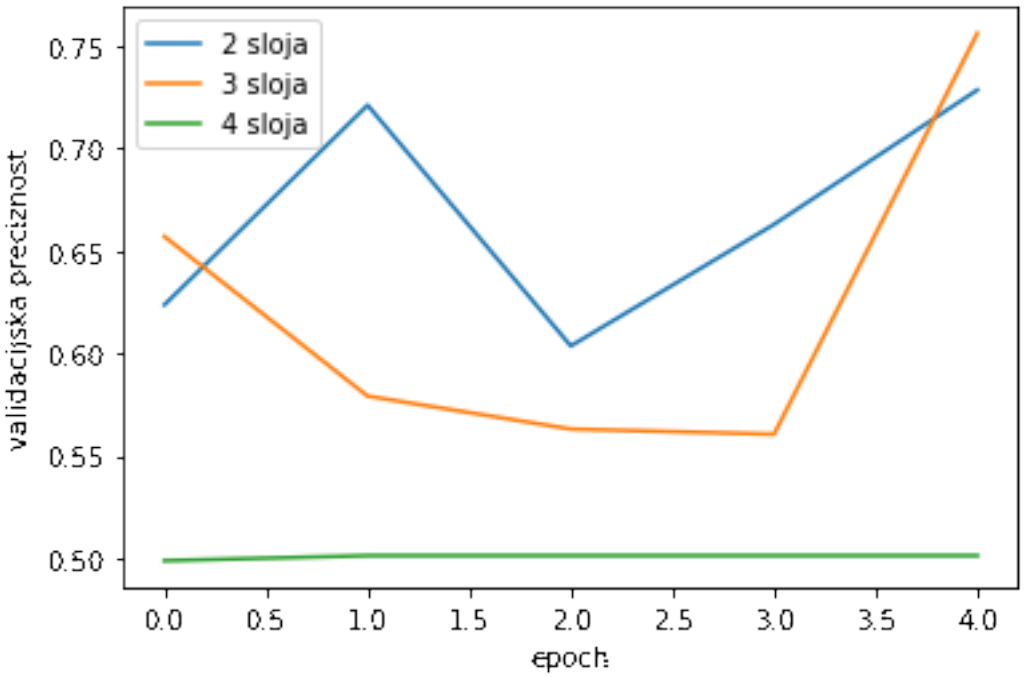
Pomoću meta-parametara zatim stvaramo model. Na analogan način na kojim smo dobili indeks padding tokena uzimamo indeks "UNK" tokena. Vrijednosti oba tokena inicijaliziramo na nule.

Za optimizator uzimamo Adam optimizator koji pomaže u ubrzavanje treniranja kao što je argumentirano u [citat].

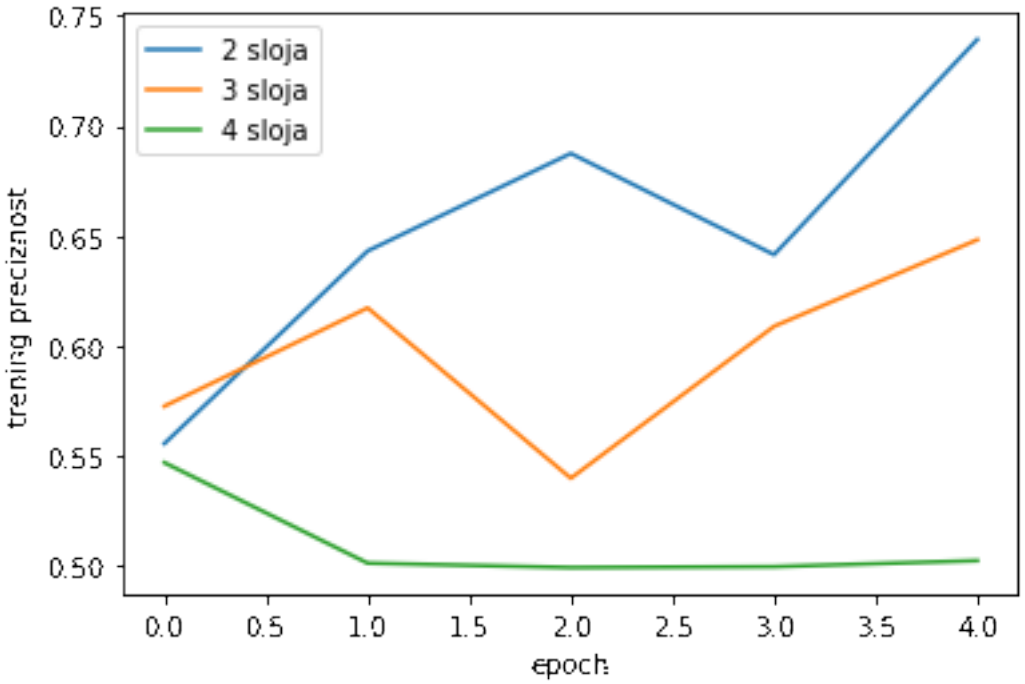
Zatim funkciju greške i model prebacujemo na GPU. Naš model, optimizator, funkciju greške i iterator šaljemo "odradi_trening" funkciji. Ona vrati objekt traženih vrijednosti koji dodamo listi "modeli".

[ ]:
```
modeli = []

for num in range(3):
    #metaparametri
    ULAZNA_DIM = len(TEKST.vocab)
    VEKTORIZACIJSKA_DIM = 100
    SKRIVENA_DIM = 200
    IZLAZNA_DIM = 1
    N_SLOJEVA = num + 2
    BIDIRECTIONAL = False
    DROPOUT = 0.5

    #padding indeks
    PAD_IDX = TEKST.vocab.stoi[TEKST.pad_token]

    model = RNN(ULAZNA_DIM,
                VEKTORIZACIJSKA_DIM,
                SKRIVENA_DIM,
                IZLAZNA_DIM,
                N_SLOJEVA,
                BIDIRECTIONAL,
                DROPOUT,
                PAD_IDX)

    UNK_IDX = TEKST.vocab.stoi[TEKST.unk_token]

    model.embedding.weight.data[UNK_IDX] = torch.
↪zeros(VEKTORIZACIJSKA_DIM)
    model.embedding.weight.data[PAD_IDX] = torch.
↪zeros(VEKTORIZACIJSKA_DIM)

    optimizator = optim.Adam(model.parameters())

    greska = nn.BCEWithLogitsLoss()

    model = model.to(device)
    greska = greska.to(device)

    model_podatci = odradi_treniranje(model, optimizator, greska,␣
↪treniranje_it, validacija_it)
    modeli.append(model_podatci)
```

Pogledajmo validacijsku i trening preciznost ta tri modela i usporedimo.

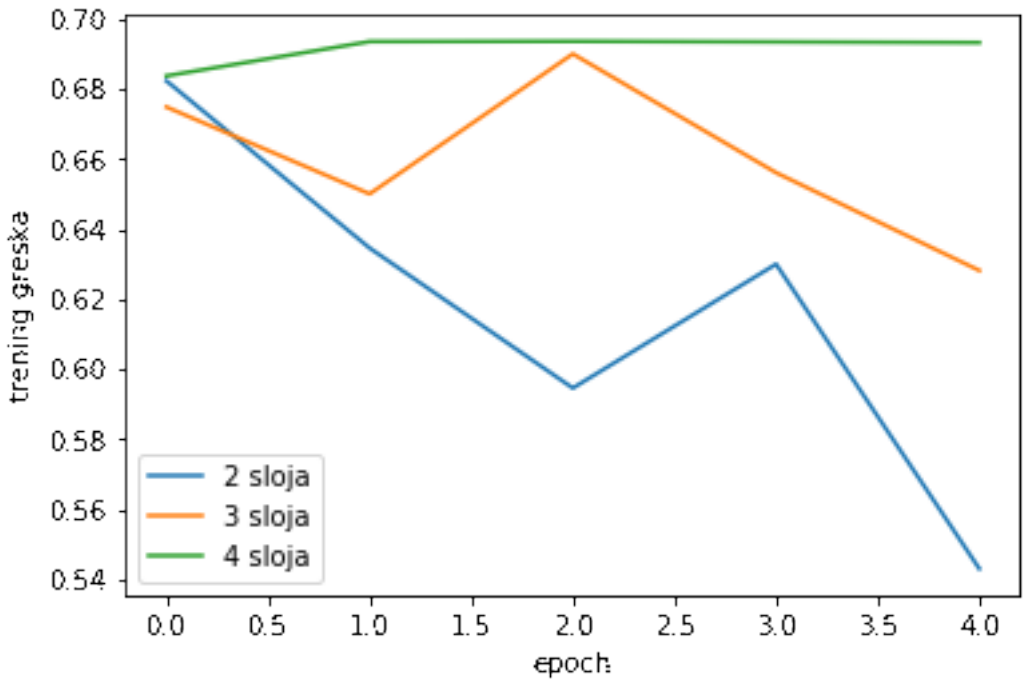


Slika 8: Grafovi validacijske preciznosti(za slojeve)

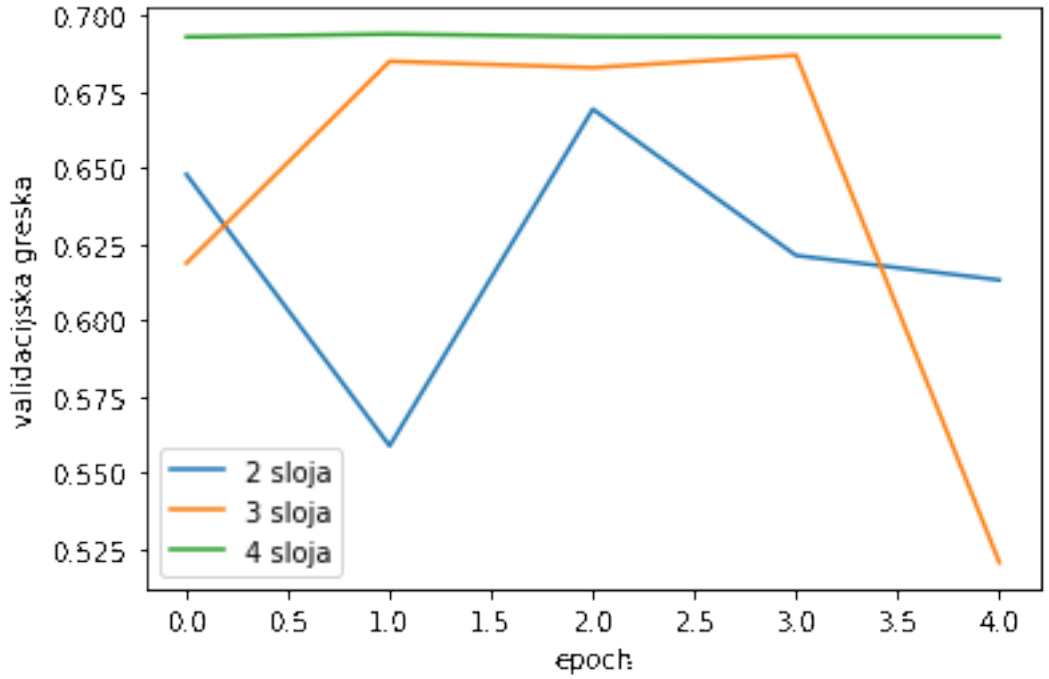


Slika 9: Grafovi trening preciznosti(za slojeve)

Vidimo da su rezultati najbolji za model sa samo dva sloja. Ako pogledamo grafove greške vidimo da oni daju slične rezultate. Tj. greška je najmanja(najbolja) za model sa samo dva sloja.



Slika 10: Grafovi trening greški(za slojeve)



Slika 11: Grafovi validacijska greški(za slojeve)

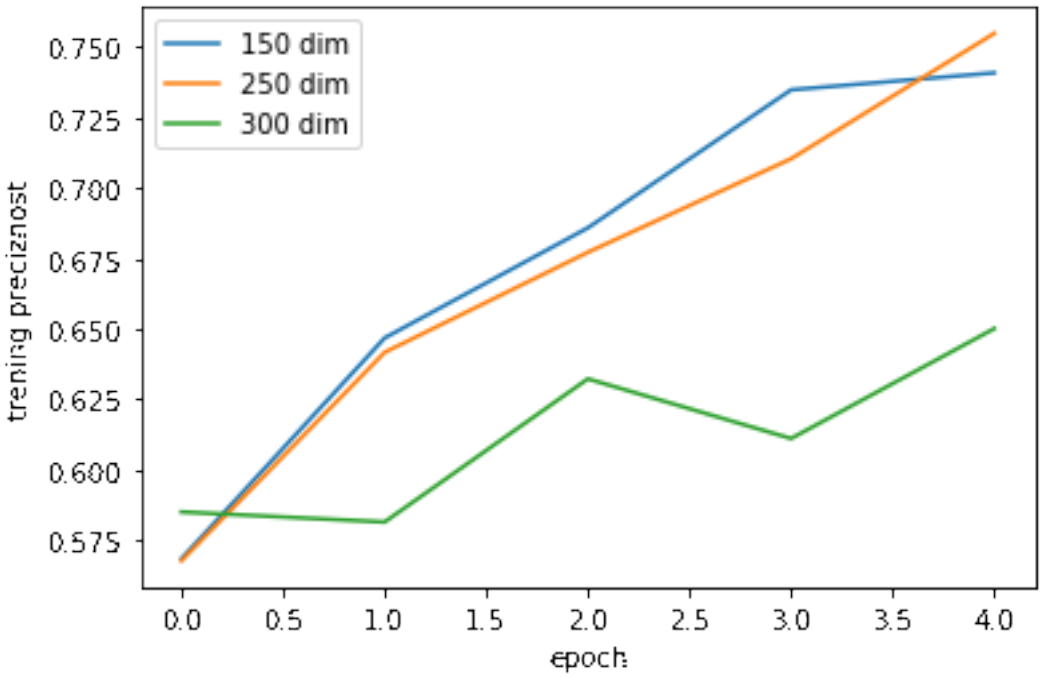
Pošto nam je model sa dva sloja dao najbolje rezultate probat ćemo promijeniti druge meta-parametre radi dodatnog poboljšanja modela.

Sljedeći dio koda analogan je prijašnjem samo se umjesto broja slojeva koji je sada konstantan mijenja dimenzija skrivenog sloja. Na kraju se podatci o modelima spremaju u listu

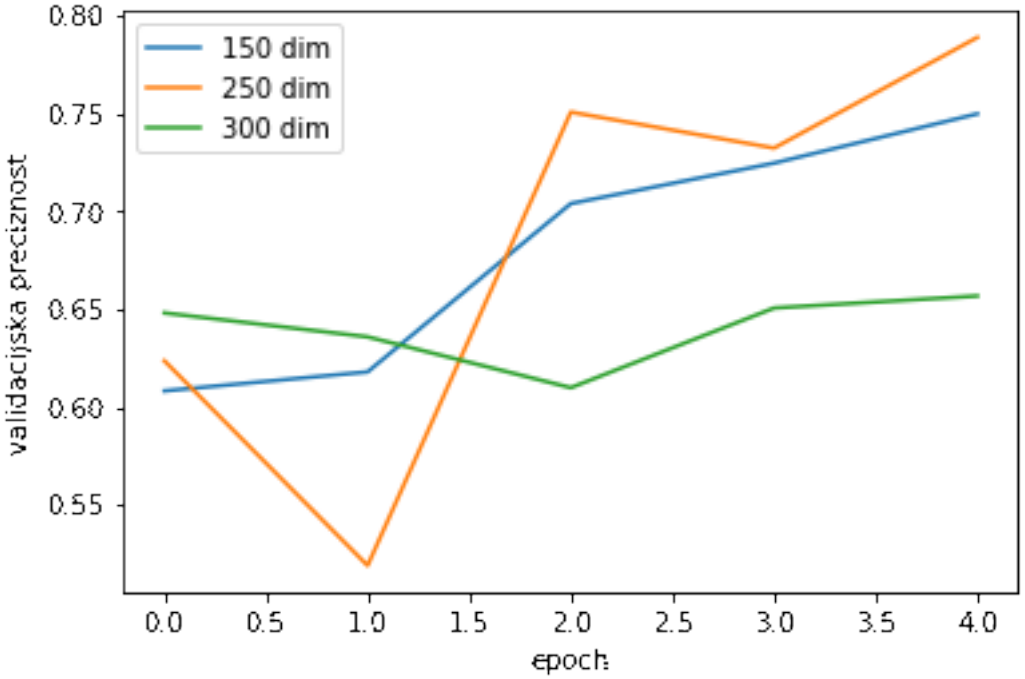
"modeli_dim".

[ ]:
```
modeli_dim = []

for num in range(3):
    ULAZNA_DIM = len(TEKST.vocab)
    VEKTORIZACIJSKA_DIM = 100
    SKRIVENA_DIM = 150 if num == 0 else num*50 + 200
    IZLAZNA_DIM = 1
    N_SLOJEVA = 2
    BIDIRECTIONAL = False
    DROPOUT = 0.5
    PAD_IDX = TEKST.vocab.stoi[TEKST.pad_token]

    model = RNN(ULAZNA_DIM,
                VEKTORIZACIJSKA_DIM,
                SKRIVENA_DIM,
                IZLAZNA_DIM,
                N_SLOJEVA,
                BIDIRECTIONAL,
                DROPOUT,
                PAD_IDX)

    UNK_IDX = TEKST.vocab.stoi[TEKST.unk_token]

    model.embedding.weight.data[UNK_IDX] = torch.
↪zeros(VEKTORIZACIJSKA_DIM)
    model.embedding.weight.data[PAD_IDX] = torch.
↪zeros(VEKTORIZACIJSKA_DIM)

    optimizator = optim.Adam(model.parameters())

    greska = nn.BCEWithLogitsLoss()

    model = model.to(device)
    greska = greska.to(device)

    model_podatci = odradi_treniranje(model, optimizator, greska,␣
↪treniranje_it, validacija_it)
    modeli_dim.append(model_podatci)
```

Pogledajmo sada validacijsku i trening preciznost za nova tri modela i usporedimo.

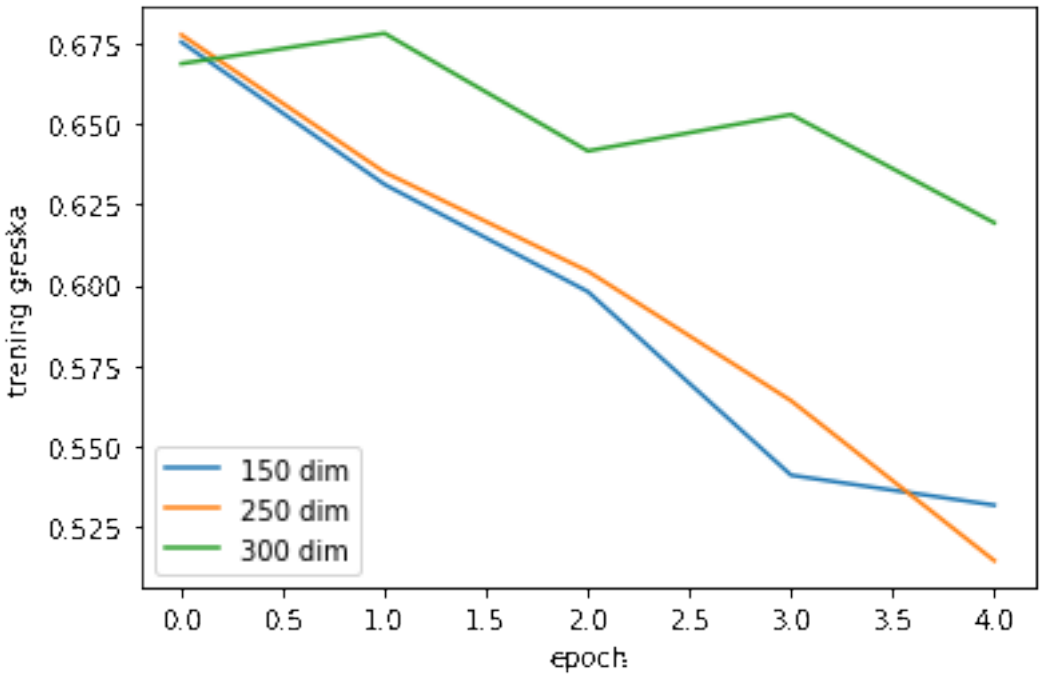


Slika 12: Grafovi trening preciznosti(za dimenzije)

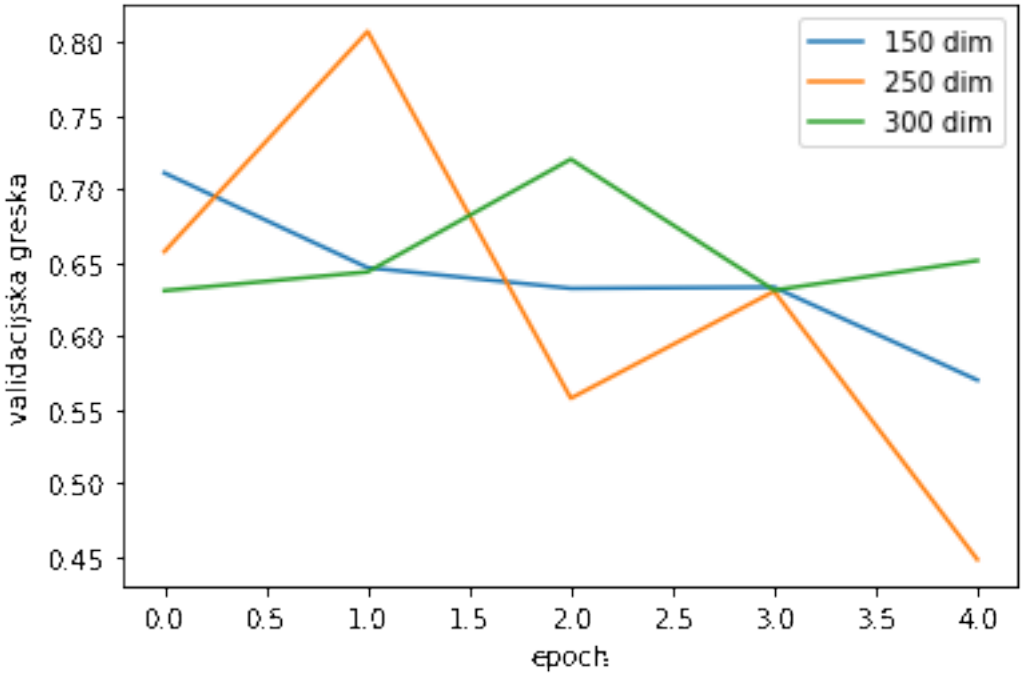


Slika 13: Grafovi validacijske preciznosti(za dimenzije)

Preciznost je najbolja za model sa skrivenim slojevima dimenzije 250, kao i u prethodnom dijelu graf greške nam daje slične rezultate.



Slika 14: Grafovi trening greške(za dimenzije)



Slika 15: Grafovi validacijske greške(za dimenzije)

U našem slučaju model sa skrivenim slojevima dimenzije 250 je dao najbolje rezultate. Ti meta-parametri možda ne budu proizvoljni za druge primjene analize sentimenta.

5 Zaključak

Iako su metode i alati navedeni u ovom radu korisni i jako dobro predviđaju ton tekst, svaki zadatak zahtjeva dodatno istraživanje da bi se točnost modela približila 100%. Iako sve ima svoje matematičke temelje ne možemo točno izračunati koji će meta-parametri biti najbolji za naš zadatak. Dobro je iterativno mijenjati određene značajke modela i empirički zaključiti što najviše pomaže u našem zadatku. Ovaj rad je kolekcija najkorisnijih alata za analizu sentimenta ali pošto su NLP i strojno učenje teme na kojima se radi puno istraživanja ove informacije možda budu zastarjele za nekoliko godina.

Popis slika

Literatura

[1] https://medium.com/datadriveninvestor/how-do-lstm-networks-solve-the-problem-of-vanishing-gradients-a6784971a577 10. 02. 2020.

[2] Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean *Efficient Estimation of Word Representations in Vector Space* 16. 01. 2013.

[3] https://dsgiitr.com/blogs/deepwalk/ 10. 02. 2020.

[4] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies

[5] S. Hochreiter, J. Schmidhuber Long Short-Term Memory 1997.

[6] Dupond, Samuel (2019). "A thorough review on the current advance of neural network structures". Annual Reviews in Control. 14: 200–230.

[7] https://pytorch.org/docs/stable/generated/torch.nn.utils.rnn.PackedSequence.html #torch.nn.utils.rnn.PackedSequence

[8] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever Ruslan Salakhutdinov(2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting