Problem klasifikacije programskog koda

Patajac, Matija

Undergraduate thesis / Završni rad

2021

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Department of Mathematics / Sveučilište Josipa Jurja Strossmayera u Osijeku, Odjel za matematiku**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:126:228108

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

Matija Patajac

Problem klasifikacije programskog koda

Završni rad

Osijek, 2021.

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

Matija Patajac

Problem klasifikacije programskog koda

Završni rad

Mentor: izv. prof. dr. sc. Domagoj Matijević

Sumentor: Antonio Jovanović

Osijek, 2021.

Source code classification problem

Sažetak

U ovom radu bavit ćemo se problemom klasifikacije programskog koda, gdje nam je cilj pomoću obrade programskog jezika odrediti funkcionalnost nekog isječka koda. Predložit ćemo rješenje u obliku neuronske mreže bazirane na long short-term memory (LSTM) rekurentnoj mreži te dobivene rezultate usporediti s nekoliko radova koji su se bavili ovom temom.

Ključne riječi

Programski kod, obrada programskog jezika, neuronske mreže, LSTM, vektorizacija, attention mehanizam, POJ-104.

Abstract

In this paper we discuss source code classification problem, the goal of which is to determine the functionality of a code snippet using programming language processing. We propose a solution in the form of a neural network based on the long short-term memory (LSTM) recurrent network and compare our results to several other papers that had discussed this topic.

Key words

Source code, programming language processing, neural networks, LSTM, embedding, attention mechanism, POJ-104.

Sadržaj

Uvod

Prošlog smo desetljeća doživjeli nagli rast popularnosti strojnog učenja, odnosno neuronskih mreža. Njihova primjena proširila se diljem industrije - u medicini, bankarstvu, automobilskoj industriji itd.
Jedna od primjena je pomoć i interakcija s ljudima kroz digitalne asistente i *chatbotove*. Za njihov je razvitak bio potreban razvoj alata za obradu prirodnog jezika (engl. *natural language processing*, NLP). Iako su ti alati razvijani primarno s ciljem obrade *prirodnog* jezika (jezika kojima ljudi međusobno komuniciraju, npr. hrvatski, engleski, njemački itd.), njihova upotreba moguća je u sklopu bilo čega što možemo nazvati jezikom.
Mi ćemo te alate primjeniti u problemu klasifikacije programskog koda, gdje ćemo koristeći obradu *programskog* jezika pokušati odrediti funkcionalnost programa prema njegovom programskom kodu. Predložit ćemo rješenje u obliku neuronske mreže bazirane na *long short-term memory* (LSTM) rekurentnoj mreži te dobivene rezultate usporediti s nekoliko drugih radova koji su se bavili ovom temom.

Na početku rada opisat ćemo promatrani problem, njegov značaj u svijetu softverskog inženjerstva, a i izvan njega, te nešto reći o korištenom skupu podataka i našem programskom rješenju.
Nakon toga ćemo se upoznati s neuronskim mrežama kroz njihov najjednostavniji oblik, *feedforward* neuronsku mrežu - opisati kako ona radi te koja nas njena ograničenja sprječavaju da ju koristimo u ovoj domeni. Poglavlje završavamo kratkim opisom *rekurentnih* neuronskih mreža - vrste mreža koje zadovoljavaju sve zahtjeve našeg problema.
Nastavljamo analizom arhitekture našeg modela - navođenjem i kratkim opisom korištenih komponenti te pregledom konačne arhitekture modela.
Rad zaključujemo opisom postupka učenja našeg modela, navođenjem dobivenih rezultata te usporedbom sa srodnim radovima.

1 O problemu

U ovom radu bavit ćemo se problemom klasifikacije programskog koda. Jednostavno rečeno, želimo za dani isječak programskog koda reći koja je njegova funkcionalnost, odnosno što on radi.
Takav se problem možda nekima čini jednostavnim, budući da se programeri svakodnevno nalaze u situacijama gdje moraju odrediti funkcionalnost koda, no pritom zaboravljaju na brojna dogovorena vizualna pomagala koje često uzimamo zdravo za gotovo, kao što su

- formatiranje koda - dodavanje razmaka, uvlaka te praznih redova kako bi kod bio pregledniji,
- sintaktički naglašivač (engl. *syntax highlighter*) - bojenje riječi i znakova prema ulozi,
- smisleni, prikladni i semantički bogati nazivi varijabli, funkcija, struktura itd.,

kao i godine iskustva u radu s nekim programskim jezikom te programiranjem općenito.
Bez navedenih pomagala, kod bi izgledao ovako[1]:

```
void a(int*b,int c){int d,e,f;for(d=0;d<c-1;d++)for(e=0;e<c-d-1
;e++)if(*(b+e)>*(b+e+1)){f=*(b+e);*(b+e)=*(b+e+1);*(b+e+1)=f;}}
```

Upravo tako programski kod izgleda nenaučenom modelu.

Iako je prepoznavanje funkcionalnosti koda problem s kojim se programeri često susreću, to ga ne čini lakim - štoviše, čak je i vrlo iskusnim programerima potrebno od nekoliko minuta pa sve do nekoliko dana kako bi odredili što program radi (ovisno o složenosti programa, upoznatosti programera s korištenim programskim jezikom, tehnologijama itd.). Iz tog je razloga automatska klasifikacija programskog koda poželjna funkcionalnost, no kao što ćemo ubrzo vidjeti, to nije njena jedina primjena.

1.1 Značaj teme

Sposobnost klasifikacije programskog koda uvelike bi pomogla programerima u razvoju točnog, sigurnog i efikasnog koda, no njena korisnost nadilazi svijet softverskog inženjerstva. Navedimo neke od mogućih primjena.

Programerski alati

Najviše koristi od ovakve funkcionalnosti imali bi ljudi čiji je posao pisanje koda - programeri. Statička analiza programa omogućuje programerima da izbjegnu česte greške i *antiobrasce* (engl. *anti-patterns*), no njen je nedostatak to što obuhvaća samo vrlo općenite slučajeve. Kada bi programsko okruženje moglo odrediti funkcionalnost napisanog koda, moglo bi upozoriti na sigurnosne propuste koji su česti kod te funkcionalnosti ili preporučiti efikasniju implementaciju iste.

[1] Radi se o C/C++ implementaciji popularnog `Bubble sort` algoritma.

Prepoznavanje funkcionalnosti koda također bi programerima olakšao prelazak na novu bazu koda - ne bi morali čitati kod red po red ili listati dokumentaciju (pod pretpostavkom da ona uopće postoji) kako bi shvatili što koji dio koda radi, već bi mogli (npr. postavljanjem kursora na naziv funkcije/klase) pročitati koja je njegova funkcionalnost.

Problem ranije spomenute nepostojeće dokumentacije također bi se mogao riješiti automatskim prepoznavanjem funkcionalnosti koda - razvijanjem dodatnog modela koji bi prema određenoj funkcionalnosti automatski generirao pripadnu dokumentaciju. Na taj bi način osigurali da je dokumentacija

(a) uvijek prisutna - ne bi ju morali pisati programeri te

(b) uvijek ažurna - izbjegli bismo situacije gdje programer promjeni kod, a zaboravi potom ažurirati postojeću dokumentaciju.

Kompajlerske optimizacije

Dobrobiti klasifikacije programskog koda nisu ograničene samo na pisanje koda. Kompajlerske optimizacije trenutno se najčešće provode na razini naredbe ili bloka naredbi. Poznavanje funkcionalnosti koda omogućilo bi kompajleru da provede posebne optimizacije namijenjene baš toj funkcionalnosti (do razine da potpuno promijeni implementaciju) kako bi osigurao da se konačni program izvodi što je efikasnije moguće.

Antivirusni programi

Jedna od primjena koja nije povezana sa softverskim inženjerstvom je rad antivirusnih programa. Tipičan način rada antivirusnog programa je da provjeri nalazi li se datoteka u bazi poznatih virusa[2]. Nedostatak takvog pristupa je teško detektiranje novih virusa - da bi se stvorila zaštita, netko mora primjetiti virus, dodati ga u postojeću bazu antivirusnog programa te korisnici moraju ažurirati baze na svojim uređajima. Ukoliko bi antivirusni program mogao iz samog programa odrediti je li on maliciozan ili bezopasan (prema njegovoj funkcionalnosti), mogao bi detektirati virus bez da se s njim ikad prije susreo.

1.2 Korišteni skup podataka

Budući da je univerzalno određivanje funkcionalnosti programskog koda iznimno ambiciozan i zahtjevan zadatak, u programskom smo se rješenju[3] odlučili ograničiti na jedan skup podataka. Konkretno, koristimo POJ-104[4] skup podataka koji se sastoji od 104 problema i 500 rješenja svakog problema napisanih u C/C++ programskom jeziku. POJ-104 skup podataka prikupljen je i prvotno korišten u [7], a potom i u [2], stoga smo i mi odlučili koristiti taj skup kako bismo mogli napraviti značajnu usporedbu s navedenim radovima.

[2] https://whatis.techtarget.com/definition/virus-signature-virus-definition

[3] Izvorni kod programskog rješenja možete pronaći ovdje: https://github.com/mpatajac/algorithm-classification

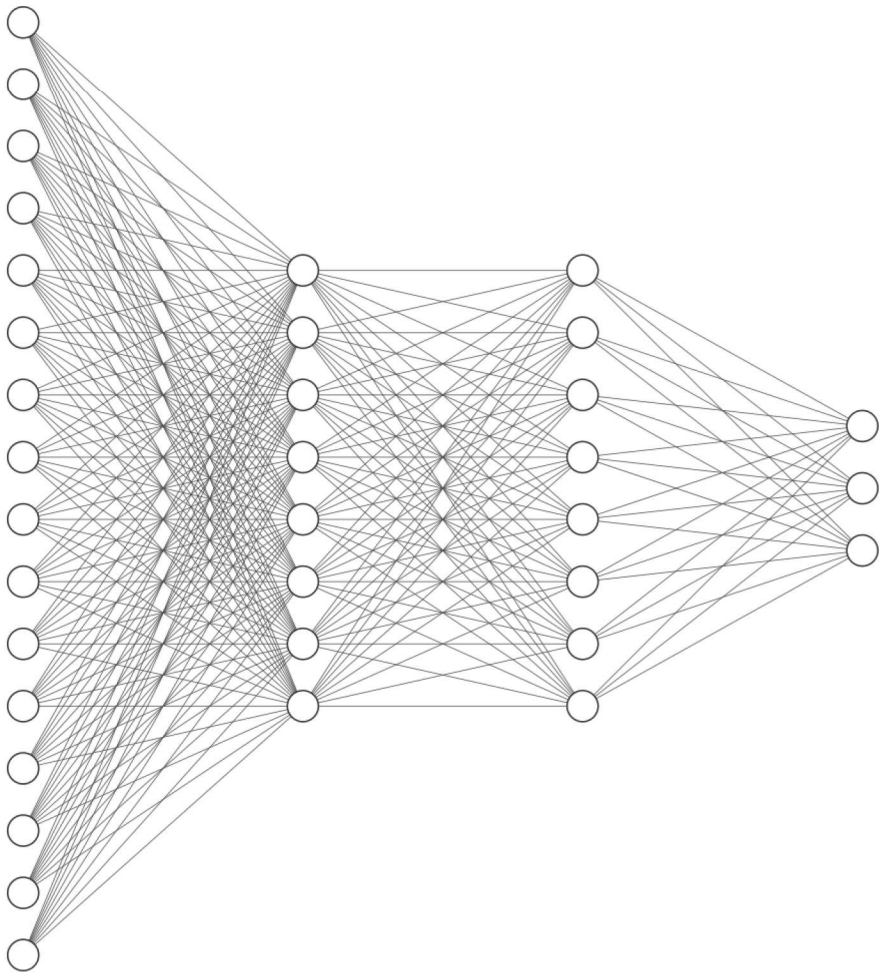
[4] Poveznica na podatke: https://sites.google.com/site/treebasedcnn/

2 Odabir mreže

U ovom poglavlju pogledat ćemo najosnovniju vrstu neuronske mreže: *feedforward* neuronsku mrežu. Proučit ćemo kako ona funkcionira i navesti neka njena ograničenja zbog kojih nije prigodna za korištenje u ovakvom tipu problema. Za kraj ćemo opisati *rekurentnu* neuronsku mrežu čija struktura nema spomenuta ograničenja, zbog čega ju koristimo u našem modelu.

2.1 *Feedforward* neuronska mreža

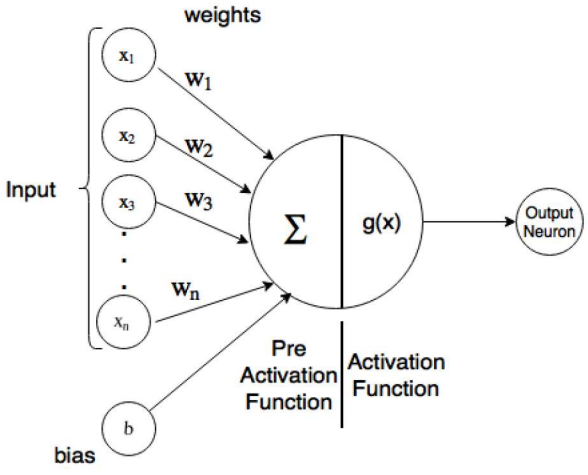
Feedforward neuronska mreža (FFNN) sastoji se od dva ili više *sloja*, koji se sastoje od skupa *aktivacijskih jedinica* (njih često još nazivamo i *neuronima*). Prvi sloj u mreži predstavlja ulazne podatke, odnosno parametre koje oni sadrže, dok zadnji sloj sadrži *izlaze*, odnosno rezultate naše mreže. Nerijetko FFNN ima više od dva sloja kako bi se povećala složenost modela, a samim time i njegova *točnost* - tada slojeve između prvog i zadnjeg nazivamo *skriveni* slojevi.



Slika 1: Primjer *feedforward* neuronske mreže s ulaznim slojem od 16 aktivacijskih jedinica, izlaznim slojem od 3 jedinice te 2 skrivena sloja s po 8 jedinica.

Vrijednost unutar svake aktivacijske jedinice (izuzev onih iz prvog sloja) dobiva se kao linearna kombinacija svih aktivacijskih jedinica iz prethodnog sloja (iz tog se razloga ovakve mreže često još nazivaju i *potpuno povezane* (engl. *fully connected*) ili *guste* (engl. *dense*)). Kada bismo kao vrijednost gledali samo linearnu kombinaciju prethodnog sloja, ograničili bismo snagu svog modela - ma koliko skrivenih slojeva stavili u mrežu, konačan rezultat bi opet bio linearna kombinacija ulaza, što je jednak rezultat kao da smo imali mrežu samo s ulaznim i izlaznim slojem. Iz tog se razloga u svakoj aktivacijskoj jedinici na rezultat dobiven linearnom kombinacijom još primjenjuje i *aktivacijska funkcija*, čiji je cilj uvesti nelinearnost u model i time povećati broj funkcija koje naš model može predstavljati.
Iako se kao aktivacijska funkcija teoretski može koristiti bilo koja (nelinearna) funkcija, među najčešće korištenima su:

- sigmoid:
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$
- tangens hiperbolni:
$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$
- *rectified linear unit* (ReLU):
$$ReLU(x) = max\{0,\ x\}$$



Slika 2: Unutrašnja struktura jedne aktivacijske jedinice [5]

[5]Izvor: https://miro.medium.com/max/460/1*uyIJXyLOI4-ETDmcWFsKvQ.png

Često korištene oznake su:

- x_i - i-ta ulazna jedinica (tj. i-ti parametar ulaznog podatka)
- $w_{ij}^{(l)}$ - težina između j-te aktivacijske jedinice l-tog i i-te jedinice $(l+1)$-og sloja (npr. $w_{15}^{(2)}$ je težina koja ide u prvu aktivacijsku jedinicu trećeg sloja, iz pete jedinice drugog sloja)
- g - aktivacijska funkcija
- $a_j^{(l)}$ - vrijednost j-te aktivacijske jedinice l-tog sloja prije primjene aktivacijske funkcije (linearna kombinacija $(l-1)$-og sloja)
- $z_j^{(l)}$ - vrijednost j-te aktivacijske jedinice l-tog sloja nakon primjene aktivacijske funkcije
- s_l - broj aktivacijskih jedinica u l-tom sloju

Prateći navedene oznake, račun unutar FFNN možemo zapisati na sljedeći način:

$$z_i^{(1)} = x_i$$

$$a_i^{(l)} = \sum_{j=0}^{s_l} w_{ij}^{(l-1)} z_j^{(l-1)}$$

$$z_i^{(l)} = g(a_i^{(l)})$$

2.1.1 Ograničenja

Feedforward neuronske mreže sjajne su kod problema kao što je određivanje cijene stana (prema kvadraturi, lokaciji, broju soba...), određivanje kvalitete vina (prema tipu, količini alkohola, kiselosti...), utvrđivanje je li osoba bolesna (prema nalazima rendgena, EKG-a, izvađene krvi...), odnosno općenito problema kod kojih se ulazni podaci sastoje od *konačnog broja unaprijed poznatih parametara.* To nas dovodi do sljedeća dva ograničenja takve mreže.

Ograničen broj ulaznih vrijednosti
Programi nemaju konačan broj unaprijed poznatih parametara - sastoje se od naredbi čiji broj i uloga znatno variraju ovisno o programskom jeziku, problemu koji rješavaju, pa i o samoj implementaciji.

Jedan način na koji bismo mogli "riješiti" taj problem je da odaberemo neki broj i toliko naredbi uzmemo u obzir, no vrlo brzo uviđamo nove probleme koji su nastali kao rezultat te odluke.
Koji broj naredbi odabrati? Ako je premalen, izgubit ćemo većinu informacije potrebne za donošenje odluke. S druge strane, prevelik broj naredbi će zahtjevati *popunjavanje* kraćih programa, čime ćemo znatno produljiti vrijeme učenja modela.
Kako odabrati korištene naredbe? S početka, kraja, sredine, slijedom, nasumičnim odabirom...?
Iako rješenje ovog problema postoji, uvjerili smo se da je daleko od idealnog.

Gubitak unutrašnje strukture podataka
Računalo vidi naš program kao niz naredbi koje izvršava određenim *redom*. Iako je taj *red* samo implicitna informacija, u našem je problemu ona vrlo bitna - često upravo red kojim se naredbe izvršavaju određuje koja će biti konačna funkcionalnost programa. To možemo vidjeti već na ovom jednostavnom primjeru:

```
a := 3
b := 2 * a
a := 7
```

Nakon izvršavanja ove tri naredbe varijabla `a` poprima vrijednost 7, a varijabla `b` vrijednost 6.

```
a := 3
a := 7
b := 2 * a
```

Zamjenom druge i treće naredbe varijabla `a` i dalje ima vrijednost 7, no varijabla `b` sada ima vrijednost 14.

```
a := 7
b := 2 * a
a := 3
```

Zamjenom prve i treće naredbe sada varijabla `a` ima vrijednost 3, a varijabla `b` vrijednost 14.

Iako smo koristili iste tri naredbe, svaki od tri redoslijeda izvođenja dao nam je različite rezultate. Sličan fenomen pojavljuje se i u analizi prirodnog jezika - ako pogledamo rečenice

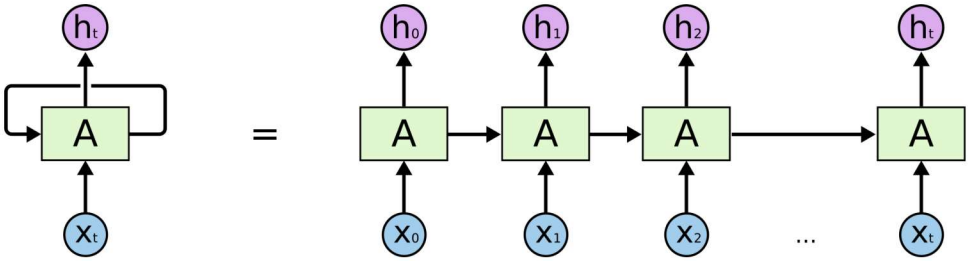
Film je bio jako dobar, nimalo loš.
Film je bio jako loš, nimalo dobar.

vidimo da unatoč tome što obje rečenice koriste potpuno iste riječi, imaju suprotno značenje.

Koriteći *feedforward* neuronsku mrežu gubimo informaciju o *redu* - ona tretira sve ulazne parametre kao izolirane, bez ikakve međusobne strukture, a to je nešto što, kao što gornji primjeri sugeriraju, ne možemo prihvatiti.

2.2 Rekurentna neuronska mreža

Za razliku od *feedforward* neuronske mreže, *rekurentna* neuronska mreža (RNN) nema ranije spomenuta ograničenja. Štoviše, njen način rada upravo odgovara našim zahtjevima. Naime, ona podatak prima kao niz *vremenskih koraka* (engl. *time steps*) - time nam omogućuje korištenje podataka varijabline duljine, kao i očuvanje *reda* kojim se izvršava program kroz koncept vremena. Kao što možete pretpostaviti, mi ćemo kao jedan vremenski korak koristiti jednu naredbu programa.



Slika 3: Lijevo: *skupljena* RNN. Desno: *razmotana* (engl. *unfolded*) RNN[6]

Način na koji rekurentne mreže rade je da koriste *skrivena stanja* (engl. *hidden states*). Skriveno stanje sadrži prikupljene informacije o dosadašnjim elementima ulaznog niza te se, uz sljedeći korak ulaza, koristi za stvaranje novog skrivenog stanja:

$$h_t = rnn(x_t,\ h_{t-1})$$

gdje je h_t skriveno stanje u koraku t, x_t ulaz u koraku t te h_{t-1} skriveno stanje u koraku $t-1$. Kao početno skriveno stanje (h_0) često se koristi nulvektor.

[6]Izvor: https://towardsdatascience.com/introduction-to-recurrent-neural-network-27202c3945f3

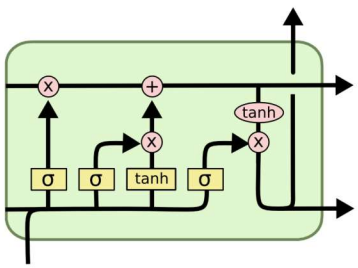
3 Arhitektura modela

U ovom poglavlju proučit ćemo *arhitekturu* modela - koje komponente sadrži i na koji su način povezane.
Analizu arhitekture našeg modela započet ćemo LSTM-om i *attention* mehanizmom - ključnim komponentama koje čine središnji dio našeg modela. Zatim ćemo reći nešto o *dropoutu*, *embeddinzima* i *linearnom sloju*, pomoćnim komponentama koje transformiraju ulaz i izlaz modela tako da čini smislenu cjelinu, te ćemo za kraj pogledati kako su sve te komponente međusobno povezane.

3.1 Komponente

LSTM

Jezgru našeg modela čini LSTM (skraćeno od *long short-term memory*) - vrsta *rekurentne* neuronske mreže (RNN).
Obične RNN pate od problema znanih kao *nestajući*, odnosno *eksplodirajući* gradijent (engl. *vanishing and exploding gradient*) - gradijenti u procesu minimizacije funkcije troška počnu težiti u 0 ili u ∞. Način na koji to tumačimo je da RNN najbolje "pamti" one korake koji su se dogodili nedavno, dok one koji su se dogodili nešto ranije zaboravlja, što je veliki problem kod podataka s velikim brojem koraka kod kojih može postojati povezanost između koraka s velikim vremenskim razmakom.
LSTM je osmišljen upravo kako bi spriječio taj problem. To postiže korištenjem *stanja ćelije*[7] - dodatnog skupa parametara čiji je cilj "dugoročno pamćenje". Stanje ćelije kontroliraju troja *vrata* (engl. *gates*): ulazna, izlazna i *zaboravna* (engl. *forget*).



Slika 4: Unutrašnja struktura jedne LSTM *ćelije*.[8]

Za svaki korak u ulaznom nizu, stanja se računaju na sljedeći način:

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

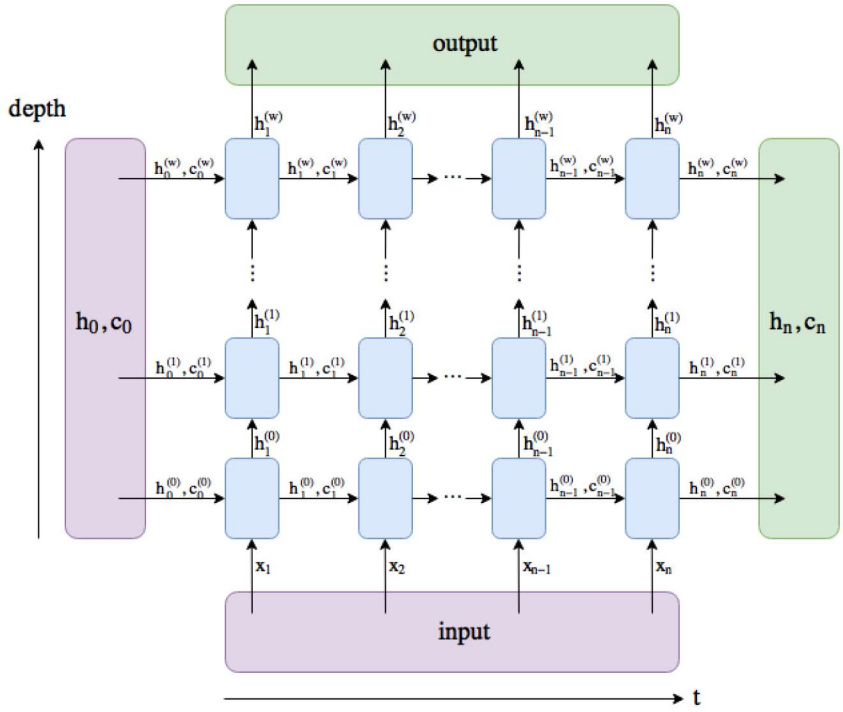
[7] Ćelijom nazivamo skup operacija kojim rekurentna mreža kombinira korak ulaza te prijašnje skriveno stanje kako bi stvorila novo skriveno stanje.
[8] Izvor: http://colah.github.io/images/post-covers/lstm.png

gdje je h_t skriveno stanje u koraku t, c_t stanje ćelije u koraku t, x_t ulaz u koraku t, h_{t-1} skriveno stanje u koraku $t-1$ (ili početno stanje, ako smo u nultom koraku), i_t, f_t, g_t, o_t su ulazna, zaboravna, ćelijska i izlazna *vrata*, σ je sigmoid funkcija, a $\odot$ Hadamardov produkt (produkt "po elementima").

Osim osnovnog oblika LSTM-a, koriste se i neke njegove složenije verzije[9].

Višeslojni LSTM

Često je praksa umjesto korištenja samo jednog LSTM-a *naslagati* (engl. *stack*) njih više uzastopno - u tom slučaju svaki od njih nazivamo *slojem*, odakle dolazi naziv *višeslojni* LSTM (engl. *multi-layer LSTM*).



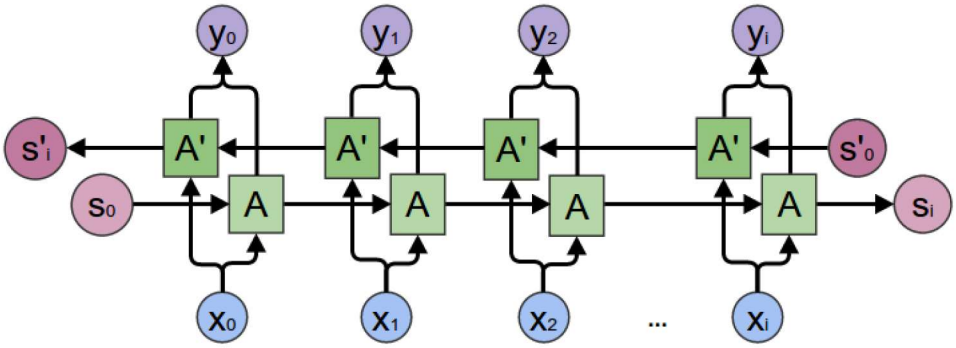
Slika 5: Struktura višeslojnog LSTM-a.[10]

Višeslojni LSTM radi tako što prvi sloj prima ulazni niz i obrađuje ga kao što bi i inače. Tokom obrade ulaza, prvi sloj LSTM-a stvara svoje *izlaze* (skrivena stanja na svakom koraku ulaznog niza). Kod jednoslojnog LSTM-a bismo ovdje stali, uzeli zadnje skriveno stanje (stanje nakon zadnjeg koraka ulaznog niza) te ga dalje koristili u svom modelu. No, kod višeslojnog LSTM-a postupak je sljedeći: izlaze koje je generirao prvi sloj sada tretiramo kao novi ulazni niz te ga dajemo drugom sloju na obradu. Drugi sloj tokom obrade novog niza generira svoje izlaze, koje opet dajemo trećem sloju kao ulaz. Taj postupak ponavljamo sve dok ne dođemo do zadnjeg sloja - tek tada uzimamo zadnje skriveno stanje i njega dalje koristimo u modelu.

[9]Ekvivalentne verzije prisutne su i kod drugih vrsta rekurentnih mreža.
[10]Izvor: https://i.stack.imgur.com/SjnTl.png

Dvosmjerni LSTM

Budući da LSTM podatke prima linearno (korak po korak, od početka do kraja), u svakom trenutku "zna" samo ono što je dosad "vidio", odnosno korake koji su došli *prije* trenutnog. Nekad je to dovoljno, no nekad je za bolje razumijevanje potrebno i ono što dolazi *nakon* trenutnog koraka. Iz tog se razloga ponekad koristi *dvosmjerni* LSTM (engl. *bidirectional LSTM*).



Slika 6: Prikaz rada dvosmjernog LSTM-a.[11]

Dvosmjerni LSTM možemo promatrati kao dva zasebna LSTM-a od kojih jedan podatak prima normalno (kao što bi običan LSTM), dok drugi prima isti taj podatak, samo u obratnom poretku (od zadnjeg koraka prema prvom). Zadnje skriveno stanje oba LSTM-a se najčešće konkatenira te tretira kao skriveno stanje čitavog (dvosmjernog) LSTM-a. Time se postiže efekt da LSTM "vidi" čitavo okruženje - ono što se nalazi i prije, a i poslije trenutnog koraka.

Pridruživanje pozornosti pojedinim naredbama

LSTM je značajan napredak od obične rekurentne mreže, ali svejedno s njom dijeli veliko ograničenje - svu informaciju o ulaznom nizu mora pohraniti u jedno (zadnje) skriveno stanje. Kako bismo izašli na kraj s tim ograničenjem, predložen je `attention` mehanizam.
Ideja `attention` mehanizma je da radi sa svim skrivenim stanjima LSTM-a i svakom pridruži određenu *pozornost*, odnosno odredi koje stanje (pa onda i pripadna naredba u nizu) koliko utječe na donošenje odluke.

S vremenom su se razvile razne verzije `attention` mehanizma - mi koristimo modificiranu[12] verziju Loungova mehanizma (vidi [6]) s *generalnim* načinom računanja *alignment scorea*[13]. Započinjemo određivanjem pozornosti koju će `attention` mehanizam pridružiti t-toj naredbi u nizu - izlazno stanje LSTM-a nakon t-te naredbe o_t pomnožimo s konačnim skrivenim stanjem h, pomnožimo matricom težina W_a i primjenimo $softmax$ funkciju:

[11] Izvor: http://colah.github.io/posts/2015-09-NN-Types-FP/img/RNN-bidirectional.png

[12] Verzija predložena u radu namijenjena je modelima koji niz pretvaraju u niz (*seq2seq*), stoga su bile potrebne izmjene kako bi mehanizam radio na klasifikacijskom modelu.

[13] Uz pomoć *alignment scorea* određuje koliko će *pozornosti* dati kojem koraku (naredbi).

$$a(t) = \frac{\exp(W_a(o_t \cdot h))}{\sum_{j=1}^{T} \exp(W_a(o_j \cdot h))}$$

Koristeći dobivene vrijednosti određujemo *kontekst* čitavog ulaznog niza:

$$c = \sum_{j=1}^{T} a(j) o_j$$

Konačni vektor parametara dobivamo konkatenacijom konteksta c sa skrivenim stanjem h, množenjem novom matricom težina W_c te primjenom funkcije *tanh*:

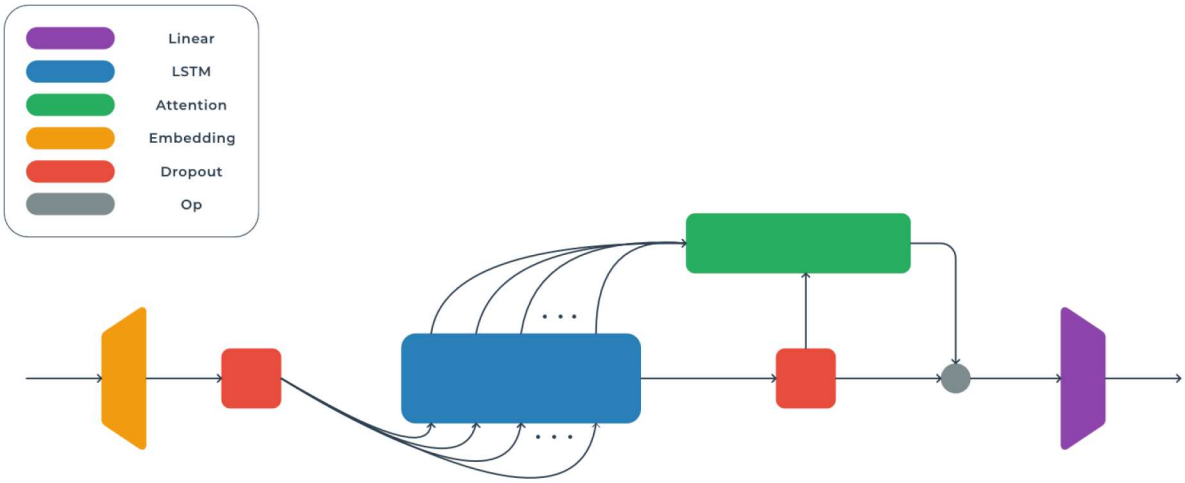
$$H = \tanh\left(W_c \begin{bmatrix} c \\ h \end{bmatrix}\right)$$

Reprezentacija naredbi kao višedimenzionalnih vektora
Kako bismo modelu olakšali učenje problema, često ulazne podatke reprezentiramo kao višedimenzionalne vektore - podatke ugrađujemo (engl. *embedding*) u d-dimenzionalni vektorski prostor. Drugim riječima, svakom podatku pridružujemo d vrijednosti kojima opisujemo taj podatak, odnosno određujemo izraženost svakog od d svojstava u pojedinom podatku. Koristeći ta svojstva možemo odrediti sličnost, odnosno povezanost dvaju naredbi.
Iako je ručni odabir svojstava i određivanje pripadnih vrijednosti za svaku naredbu moguće, bilo bi vrlo zahtjevno i neefikasno, stoga to ostavljamo modelu kao dio procesa učenja, odnosno dodatan set parametara.

Linearni sloj
Iako smo u prošlom poglavlju pokazali da *feedforward* neuronsku mrežu ne možemo koristiti kao središnji dio našeg modela, od koristi će nam biti *linearni sloj* (engl. *linear layer*) kao pomoćna komponenta. Linearni sloj je *feedforward* neuronska mreža koja se sastoji samo od ulaznog i izlaznog sloja aktivacijskih jedinica, bez skrivenih slojeva i aktivacijskih funkcija. Često se koristi kako se bi unutrašnje stanje čitavog modela mapiralo u konačni vektor odabira kategorija.

Regularizacija poništavanjem nasumično odabranih vrijednosti
Posljednja komponenta koju koristimo je `dropout` - ona s vjerojatnošću p neke od elemenata ulaznog vektora postavi na 0. To se pokazalo kao efektivna tehnika *regularizacije* (vidi [4]) (tj. sprječavanja prenaučenosti modela).

3.2 Model



Slika 7: Konačna arhitektura modela

Model na ulazu prima izvorni kod programa kojeg promatramo kao niz naredbi. Započinjemo tako da naredbe reprezentiramo kao višedimenzionalne vektore - svakoj naredbi pridružimo određen broj vrijednosti kojima opisujemo izraženost pojedinog svojstva. Neke od elemenata dobivenih vektora postavimo na 0 s ciljem regularizacije.
Transformirani ulaz dajemo LSTM-u, koji za svaku naredbu u nizu vraća odgovarajuće skriveno stanje. U slučaju dvosmjernog LSTM-a, skrivena stanja koja je vratio prolaz u jednom smjeru konkateniramo sa skrivenim stanjima dobivenim prolaskom u drugom smjeru, čime se broj skrivenih stanja udvostručuje. Ukoliko koristimo višeslojni LSTM, stanja koja je vratio prvi sloj koristimo kao novi niz kojeg dajemo drugom sloju, izlaz drugog sloja kao ulaz trećem itd. Kao konačan rezultat LSTM-a koristimo skrivena stanja iz zadnjeg sloja. Neke elemente konačnog skrivenog stanja postavimo na 0 s ciljem regularizacije.
Sva skrivena stanja (*izlaz* LSTM-a) kao i skriveno stanje vraćeno nakon zadnje naredbe u nizu (*zadnje* skriveno stanje) prosljeđujemo `attention` mehanizmu - on nam vraća novi vektor parametara, dimenzije jednake onoj zadnjeg skrivenog stanja LSTM-a. Ovisno o zadanim hiperparametrima, parametre `attention` mehanizma povezujemo sa zadnjim skrivenim stanjem LSTM-a tako da ih

- zbrojimo (čime dimenzija ostaje ista) ili
- konkateniramo (čime se dimenzija udvostručuje).

Za kraj nad dobivenim vektorom parametara provodimo *softmax* regresiju kako bismo završili proces klasifikacije (tj. utvrdili odabranu kategoriju).

4 Rezultati

Za pripremu podataka pratili smo postupak iz [7] i [2] - skup smo podataka podijelili na 3 dijela (za učenje, validaciju i testiranje) u omjeru 3 : 1 : 1. Također smo kao u [2] umjesto originalnog C/C++ koda koristili LLVM IR[14] instrukcije te kod u skupu za učenje kompajlirali 8 puta koristeći različite optimizacijske parametre kompajlera (engl. *compiler flags*): `-O{0-3}` i `-ffast-math`.

Eksperimentalnom provjerom odabrali smo sljedeće hiperparametre modela:

- instrukcije su ugrađivane u 100-dimenzionalni vektorski prostor
- korišten je 2-slojni, jednosmjerni LSTM s 200 skrivenih stanja
- `attention` parametri konkatenirani su na skrivena stanja dobivena iz LSTM-a
- `dropout` je korišten s vjerojatnošću $p = 0.2$

Model smo učili 7 *epoha* (prolazaka kroz cijeli skup podataka za učenje), pri čemu smo nakon svake epohe model evaluirali na skupu podataka za validaciju kako bismo osigurali da model uči općenita svojstva problema, a ne specifičnosti skupa podataka za učenje (tj. spriječili *prenaučenost* (engl. *overfitting*)). Kao funkciju troška (engl. *loss function*) koristili smo *funkciju maksimalne vjerodostojnosti* (engl. *maximum likelihood function*)[15], a minimizaciju te funkcije ubrzali smo koristeći optimizator Adam (vidi [5]) sa zadanim hiperparametrima. U konačnici smo model evaluirali na skupu podataka za testiranje.

4.1 Usporedba sa srodnim radovima

Dobivene rezultate usporedimo s `tree-based convolutional neural network` (TBCNN, [7]) i `inst2vec` (NCC, [2]) - dvama modelima iz radovima koji su ponudili vlastita rješenja za ovaj problem.

ručna obrada [7]	TBCNN [7]	inst2vec [2]	**naš model**
88.2	94.0	94.83	94.67

Tablica 1: Točnost pojedinog modela na testnom skupu, izražena u postocima.

Kao što možemo vidjeti iz Tablice 1, naš model daje rezultate usporedive s onima iz TBCNN [7] i NCC [2], no da bismo uvidjeli značaj ovog rezultata, valja istaknuti jednu bitnu razliku našeg modela od onih iz drugih radova.

[14] https://llvm.org/docs/LangRef.html

[15] https://towardsdatascience.com/probability-concepts-explained-maximum-likelihood-estimation-c7b4342fdbb1

Unaprijed naučeno ugrađivanje i sofisticirane reprezentacije

Za razliku od našeg modela, modeli predloženi u [7] i [2] koriste *unaprijed naučeno ugrađivanje* (engl. *pre-trained embeddings*). Prisjetimo se, u procesu ugrađivanja ulazne podatke reprezentiramo kao d-dimenzionalne vektore - svakom podatku pridružujemo d vrijednosti kojima opisujemo taj podatak, odnosno određujemo izraženost svakog od d svojstava u pojedinom podatku. Kojih d vrijednosti ćemo pridružiti kojem podatku određuju parametari pridruženi komponenti ugrađivanja. Što bolje parametre imamo, bolje ćemo opisati podatke te će ih naš model lakše kategorizirati.
Upravo iz tog razloga su [7] i [2] svoje komponente ugrađivanja unaprijed naučili - dali su im velike količine podataka[16] kojima su komponente naučile kako dobro reprezentirati programski kod, a potom su te (već naučene) komponente iskoristili u svojim modelima.

Osim što su unaprijed naučili svoje komponente ugrađivanja, koristili su znatno složenije reprezentacije koda - TBCNN ([7]) pri reprezentaciji koristi *apstraktno sintaksno stablo* (engl. *abstract syntax tree*, *AST*)[17] čitavog C/C++ programa, dok je NCC ([2]) otišao korak dalje te stvorio *graf kontekstnog toka* (engl. *contextual flow graph*, *XFG*), reprezentaciju koja prati povezane podatke te kontrolu toka naredbi.

Unatoč tome što smo u našem modelu koristili relativno jednostavnu reprezentaciju podataka koju nismo unaprijed trenirali, već koristili nasumično odabrane vrijednosti iz $\mathcal{N}(0, 1)$ (standardne normalne distribucije), postigli smo točnost usporedivu s TBCNN ([7]) i NCC ([2]), čime smo pokazali da za uspješnu klasifikaciju nisu potrebne sofisticirane komponente ugrađivanja, nego je dovoljan i standardan LSTM model.

[16] NCC koristi ~ 50 milijuna linija koda ([2], poglavlje 5.1, Tablica 1), dok za [7] podaci nisu pronađeni.
[17] https://en.wikipedia.org/wiki/Abstract_syntax_tree

Literatura

[1] Charu C. Aggarwal. *Neural Networks and Deep Learning.* Springer, Cham, 2018.

[2] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 3588–3600. Curran Associates, Inc., 2018.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org`.

[4] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors, 2012.

[5] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[6] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation, 2015.

[7] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing, 2015.