SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1675

Dekompajliranje Java aplikacija potpomognuto metodama strojnog učenja

Mihovil Kucijan

Zagreb, ožujak 2019.

Umjesto ove stranice umetnite izvornik Vašeg rada.
Da bi ste uklonili ovu stranicu obrišite naredbu `\izvornik`.

SADRŽAJ

1. Uvod

U današnje vrijeme okruženi smo velikom količinom računala koja konstantno izvršavaju razne aplikacije radi pružanja traženih usluga. Pri tome računala posjeduju ogromne količine povjerljivih informacija koje je potrebno zaštititi kako bi se očuvala njihova povjerljivost. U svrhu veće sigurnosti aplikacija potrebno je provoditi sigurnosne analize i penetracijska ispitivanja. Međutim, aplikacije koje se analiziraju najčešće nemaju na raspolaganju svoj izvorni kod, već samo izvršni kod, što otežava analizu. Kako bi se postupak analize pojednostavio te povećala efikasnost analize, koriste se razne tehnike reverznog inženjerstva. Glavni tipovi tehnika uključuju analizu tokova podataka, deasembliranje te dekompajliranje.

Dekompajliranje je proces prevođenja izvršnog programa u njegov izvorni kod. Izvršni kod je najčešće dan u binarnom obliku, te se teško razaznaje kakav je sadržaj njegovog programa. Dekompajliranje pretvara jezik niske razine apstrakcije u jezik visoke razine apstrakcije. To se postiže tako da programi za dekompajliranje pokušavaju provesti obratan proces od onog koji se provodi prevođenjem u izvršni kod. Kako se dio informacija prilikom prevođenja gubi, dekompajleri nisu u mogućnosti savršeno rekonstruirati izvorni kod, stoga dijelovi programa često ostanu nejasno prevedeni.

Proces prevođenja izvornog jezika u strojni je kompleksan, te ovisan o jeziku koji se prevodi i platformi za koju se prevodi. Osim toga, postoji više od jednog načina da se program napisan u specifičnom programskom jeziku prevede u njegov izvršni ekvivalent. Posljedica toga je veliki broj dekompajlera koji se razlikuju po svojoj metodologiji i uspješnosti. Najčešći pristup je izgradnja dekompajlera za specifičan prevoditelj, gdje se uzima u obzir način prevođenja specifičnih izraza i nastoje se prepoznati obrasci niza instrukcija koje bi generirao specifičan prevoditelj te ih se reducira u višu razinu apstrakcije. Postoje i dekompajleri opće namjene za specifičan jezik i platformu, ali generalno imaju nižu točnost iako svladavaju neke dijelove prevođenja bolje [12][17]. Zbog opsežnosti ovog područja u ovom radu ćemo se ograničiti samo na Java programski jezik i platformu.

Pojavom strojnog učenja otvorile su se nove mogućnosti za pristupanje ovom pro-

blemu. Strojno učenje je vrsta umjetne inteligencije koja uglavnom koristi statističke metode kako bi računalu dala mogućnost učenja. Strojno učenje se razvilo iz područja raspoznavanja uzoraka i teorije računalnog učenja te se zasniva na algoritmima koji svoju učinkovitost poboljšavaju na temelju empirijskih podataka. Korištenjem neuronskih mreža do sada se pokazalo kako je moguće prepoznati početke i krajeve funkcije u izvršnom kodu [26], te tipove povratnih vrijednosti [9]. Također, prikazane su ideje o cjelovitijim postupcima za dekompajliranje gdje se koristi genetski algoritam kako bi se izgradio izvorni kod koji se prevodi u ekvivalentan izvršni program [25]. Nadalje, tehnike poput probabilističkog modela koristile su se za deobfuskaciju Android aplikacija, pomoću kojega se, trenirajući na skupu izvornih programa, pokušalo predvidjeti prikladno ime za razrede, metode i varijable [4].

Ovaj rad će se fokusirati na uporabu modela dubokog učenja koji se često koriste pri prevođenju prirodnih jezika kako bi se istražila njihova primjena na umjetnim jezicima poput programskog jezika Java. Za ovaj rad koristile su se povratne neuronske mreže kako bi se klasificirale instrukcije koje pripadaju izrazima viših programskih jezika.

U sljedećim poglavljima ovog rada opisana je struktura i rad programskog jezika Java te opis korištenih metoda neuronskih mreža. Nadalje su opisani eksperimenti koji su se provodili te dobiveni rezultati. Zadnje poglavlje je zaključak rada.

2. Programski jezik Java

U ovom poglavlju opisan je programski jezik Java, njegov dizajn, svojstva i struktura te kako se izvršava i funkcionira njegova radna okolina.

2.1. Povijest jezika

Programski jezik Java osmišljen je u tvrtki Sun Microsystems početkom 90-tih godina prošlog stoljeća. Razvoj jezika počeo je 1990. godine kao *Stealth project* (nevidljivi, prikriveni projekt), poslije nazvan *Green project* (zeleni projekt), a vodili su ga Bill Joy, Patrick Naughton, Mike Sheridan i James Gosling.

Prve inačice programskog jezika Java izlazile su pod nazivom *Oak* (hrast), naziv koji je Gosling dao prema starom hrastu koji je rastao u dvorištu ispred prozora njegove radne sobe. Od tog naziva se moralo odustati zbog problema oko autorskih prava, nakon čega je Java programski jezik poprimio današnje ime po otoku u Indoneziji poznat po prvim plantažama kave. Projekt je započeo zbog nezadovoljstva s tadašnjim stanjem programskih jezika C/C++, njegovih primjenskih programskih sučelja i alata. No kako se projekt razvijao zahtjevi su se promijenili, od nastojanja da se napravi programski jezik koji je bolji od dotadašnjih jezika, u jezik koji će služiti za primjenu na raznim platformama. Stealth projekt je nastojao predvidjeti smjer razvoja računarske tehnologije. Budućnost se očekivala u vidu nastavka minijaturizacije računalnih komponenti i potrebe za razvojem računarske tehnologije na područje postojećih elektroničkih uređaja kao što je bijela tehnika. Cilj projekta je bio izgradnja sustava koji će omogućiti mrežno povezivanje i međusobnu komunikaciju velikog broja različitih elektroničkih uređaja, a Java je trebala biti jezik koji bi bio prikladan za izgradnju takvog sustava. Prvi program napisan u Java programskom jeziku koristio se za inteligentni daljinski upravljač televizora u sklopu projekta "*7" [**?**]. Proizvod je bio neuspješan pa se Java pokušala upotrebljavati u kabelskoj televiziji, no ni to nije urodilo plodom. Java je zatim bila prilagođena potrebama za sve popularniju okolinu Internet, te je 1995. godine ugrađena podrška za Javu u Internet preglednik Netscape

Navigator, gdje je u konačnici postigla veliki uspjeh.

Sintaksa, kao i mnogo svojstava, temeljena su na programskim jezicima C/C++, ali izostavljene su neke mogućnosti radi povećanja sigurnosti i jednostavnosti. Radi mogućnosti izvođenja na različitim platformama, Java je zamišljena da se izvodi u virtualnom stroju koji komunicira s operacijskim sustavom kako bi izvodio naredbe zapisane u Java programu.

2.2. Svojstva programskog jezika Java

Glavni ciljevi Java programskog jezika su sigurnost i prenosivost na različite platforme. Zbog toga Java ima sljedeća svojstva: jednostavnost, objektna paradigma, distribuiranost, prenosivost i interpretiranje, robusnost, sigurnost, višedretvenost, dinamičnost.

Jednostavnost se očituje po prepoznatljivoj sintaksi, bogatim standardnim programskim knjižnicama i dobro definiranim obrascima izvođenja operacija.

Objektna paradigma je programska paradigma koja se temelji na konceptu objekta. Objekt sadrži podatke koji se nazivaju atributi i sadrži procedure koje se nazivaju metode te definiraju funkcionalnost objekta. Java definira strukturu objekta u razredu. Svaka izvorna datoteka programskog jezika Java predstavlja jedan razred. Pri prevođenju, za svaki razred se generira datoteka izvornog koda, u kojoj je održana struktura razreda, a metode i atributi su odvojeni prema njihovim nazivima.

Prenosivost i interpretiranje su važna svojstva koja su karakteristična za Javu. Svojstva se postižu tako da se Java izvršni kod, za razliku od ostalih programskih jezika, ne izvršava izravno na procesoru, nego u Java virtualnom stroju. Java virtualni stroj je program koji interpretira Java izvršni kod, to jest bajtkod (*engl. bytecode*), i izvršava naredbe na računalu. Ovakav pristup omogućuje prenosivost Java programa. Da bi se Java programi izvodili na određenoj platformi potrebno je samo izraditi Java virtualni stroj za određenu platformu. Danas su virtualni strojevi vrlo napredni pa, osim što interpretiraju bajtkod, često korištene dijelove koda izravno prevode i optimiziraju u strojni kod za određenu platformu. To uvelike povećava brzinu izvođenja, a održava prenosivost samih aplikacija.

Robusnost i sigurnost Java programskog jezika se postiže zahtjevima za striktnim deklaracijama tipova i metoda te memorijskim sustavom koji sam vodi brigu o oslobađanju i zauzimanju memorije te ne dopušta programeru izravan pristup memoriji.

Višedretvenost i dinamičnost omogućuje Java programima paralelno izvođenje zadataka, te dinamično nadopunjavanje radnog okruženja što je pogodno za interaktivna okruženja poput Internet mreže.

2.3. Tipovi podataka

Java virtualni stroj, jednako kao i Java programski jezik, prepoznaje dva tipa podataka: primitivni tipovi i reference. Ti tipovi podataka se mogu spremati u varijable, slati kao argumenti, vraćati iz metoda i izvoditi operacije nad njima. Virtualni stroj očekuje da sve provjere tipova budu obavljanje prije izvođenja, obično od strane prevoditelja. Vrijednosti primitivnih tipova ne trebaju biti označene da bi se znao njihov tip za vrijeme izvođenja zbog toga što postoje posebne instrukcije virtualnog stroja za svaki od postojećih primitivnih tipova. U Javi postoje 8 primitivnih tipova. Tablica 2.1 sadrži popis primitivnih tipova i njihov opis.

Java definira povratnu vrijednost (*engl. returnAdress*) kao poseban tip podatka. Kako rad s pokazivačima u Javi nije omogućen, ovaj tip podatka se može naći samo u izvršnom kodu i koristi se s instrukcijama skoka. Kao zamjena za pokazivače, Java koristi reference. Postoje tri tipa referenci u Javi: referenca razreda, polja i sučelja.

Tablica 2.1: Primitivni tipovi

Tip	**Opis**
byte	8-bitni cijeli broj s predznakom koji koristi dvojni komplement.
short	16-bitni cijeli broj s predznakom koji koristi dvojni komplement
int	32-bitni cijeli broj s predznakom koji koristi dvojni komplement
long	64-bitni cijeli broj s predznakom koji koristi dvojni komplement
float	32-bitni decimalni broj jednostruke preciznosti
double	64-bitni decimalni broj dvostruke preciznosti
boolean	tip podatak za prikaz istinitosti, poprima vrijednosti:*true* ili *false*
char	tip podataka za prikaz jednog znaka, 16-bitni cijeli broj bez predznaka

2.4. Java bajtkod

Java bajtkod sastoji se od jednog okteta koji označava tip instrukcije (opcode) i može sadržavati dodatne oktete koji se koriste za izvođenje operacije te se zovu operandi. Broj i veličina operanada ovisi o tipu instrukcije. Ako je veličina operanda veća od jednog okteta sprema se u *Big Endian* poretku, što znači da se najznačajniji bitovi zapisuju prvi, od manjih vrijednosti adresa prema većim.

Bajtkod instrukcije možemo podijeliti u nekoliko grupa:

- Instrukcije za učitavanje i spremanje
- Instrukcije aritmetičkih i logičkih operacija
- Instrukcije konverzije tipa
- Instrukcije za stvaranje i rukovanje objektima
- Instrukcije za upravljanje stogom
- Instrukcije skoka
- Instrukcije za poziv i vraćanje metode

Osim ovih grupa postoji još nekoliko instrukcija koje služe za podizanje iznimki i sinkronizaciju.

S obzirom da se jedan oktet koristi za označavanje pojedine instrukcije, ukupan broj mogućih instrukcija je 256. Od 2015. godine, od mogućih 256 instrukcija su u upotrebi 202 instrukcija, 3 su trajno zauzete radi korištenja u implementaciji virtualnog stroja (impdep1, impdep2, breakpoint), a ostale su slobodne za buduću uporabu.

Većina instrukcija su ovisne o tipu podataka nad kojim mogu izvoditi operacije, a radi lakšeg prepoznavanja u imenu sadrže prefiks ili sufiks ovisno o tipu. U tablici 2.2 može se vidjeti popis instrukcija ovisno o njihovom tipu. Svaki redak tablice označava semantički istu instrukciju naznačenu u prvom stupcu sa oznakom *T* koja predstavlja mjesto u imenu instrukcije na kojem se doznačava tip instrukcije. Ostali stupci sadrže ime instrukcije, ako postoji, za odgovarajući tip.

Vezano uz tip operanda, Java pri vođenju operacija neke tipove ne razlikuje. Uz to instrukcije koje nisu vezane uz tip kao *pop* i *swap* rade na tipovima samo određene kategorije koja je ovisna o veličini tipa podatka. Tako u tablici 2.3 možemo vidjeti podjelu tipova po tipu za izračun i kategoriji. Kategorija 1 označava širinu podatka od 4 okteta, a kategorija 2 širina od 8 okteta.

Tablica 2.2: Instrukcije i podržani tipovi

opcode	byte	short	int	long	float	double	char	reference
Tipush	bipush	sipush						
Tconst			iconst	lconst	fconst	dconst		aconst
Tload			iload	lload	fload	dload		aload
Tstore			istore	lstore	fstore	dstore		astore
Tinc			iinc					
Taload	baload	saload	iaload	laload	faload	daload	caload	aaload
Tastore	bastore	sastore	iastore	lastore	fastore	dastore	castore	aastore
Tadd			iadd	ladd	fadd	dadd		
Tsub			isub	lsub	fsub	dsub		
Tmul			imul	lmul	fmul	dmul		
Tdiv			idiv	ldiv	fdiv	ddiv		
Trem			irem	lrem	frem	drem		
Tneg			ineg	lneg	fneg	dneg		
Tshl			ishl	lshl				
Tshr			ishr	lshr				
Tushr			iushr	lushr				
Tand			iand	land				
Tor			ior	lor				
Txor			ixor	lxor				
i2T	i2b	i2s		i2l	i2f	i2d		
l2T			l2i		l2f	l2d		
f2T			f2i	f2l		f2d		
d2T			d2i	d2l	d2f			
Tcmp				lcmp				
Tcmpl					fcmpl	dcmpl		
Tcmpg					fcmpg	dcmpg		
if_TcmpOP			if_icmpOP					if_acmpOP
Treturn			ireturn	lreturn	freturn	dreturn		areturn

Tablica 2.3: Pravi i izračunski tipovi podataka

Pravi tip	Tip za izračun	Kategorija
boolean	int	1
byte	int	1
char	int	1
short	int	1
int	int	1
float	float	1
referenca	referenca	1
returnAdress	returnAdress	1
long	long	2
double	double	2

2.5. Java izvršna datoteka

Java izvršna datoteka (*engl. Java class file*) sadrži Java bajtkod koji se može izvoditi u Java virtualnom stroju. Java prevoditelj generira izvršne datoteke na temelju izvornog koda. Java izvorni kod ima ekstenziju *.java*, dok izvršni kod koristi ekstenziju *.class*. Ako izvorni kod sadrži više od jednog razreda, svaki od razreda se prevodi u svoju zasebnu *.class* datoteku.

Struktura izvršne datoteke može se vidjeti u kodu 2.1.

```
ClassFile {
    u4             magic;
    u2             minor_version;
    u2             major_version;
    u2             constant_pool_count;
    cp_info        constant_pool[constant_pool_count-1];
    u2             access_flags;
    u2             this_class;
    u2             super_class;
    u2             interfaces_count;
    u2             interfaces[interfaces_count];
    u2             fields_count;
    field_info     fields[fields_count];
    u2             methods_count;
    method_info    methods[methods_count];
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

Kod 2.1: Struktura izvršne datoteke

Svaka *class* datoteka sadrži definiciju jednog razreda ili sučelja. Datoteka je u for-

matu niza okteta, to jest 8 bitova. Sve 16, 32 i 64 bitne vrijednosti su sagrađene od 2, 3 i 4 uzastopnih okteta. U prikazu strukture *class* datoteke u kodu 2.1, u1, u2 i u4 označavaju veličinu podatka kao broj okteta bez predznaka. Svi članovi strukture *class* datoteke u memoriji se nalaze slijedno jedan za drugim bez dopunskih okteta ili poravnanja.

Atributi u strukturi u kodu 2.1 označavaju sljedeće:

- *magic* – Označava magični broj *class* datoteke i iznosi 0xCAFEBABE.
- *minor_version*, *major_version* – Članovi označavaju verziju *class* datoteke, pomoću koje se određuje format same datoteke.
- *constant_pool_count* – Označava broj članova u *constant_pool* tablici povećan za jedan
- *constant_pool* – Tablica konstanti sadrži strukture koje predstavljaju različite vrijednosti nizova znakova, razreda, sučelja, atribute razreda, imena metoda i druge strukture. Format svakog člana tablice je najavljen prvim oktetom, u tablici 2.4 možemo vidjeti popis tipova tablice konstanti. Važno je napomenuti da se indeksiranje tablice počinje s brojem jedan.
- *access_flags* – Ova vrijednost označava vrstu pristupa razredu.
- *this_class* – Vrijednost ovog člana odgovara indeksu u bazenu konstanti koji pokazuje na ime razreda ili sučelja.
- *super_class* – Vrijednost ovog člana treba biti 0 ili indeks tablice konstanti koji pokazuje na ime nadrazreda. Ako je vrijednost 0, ova *class* datoteka mora reprezentirati razred *Object*, pošto je to jedini razred bez nadrazreda, svi ostali imaju barem razred *Object* za roditelja.
- *interface_count, interfaces[]* – Ove strukture označavaju broj i imena sučelja koja implementira ovaj razred.
- *fields_count, fields[]* – Ove strukture označavaju broj, imena i informacije atributa razreda koji su deklarirani.
- *methods_count, methods[]* – Ove strukture označavaju broj metoda, ime metoda, te sve informacije definirane za metode kao niz instrukcija koje izvodi, popis lokalnih varijabli itd.
- *attributes_count, attributes[]* – Strukture označavaju broj atributa i informacija o atributima koje se odnose na cijelu *class* datoteku.

Tablica 2.4: Oznake tablice konstanti

Vrsta konstante	Vrijednost okteta
CONSTANT_Class	7
CONSTANT_Fieldref	9
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_String	8
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_NameAndType	12
CONSTANT_Utf8	1
CONSTANT_MethodHandle	15
CONSTANT_MethodType	16
CONSTANT_InvokeDynamic	18

Atributi su definirani svojom vrstom i duljinom, iza čega slijede sve informacije pohranjene u atribut. Atributi su struktura u *class* datoteci koja služi za kao nosioc informacija. U tablici 2.5 nalazi se popis vrsta atributa te u kojem dijelu strukture *class* datoteke se mogu naći.

Neki od važniji atributa, u kontekstu ovog rada, nalaze se pod imenom *Code* i *LineNumberTable*. Atribut *Code* sadrži niz instrukcija koja određena metoda obavlja, te koji se dobiva prevođenjem izvornog koda. Atribut *LineNumberTable* je tablica koja popisuje pozicije instrukcija iz *Code* atributa i poziciju linije koda iz izvorne datoteke, te tako označava izraz u izvornom kodu od kojeg je instrukcija potekla. Pozicija instrukcije u tablici je označena kao relativan odmak okteta na kojem počinje, uz to da prva instrukcija u metodi uvijek počinje na odmaku 0. Tablica ne sadrži član za svaku instrukciju, nego samo odmak na prvu instrukciju koja je generirana iz određenog izraza na određenoj liniji u izvornom kodu, pa stoga informacija nije potpuna.

Tablica 2.5: Definirane vrste atributa

Vrsta atributa	**Lokacija atributa**
SourceFile	ClassFile
InnerClasses	ClassFile
EnclosingMethod	ClassFile
SourceDebugExtension	ClassFile
BootstrapMethods	ClassFile
ConstantValue	field_info
Code	method_info
Exceptions	method_info
RuntimeVisibleParameterAnnotations, RuntimeInvisibleParameterAnnotations	method_info
AnnotationDefault	method_info
MethodParameters	method_info
Synthetic	ClassFile, field_info, method_info
Deprecated	ClassFile, field_info, method_info
Signature	ClassFile, field_info, method_info
RuntimeVisibleAnnotations, RuntimeInvisibleAnnotations	ClassFile, field_info, method_info
LineNumberTable	Code
LocalVariableTable	Code
LocalVariableTypeTable	Code
StackMapTable	Code
RuntimeVisibleTypeAnnotations, RuntimeInvisibleTypeAnnotations	ClassFile, field_info, method_info, Code

2.6. Java virtualni stroj

Java virtualni stroj je program koji izvodi izvršni Java kod. Stroj je definiran specifikacijom [14] koja definira skup podržanih instrukcija koje su reprezentirane bajtkodom. Specifikacija propisuje samo zahtjeve kojih se implementacija virtualnog stroja mora držati kako bi se Java aplikacije ispravno izvodile, svi ostali implementacijski detalji prepušteni su onima koji razvijaju virtualni stroj. Kako bi se Java bajtkod dekompajlirao u Java izvorni kod potrebno je poznavanje strukture i rada Java virtualnog stroja.

2.6.1. Izvršni sustav

Izvršni sustav Java virtualnog stroja može se podijeliti na šest dijelova:

- programski brojač
- stog Java virtualnog stroja
- stog vanjskih metoda
- memorijska gomila
- tablica konstanti
- područje za pohranjivanje metoda

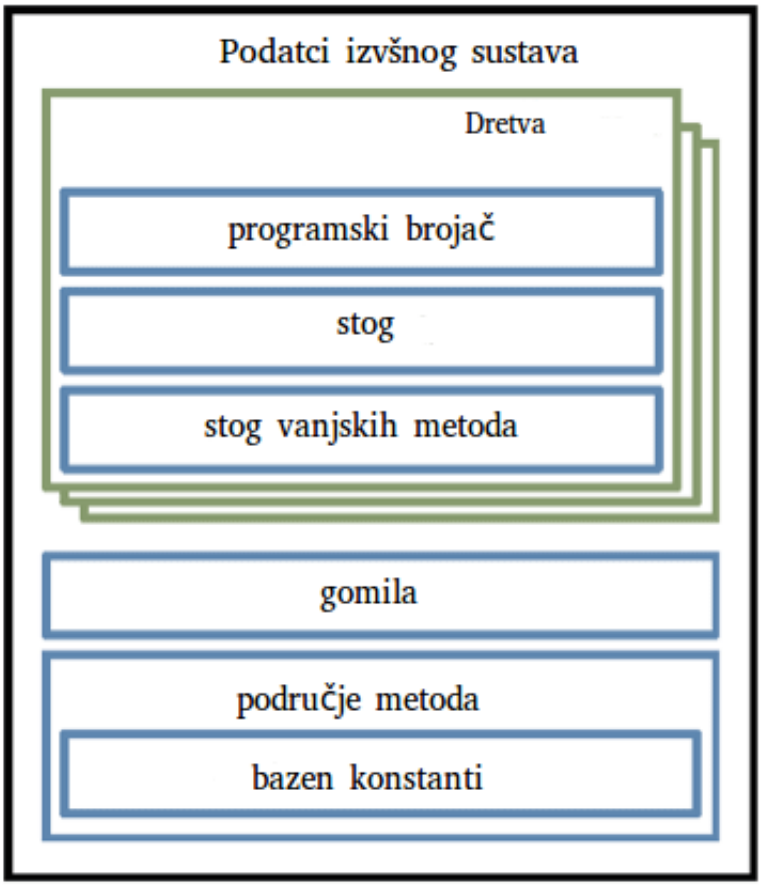
Na slici 2.1 može se vidjeti struktura podataka izvršnog sustava. Podaci su podijeljeni u dvije grupe, oni koji su dijeljeni među dretvama, te oni čiji prostor je specificiran za pojedinu dretvu.

Programski brojač postoji za svaku dretvu i stvara se kada dretva počinje s radom. U njemu je sadržana adresa trenutačne instrukcije koja se izvodi.

U Java virtualnom stroju nema velik broj registara, razlog tome je što bajtkod instrukcije koriste isključivo stog za razmjenu podataka u programu. Stog se stvara za svaku pojedinu dretvu i sadrži referentne okvire koji se stvaraju pri svakom pozivu metode.

Memorijski prostor gomile u Javi se dijeli između svih dretvi. Na gomili se spremaju sve instance razreda i polja u Javi. Gomila se stvara na početku rada virtualnog stroja, a objekti na njoj se brišu isključivo automatskim sustavom brisanja memorijskog smeća (*engl. garbage collector*).

Isto kao gomila, područje za pohranjivanje metoda je dijeljeno među dretvama. Ovo područje služi za pohranu prevedenog koda i analogno je *text* segmentu procesa operacijskog sustava. Virtualni stroj pohranjuje strukture razreda kao tablica konstanti,



Slika 2.1: Struktura podataka izvršnog sustava [23]

varijable i metode razreda i njihove instrukcije. Područje za pohranjivanje metoda dio je gomile u logičkom memorijskom prostoru.

Tablica konstanti u izvršnoj okolini je nastaje na temelju tablice konstanti koja je zapisana u *class* datoteke. Tablica sadrži reference na varijable i metode razreda u obliku imena njihovog paketa i razreda. Osim toga, sadrži razne konstante kao brojevi i nizovi znakova. Tablica je analogna simboličkoj tablici u programskim jezicima kao što su C/C++, ali služi za veći skup različitih tipova podataka.

Stog vanjskih metodi (*engl. Native Method Stacks*) je zaseban stog koji koriste metode pisane u drugim programskim jezicima, primjerice programskom jeziku C. Specifikacija ne zahtijeva postojanje stoga vanjskih metodi ako implementacija nema podršku za poziv metoda pisanih u drugim programskim jezicima.

Okvir (*engl. frame*) se koristi za pohranjivanje podataka i međurezultata te ujedno služi za obavljanje dinamičkog povezivanja, vraćanje povratnih vrijednosti metoda i za slanje iznimki. Sve instrukcije se izvode unutar trenutnog okvira. Pri završetku metode, sa stoga se dohvaća prijašnji okvir, povraća stanje i nastavlja izvođenje. Lokalne

varijable metode se čuvaju u polju koji je dio okvira, te se dohvaćaju indeksom polja. Svaki okvir sadrži stog za operande instrukcija. Veličina tog stoga se izračunava pri prevođenju te na početku ne sadrži nikakave podatke. Tako na primjer instrukcija *iadd* očekuje dva cijela broja na vrhu stoga, koja potom zbraja te rezultat postavlja na vrh stoga. Stog operanada se isto tako koristi za prosljeđivanje argumenata metodama i primanje njihovog rezultata. Okvir sadrži i referencu na lokaciju tablice konstanti u izvršnoj okolini. U tablici konstanti nalaze se imena metode preko kojih se njihov poziv referencira. Te simboličke reference se dinamičkim povezivanjem razrješavaju tako da se razred pripadajuće metode učita u memoriju i simbolička referenca se prevodi u memorijsku lokaciju preko koje se metoda referencira u izvršnoj okolini.

3. Neuronske mreže

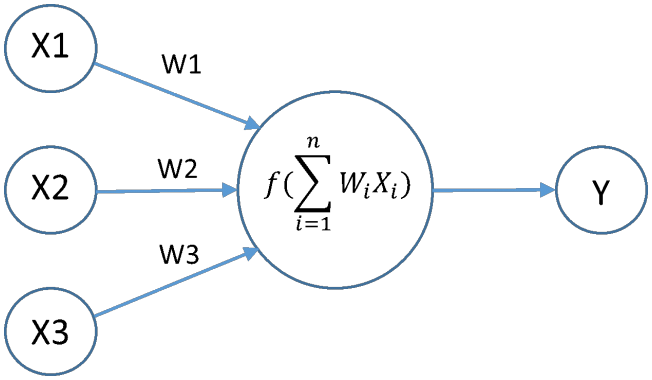
Umjetne neuronske mreže (*engl. artificial neural networks*) su računalni sustavi i algoritmi koji su inspirirani mehanizmima bioloških neuronskih mreža u životinjskim mozgovima. Bazira se na "učenju" obavljanja zadataka iz podataka, umjesto da se problem rješava programirajući znana pravila specifična za određenu domenu. Slično kao u mozgu, umjetne neuronske mreže se sastoje od skupa jednostavnih elemenata, neurona, koji su međusobno povezani velikim brojem veza i imaju mogućnost paralelnog rada.

Razvoj umjetnih neuronskih mreža počeo je s Warrena McCullochom i Waltera Pittsom koji su 1943. stvorili prvi računalni model umjetne neuronske mreže koji se zvao logika praga (*engl. threshold logic*) [16]. Model se temelji na uočenom ponašanju neurona koje zahtjeva prekoračenje određenog praga podražaja prema neuronu kako bi neuron generirao impuls. Inspiriran tim radom, američki psiholog Frank Rosenblatt konstruirao je 1957. prvi umjetni neuron, nazvan perceptron. Perceptron je zamišljen kao binarni klasifikator. Perceptron prima niz ulaza, to jest signala, koji se množe sa težinskim faktorima koji definiraju unutarnje stanje perceptrona, zatim se zbrajaju kako bi producirali jedan izlaz. Kako bi perceptron radio kao binarni klasifikator, na izlazu je bio definiran prag koji bi odlučivao kako će se izlazna vrijednost klasificirati. Perceptron se formalno može izraziti kao:

$$f(x) = \begin{cases} 1, & \text{ako } \mathbf{w} \cdot \mathbf{x} + \mathrm{b} > 0 \\ 0, & \text{inače} \end{cases}$$

gdje je **w** vektor težina, a **x** ulazni vektor. Na slici 3.1 možemo vidjeti osnovnu strukturu umjetnog neurona. U slučaju perceptrona funkcija **f** na slici 3.1 je *step* funkcija. Učenje ovakvog modela se svodi na podešavanje težina **w** kako bi se dobio odgovarajući izlaz za tražene ulaze.

Unatoč početnom uspjehu zbog očekivanih mogućnosti ovakvog pristupa, područje je dugo vremena stagniralo. U velikoj mjeri je za to bila zaslužna niska moć procesiranja, ali i problem implementacije funkcije *isključivo-ili* kao što su naveli [19]



Slika 3.1: Umjetni neuron

u svom radu. Problem *isključivo-ili* je nelinearni problem, s obzirom da se točke u n-dimenzionalnom prostoru ne mogu razdvojiti linearnom funkcijom, to jest ravninom. Perceptron je linearni klasifikator, a njegove težine su zapravo koeficijenti n-dimenzionalne ravnine, to jest *hiperravnine*, stoga perceptron sam po sebi nije mogao modelirati funkciju *isključivo-ili*. Kako bi se riješio nelinearni *isključivo-ili* problem, upotrebljeno je više neurona koji su tvorili višeslojni perceptron, o čemu će više biti rečeno u poglavlju 3.3.

Neuronske mreže su opet pobudile pažnju kada je algoritmom propagacije pogreške unatrag riješen problem učenja višeslojne mreže [29]. Do tada su se poznavali optimizacijski postupci poput, simuliranog žarenja (*engl. Simulated annealing*) ili genetskog algoritma, ali tek je algoritam propagacije unatrag je omogućio efikasno učenje, to jest optimizaciju parametara višeslojnih mreža.

Postepeno jednostavniji linearni modeli poput SVM-a (*Support Vector Machine*), dobivaju na popularnosti. Neuronske mreže su još uvijek imale problem performansi i nestajućeg gradijenta. Tek zadnje desetljeće, iskorištavanjem procesorske moći grafičkih kartica, te prilagođavajući modele paralelnom izvođenju i nestajućem gradijentu, neuronske mreže se vraćaju kao jako popularna tehnika u strojnom učenju.

3.1. Proces učenja

Modeli strojnog učenja mogu se podijeliti na više načina, jedan od načina je podjela po načinu učenja koji primjenjuju kao što su nadzirano i nenadzirano učenje. U ovom radu problem ćemo definirati kao klasifikacijski problem koji se rješava postupkom nadziranog učenja. U tu svrhu ćemo definirati osnovne pojmove i postupke za rješavanje takvog problema.

Nadzirano učenje je tip zadatka koji povezuje ulaze s jednom od definiranih klasa.

Ulazni skup podataka na kojem se model uči zove se skup za treniranje. Algoritam nadziranog učenja analizira podatke iz skupa za treniranje, te modelira rezultantnu funkciju za zaključivanje koja se koristi za označavanje novih neviđenih podataka. Tu se javlja pojam generalizacije, to jest koliko dobro rezultantni model zaključuje o neviđenim podacima.

Postupak rješavanja problema nadziranim učenjem može se svesti na sljedeći niz koraka:

1. Kreiranje i definiranje baze primjera koja će se koristiti kao primjerci za treniranje.

2. Odvajanje baze primjera na skupove za treniranje, validaciju i testiranje, što znači definiranje podataka u parove ulaznih i izlaznih vrijednosti, to jest primjeraka i njihovih klasifikacija. Bitno je da uzorci u skupu za treniranje budu reprezentativni stvarnim uzorcima kako bi model mogao naučiti dobro generalizirati.

3. Definiranje značajki ulaznih podataka. Način reprezentacije je jako bitan za dobro funkcioniranje modela, ako podaci nisu dovoljno dobro opisani model će imati problema sa raspoznavanjem uzoraka i učenjem obrazaca.

4. Definiranje modela, algoritma učenja i funkcije gubitka. Funkcija gubitka je ključna za optimizaciju modela, ona nam daje skalarnu vrijednost koja označava u kolikom su nesrazmjeru klasifikacije modela i točne klase. Cilj optimizacije je da minimizira vrijednost funkcije gubitka.

5. Pokretanje algoritma učenja na skupu za treniranje. Model je definiran strukturom neurona te parametrima koji ne utječu izravno na funkciju gubitka. Za određivanje optimalnih parametara, potrebno je optimizirati model sa različitim parametrima te koristiti poseban skupa podataka za validaciju pri izračunu mjere točnosti modela.

6. Treniranje modela na skupu za treniranje i validaciju te evaluacija rezultata na skupu za testiranje.

3.2. Algoritam propagacije pogreške unatrag

Algoritam propagacije pogreške unatrag je metoda koja se koristi pri učenju neuronskih mreža, to jest za izračunavanje vrijednosti težina koje koristi mreža. Često se

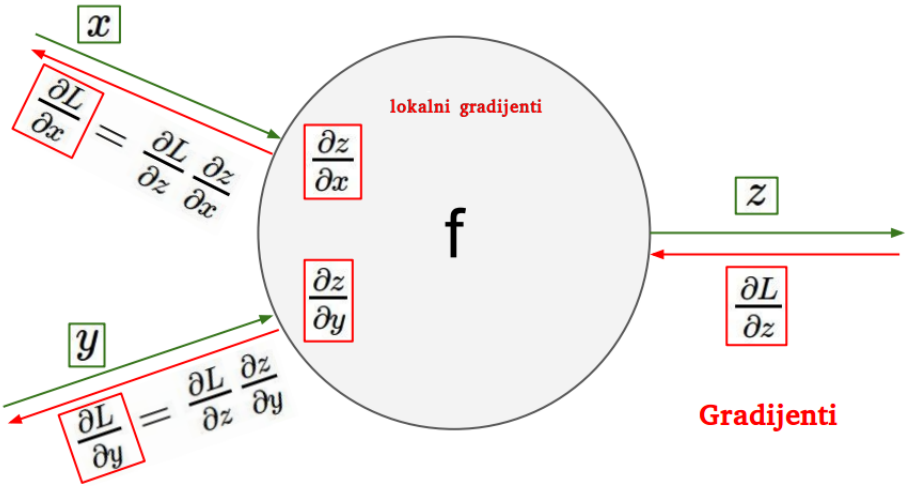
koristi za treniranje dubokih neuronskih mreža, pošto ostali algoritmi zbog povećanog broja parametara imaju uvelike lošije performanse. Ova tehnika se najčešće, ali ne nužno, koristi sa metodom gradijentnog spusta kako bi se optimizirali parametri.

Za korištenje algoritma propagacije pogreške unatrag potrebne su dvije pretpostavke. Prva pretpostavka zahtjeva da se funkcija gubitka može napisati kao srednja vrijednost funkcija gubitka pojedinačnog ulaznog primjerka. To jest ako je E_x gubitak za jedan primjerak iz skupa za treniranje, funkcija gubitka E onda iznosi: $E = \frac{1}{n}\sum_x E_x$ Druga pretpostavka zahtjeva da se funkcija gubitka može zapisati kao funkcija ovisna o izlazima neuronske mreže, to jest ako je y izlaz neuronske mreže, a $y^{'}$ željena, to jest točna vrijednost izlaza neuronske mreže, funkcija gubitka općenito treba biti $E = E(y, y^{'})$.

Algoritam propagaciju pogreške unatrag se provodi u dva koraka:

- Propagacija
    1. Izračun svih neurona u modelu i dobivanje konačnih izlaza
    2. Izračun greške, to jest funkcije gubitka na temelju dobivenih izlaza
    3. Propagacija greške unatrag kroz mrežu kako bi se za svaki neuron izračunao gradijent greške o izlazima neurona.
- Osvježavanje težina
    1. Izračun gradijenta greške ovisno o težinama, tako da se množe ulazi neurona i gradijent greške ovisano o izlazu neurona
    2. Pomak težina u smjeru negativnog gradijenta za određeni postotak koji se obično naziva stopa učenja. Stopa učenja utječe na brzinu i točnost učenja, što je stopa učenja veća model se uči brže. Prevelika stopa uzrokuje da je pomak težina prevelik te se težine odmiču od točke miniuma.

Na slici 3.2 može se vidjeti tok gradijenta greške *L* kroz jedan neuron. Kada je više neurona povezano kroz slojeve, onda su izlazi jednog neurona ulazi drugog. Analogno izlazni gradijent jednog neurona je zapravo ulazni gradijent drugog neurona. Na slici 3.2 vidimo unutar neurona u crvenim kvadratima lokalne gradijent koji se po pravilu ulančavanja množe s ulaznim gradijentom.



Slika 3.2: Tok gradijenta greške kroz jedan neuron [2]

3.3. Višeslojne neuronske mreže

Višeslojne neuronske mreže sastoje se od niza slojeva, gdje u svakom sloju može biti proizvoljan broj umjetnih neurona. Tako organizirani neuroni primaju ulaze iz izlaza prošlog sloja. Na slici 3.3 prikazan je primjer višeslojne mreže. Ulazni sloj je početni vektor podataka, zatim slijedi skriveni sloj neurona, te izlazni sloj. Skriveni sloj se sastoji od neurona s nelinearnom aktivacijskom funkcijom **f** s čime se povećava složenost modela i omogućava rješavanje nelinearnih problema u n-dimenzionalnom prostoru. Ako je aktivacijska funkcija u skrivenom sloju linearna, model nije u mogućnosti riješiti nelinearni problem. To slijedi iz činjenice da je linearna kombinacija linearnih funkcija još uvijek linearna funkcija.

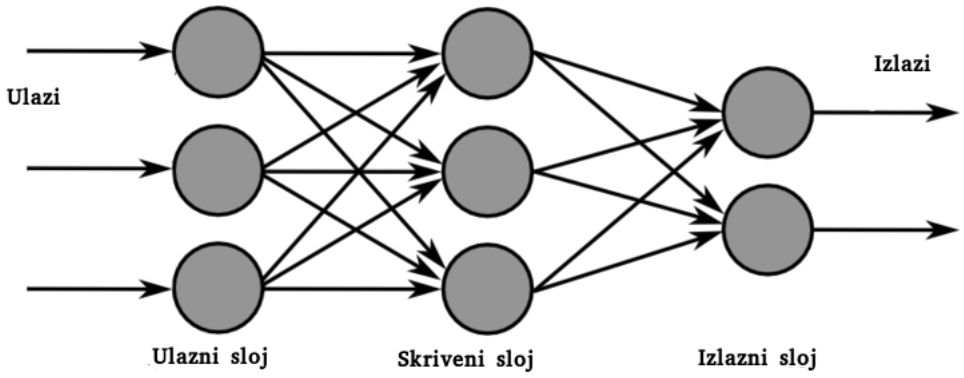
Formalno, model se može predstaviti nizom matrica $\mathbf{W}$ koja označava težine neurona u jednom sloju. Jedan neuron s aktivacijskom funkcijom f i izlazom n može se izraziti na sljedeći način: $n = f(\mathbf{w}^\top \cdot \mathbf{x} + b)$ gdje su $\mathbf{w}$ i $\mathbf{x}$ težine neurona, odnosno ulazni vektor. Slično, cijeli sloj neurona može se izraziti kao vektorska funkcija: $\mathbf{h} = f(\mathbf{W} \cdot \mathbf{x} + \mathbf{b})$. Matrica $\mathbf{W}$ u svakom svom retku sadrži jedan vektor težina za svaki neurona u sloju, pa su njezine dimenzije $m \times n$ gdje je m broj neurona u sloju, a n veličina ulaznog vektora. Ulazni vektor ovisi o veličini prijašnjeg sloja to jest veličina mu odgovara broju neurona iz prijašnjeg sloja. Stoga neuronsku mrežu na slici 3.3 možemo izraziti na sljedeći način:

$$\mathbf{o} = \mathbf{f}_2(\mathbf{W}_2 \cdot \mathbf{f}_1(\mathbf{W}_1 \cdot \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2)$$

gdje bi $\mathbf{x}$ bio ulazni vektor veličine 3, $\mathbf{W}_1$ matrica težina dimenzija 3×3,$\mathbf{W}_2$ matrica težine dimenzija 2×3, te $\mathbf{b}_1$ i $\mathbf{b}_2$ vektori odmaka veličina 3 odnosno 2.

Na slici 3.3 vidimo da su neuroni u jednom sloju potpuno povezani sa neuronima u prošlom sloju. To ne mora nužno biti tako, određeni neuron može biti povezan samo s dijelom neurona iz prošlog sloja. Na takav način se smanjuje broj parametara, pojednostavljuje učenje i omogućava bržu specijalizaciju neurona da reagiraju na samo određene ulaze.

Potpuno povezana višeslojna neuronska mreža je jako snažan model i može aproksimirati bilo koju funkciju [10], ali ovisno o problemu postoje efikasnije metode s manjim brojem parametara i bržim učenjem. Osim toga, često se koristi u kombinaciji s drugim tipovima mreže.



Slika 3.3: Višeslojna mreža

3.4. Aktivacijske funkcije

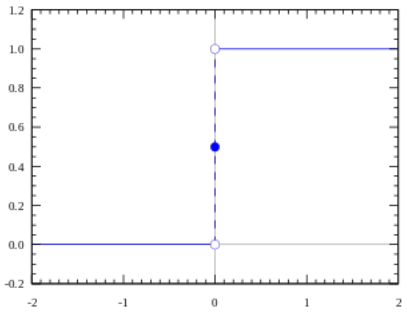
Aktivacijske funkcije su bitan segment neuronskih mreži, one odlučuju koja je domena izlaza neurona, obično čine izlaz nelinearnim, te njezin oblik utječe kako će se ponašati povratni gradijent kroz mrežu.

3.4.1. Step funkcija

Najobičnija aktivacijska funkcija je step funkcija. Ona se koristi u perceptronu, te pomoću težine odmaka određuje prag kada će se neuron aktivirati. Oblik step funkcije se može vidjeti na slici 3.4.

$$f(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

Step funkcija najčešće nije pogodna za korištenje zato što je njezina derivacija uvijek 0, osim u ishodištu gdje nije definirana. To znači da gradijentne metode optimizacije, kao algoritam propagacije pogreške unatrag, nisu iskoristive jer bi gradijent iznosio nula [24].



Slika 3.4: Step funkcija [28]

3.4.2. Logistička funkcija

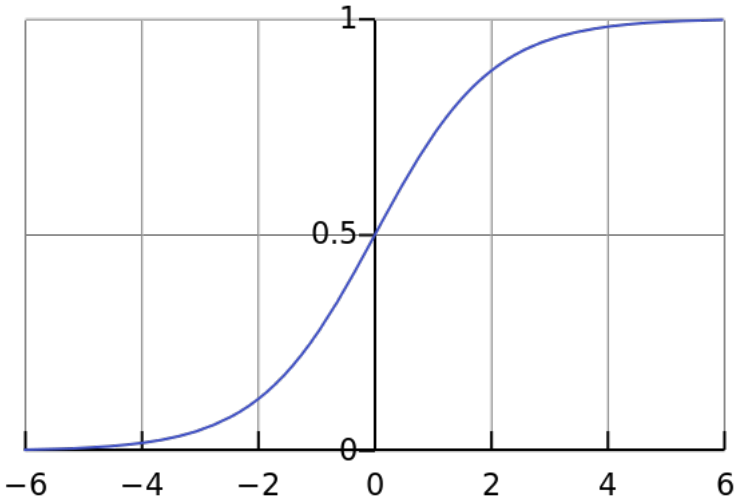
Umjesto step funkcije najčešće se koristi logistička funkcija (sigmoida), slika 3.5. To je nelinearna funkcija koja je derivabilna na cijelom svom području. Ograničuje izlaz na područje između 0 i 1 te, za razliku od step funkcije, izlaz nije binaran.

$$f(x) = \frac{1}{1 + \mathrm{e}^{-x}} = \sigma(x)$$

Derivacija logističke funkcije je sljedeća:

$$f^{'}(x) = f(x) \cdot (1 - f(x))$$

Zbog ovakve derivacije i svog oblika, logistička funkcija ima najveći gradijent u području od -2 do 2. Mala promjena ulaza u tom području može značajno promijeniti izlaz funkcije. Kako se vrijednost ulaza aktivacije odmiče od tog područja, nagib sigmoide postaje sve manji, pa stoga i gradijent sve više teži nuli. To ima za posljedicu usporavanje učenja ili potpuni gubitak gradijenta, pa ponekad nije pogodna za jako duboke mreže.



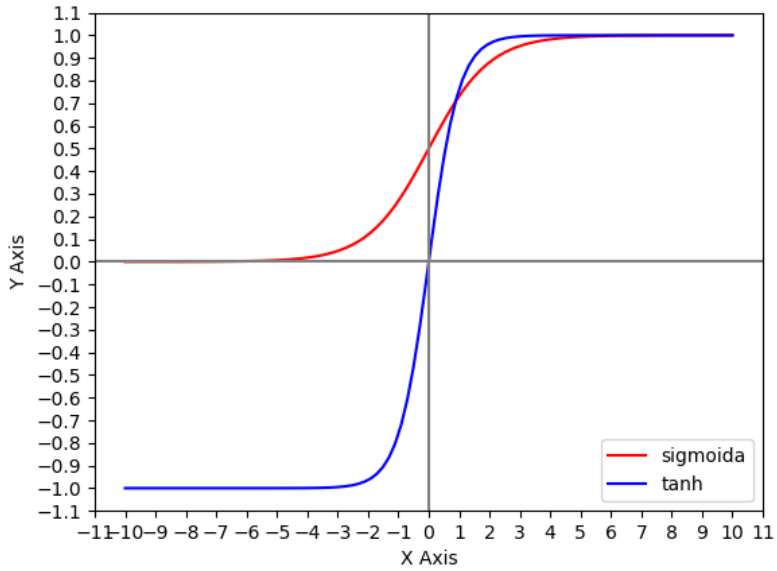
Slika 3.5: Logistička funkcija [28]

3.4.3. Tangens hiperbolni

Tangens hiperbolni je skalirana logistička funkcija, slika 3.6.

$$f(x) = tanh(x) = \frac{(\mathrm{e}^{x} - \mathrm{e}^{-x})}{(\mathrm{e}^{x} + \mathrm{e}^{-x})}$$
$$f(x) = 2 \cdot sigmoida(2 * x) - 1$$

Tangens hiperbolni ima slična svojstva kao logistička funkcija, glavna razlika je što ima oštriji nagib i kodomenu. To znači da je gradijent jači te da oko male promjene ishodišta u domeni uzrokuje još veće promjene u kodomeni. Isto kao i logistička funkcija ima problem nestajućeg gradijenta.



Slika 3.6: Tanges hiperbolni

3.4.4. ReLu funkcija

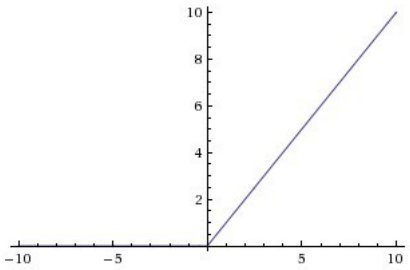
Jedan od načina rješavanje problema nestajućeg gradijenta je ReLu (*engl. Rectified linear unit*) aktivacijska funkcija, slika 3.7. To je nelinearna funkcija koja je prekinuta u ishodištu, te za ulaze manje od nula nema izlaz. Zbog toga što se aktivira samo na polovici domene efikasna je za izračun, uvelike poboljšava performanse mreže, bolje nego druge aktivacijske funkcije.

$$f(x) = \begin{cases} x & x \geq 0 \\ 0, & x < 0 \end{cases}$$

$$f(x) = max(0, x)$$

$$f^{'}(x) = \begin{cases} 1 & x \geq 0 \\ 0, & x < 0 \end{cases}$$

Zbog toga što je funkcija linearna na pozitivnom dijelu osi, gradijent ne iščezava na tom dijelu. To svojstvo uklanja problem nestajućeg gradijenta unatoč tome što je gradijent na drugoj polovici domene uvijek nula. To je tako zato što neuroni u mreži imaju rijetke aktivacije to jest uče kada izlaz neurona treba biti veći od nule, pa ukoliko postoji put za aktivaciju svakog neurona pri optimizaciji mreža će se uspješno naučiti [11]. Svejedno se može dogoditi da se za sve ulaze neuron ne aktivira. Tada se kaže da je



Slika 3.7: ReLu funkcija [28]

neuron mrtav pošto mu je izlaz nula, te je gradijent uvijek nula, pa se njegove težine ne mogu promijeniti da se odgovarajući uključi. U tom slučaju mogu se koristiti varijante ReLu aktivacijske funkcije koje imaju blaži nagib u negativnom djelu domene kao *Leaky ReLU* za koji vrijedi $f(x) = 0.01x,$ za $x < 0,$ inače x. S time se gubi pogodnost rijetkih aktivacija, pa je ova metoda računski zahtjevnija. Problem sa ReLu funkcijom je što nije ograničena odozgo, kroz nekoliko koraka u dubokim mrežama vrijednosti aktivacija eksponencijalno rastu pa dolazi do kombinatorne eksplozije. Zbog toga je bitno pripaziti pri dizajniranju mreže kako bi se dodali normalizacijski slojevi koji će spriječiti rast aktivacija unedogled.

3.5. Povratne neuronske mreže

Povratne neuronske mreže (engl. Recurrent Neural Networks, RNN) su vrsta neuronskih mreža koje spadaju u skupinu arhitektura dubokog učenja. Njihova specifičnost je u tome što su građene od posebnih neurona koje se zovu ćelije (engl. cell). Te ćelije mogu čitati niz ulaza jedan za drugim, istodobno pamteći stanje o prijašnjim ulazima. Zbog ovakve mogućnosti često se koriste za procesiranje prirodnih jezika, prepoznavanje govora i rukopisa.

Ćelija povratne neuronske mreže može se prikazati kao na slici 3.8. Prikazano je kako se ćelija razmotava kroz ulazni niz, prenoseći skriveno stanje u sljedeću iteraciju. Povratnu ćeliju možemo interpretirati kao jedan sloj neuronske mreže, a vrijeme t označava redoslijed kojim se podaci šalju na ulaz. Osnovna ćelija povratne neuronske mreže može se prikazati sljedećom formulom:

$$\mathbf{h}^{(t)} = f(\mathbf{W} \cdot \mathbf{h}^{(t-1)} + \mathbf{U} \cdot \mathbf{x}^{(t)} + \mathbf{b})$$

Vektor $\mathbf{h}^{(t)}$ označava skriveno stanje u trenutku t, težinska matrica $\mathbf{W}$ odlučuje koliko prošlo stanje utječe na trenutačno, a matrica $\mathbf{U}$ odlučuje kako će trenutačni ulaz utjecati

na trenutačno stanje. Produkti se zajedno zbrajaju s odmakom i ulaze u aktivacijsku funkciju za koju se obično koristi tangens hiperbolni.

Obično se dimenzije vektora h ne poklapaju sa potrebnim dimenzijama izlaza, pa se na izlazu dodaje jedna linearna transformacija kako bi se generirao izlazni vektor o.

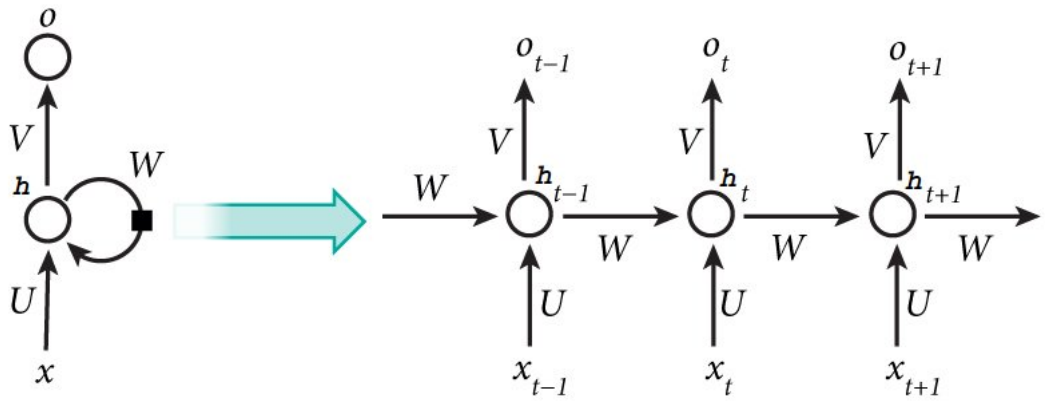
$$\mathbf{o}^{(t)} = \mathbf{V} \cdot \mathbf{h}^{(t)} + \mathbf{c}$$

Kako bi se izlazi koristili za klasifikaciju s vjerojatnosnom interpretacijom, koristi se funkcija $softmax$ to jest normalizirana eksponencijalna funkcija. To je generalizirana logistička funkcija za C klasa koja ima sljedeću formulu:

$$y_i = P(y = j|\mathbf{x}) = \frac{\mathrm{e}^{o_i}}{\sum_{j=1}^{C} \mathrm{e}^{o_j}}$$

$Softmax$ funkcija ulaze podiže na eksponent, te normalizira sa sumom svi ostalih vrijednosti u vektoru. To znači da će suma svih vrijednosti izlaza $softmax$ funkcije biti jedan. Proporcije vrijednosti neće biti očuvane, to jest ako su vrijednosti veće od 1, najveća vrijednost će postati relativno veća nad ostalim vrijednostima, no suprotno će se dogoditi ako su vrijednosti unutar pojasa od 0 do 1. Općenito za vremenski trenutak t možemo zapisati na sljedeći način:

$$\hat{\mathbf{y}}^{(t)} = softmax(\mathbf{o}^{(t)})$$



Slika 3.8: Razmotavanje ćelije povratne mreže [6]

3.5.1. Funkcija gubitka

Funkcija gubitka L povratne neuronske mreže za klasifikaciju je unakrsna entropija:

$$L^{(t)} = -\sum_{i=1}^{C} y_i^{(t)} \cdot \log \hat{y}_i^{(t)}$$

Unakrsna entropija se interpretira kao broj koji označava mjeru poklapanja dvije vjerojatnosne distribucije p i q na istom skupu događaja. Ako se koristi logaritam po bazi 2 mjera se može gledati kao prosječan broj bitova koji je potreban za kodiranje događaja iz skupa koji se ponašaju prema distribuciji p, a način kodiranja je optimiziran kao da se događaji ponašaju po distribuciji q. Formalno se zapisuje, ako je E očekivanje:

$$H(p, q) = E_p[-\log q]$$

Unakrsna entropija se može gledati kao negativna log izglednost zato što naš izlaz ima vjerojatnosnu interpretaciju. Pretpostavka je da je naš skup za treniranje reprezentativan stvarnom stanju, to jest da je distribucija našeg skupa najvjerojatnija. Stoga učimo model da, uz dane ulaze, umnožak vjerojatnosti točnih klasifikacija bude maksimalan[5].

3.5.2. Optimizacija

Optimizacija povratne neuronske mreže se obavlja algoritmom propagacije pogreške unatrag. S obzirom da povratne mreže prolaze kroz vremenske korake s istim težinama, valja naglasiti tok gradijenta kroz mrežu. Stanja i ulazi povratne neuronske mreže utječu na svaki idući izlaz pa se gradijent skrivenog stanja $\mathbf{h}^{(t)}$ računa na sljedeći način:

$$\frac{\partial L}{\partial \mathbf{h}^{(t)}} = \frac{\partial L^{(t)}}{\partial \mathbf{h}^{(t)}} + \frac{\partial L^{(t+)}}{\partial \mathbf{h}^{(t)}}$$

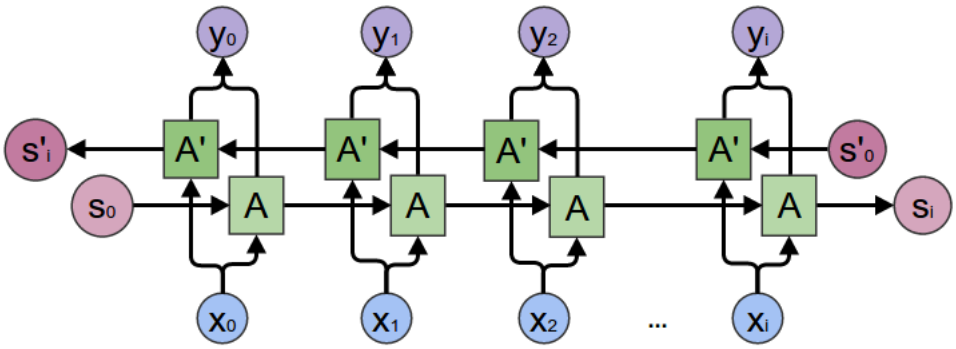
Formula označava da gradijent skrivenog stanja u koraku t ovisi gubitku u tom koraku, ali i svim koracima poslije tog koraka. To znači da se gradijent težina ćelije povratne neuronske mreže akumulira u svakom koraku.

$$\frac{\partial L}{\partial \mathbf{W}} = \sum_{t=1}^{T} \frac{\partial L}{\partial \mathbf{h}^{(t)}} \cdot \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{W}}$$

Analogno se računa gradijent ostalih težina u ćeliji. Zbog velikog akumuliranja gradijenta, moguć je problem eksplodirajućeg ili nestajućeg gradijenta, zbog čega se koristi tangens hiperbolni aktivacijska funkcija koja ima više linearan oblik oko 0 nego logistička funkcija. Još jedna posljedica je da se obrasci na dalekim vremenskim koracima teško uče, jer gradijent udaljenog stanja ima puno manji utjecaj na učenje skrivenog stanja u trenutačnom koraku. Uz to, obična povratna mreža prolazi samo u jednom smjeru te ponekad je potrebna informacija iz oba smjera za točnu odluku. Kako bi se riješili navedeni problemi koriste se dvosmjerne povratne mreže i LSTM ćelije.

3.6. Dvosmjerne povratne mreže

Kako bi se za određeni vektor iz niza ulaza pronašle ovisnosti o ulaznim vektorima prije i poslije trenutačnog, koriste se dvosmjerne povratne mreže. Na slici 3.9 vidimo primjer dvosmjerne povratne mreže. Za svaki smjer početno stanje se posebno inicijalizira, te se dvosmjerna mreža ponaša kao dvije jednosmjerne mreže koje prolaze u suprotnim smjerovima po ulaznim vektorima. Jedina razlika je što se nad izlazima mreže u svakom trenutku provodi operacija spajanja. Za spajanje se može koristi uprosječivanje ili neka druga proizvoljna funkcija, ali najčešće se koristi konkatenacija, što rezultira u povećavanju dimenzije izlaznog vektora.



Slika 3.9: Dvosmjerna povratna mreža [20]

Najveći nedostatak ovakvog pristupa je pad performansi za višeslojne povratne mreže. Naime, kod običnih višeslojnih povratnih mreža moguće je računati više slojeva paralelno, zato jer vremenski trenutak u višem sloju ovisi o prijašnjem sloju samo kroz taj isti vremenski trenutak. To nije slučaj kod dvosmjernih mreža, s obzirom da su za izračun izlaza jednog trenutka potrebni prijašnji trenutci i nadolazeći iz suprotnog smjera.

3.7. LSTM ćelija

LSTM (*engl. Long Short-Term memory*) ćelija je posebna vrsta ćelije čija namjena je da pamti obrazac ulaznih vektora, koji se mogu pojavljat uz prekide kroz duži niz. Dizajnirana je s namjenom da bolje podnosi problem nestajućeg ili eksplodirajućeg gradijenta i bolje regulira svoje unutarnje stanje.

Na slici 3.10 može se vidjeti arhitektura LSTM ćelije uz danu notaciju na slici 3.11.

U ćeliji se obavljaju sljedeće operacije:

$$\mathbf{f}_t = \sigma(\mathbf{W}_f \cdot [\mathbf{h}_{t-1}, \mathbf{x_t}] + \mathbf{b}_f) \tag{3.1}$$

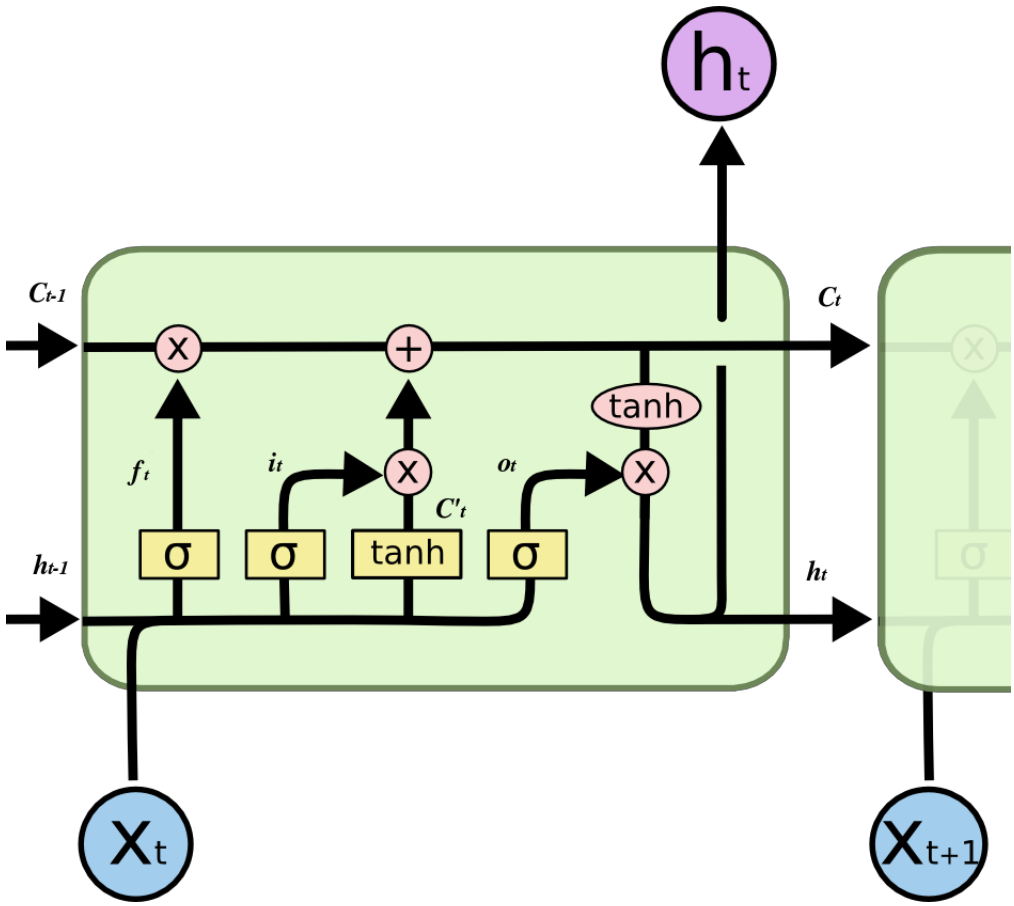
$$\mathbf{i}_t = \sigma(\mathbf{W}_i \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i) \tag{3.2}$$

$$\mathbf{C'}_t = tanh(\mathbf{W}_C \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_C) \tag{3.3}$$

$$\mathbf{C}_t = \mathbf{f}_t \odot \mathbf{C}_{t-1} + \mathbf{i}_t \odot \mathbf{C'}_t \tag{3.4}$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_o \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o) \tag{3.5}$$

$$\mathbf{h}_t = \mathbf{o}_t \odot tanh(\mathbf{C}_t) \tag{3.6}$$



Slika 3.10: LSTM ćelija [21]

Slika 3.11: Notacija LSTM grafa [21]

Za razliku od obične ćelije, LSTM ćelija prenosi u sljedeći korak dva vektora stanja: $\mathbf{h}_t$ koji služi za računanje trenutačnog izlaza ćelije i $\mathbf{C}_t$ koji služi kao memorija ćelije. Formula 3.1 računa funkciju $\mathbf{f}_t$ koja odlučuje kako će se izbrisati memorije iz prijašnjeg stanja $\mathbf{C}_t$. Funkcija ovisi o prošlom stanju $\mathbf{h}_t$ i ulaznom vektoru $\mathbf{x}_t$, te koristi logističku funkciju aktivacije kako bi vrijednosti bile u pojasu između 0 i 1, kao postotak informacija koje želimo sačuvati. Formula 3.2 se koristi po istom principu, ali služi za odluku koliko će se novo stanje $\mathbf{C'}_z$ propustiti u memoriju. Novo stanje se računa formulom 3.3, zatim se u formuli 3.4 izračunava nova memorija pomoću funkcija $\mathbf{f}_t$ i $\mathbf{i}_t$, množeći vektore stare memorije i novog stanja po svakom elementu (Hadamardov produkt). Zatim se po istom principu kao funkcije 3.1 i 3.2 računa funkcija $\mathbf{o}_t$ u formuli 3.5 za izračun izlaza ćelije. Konačno, u formuli 3.6 se računa izlaz ćelije, to jest skrivena reprezentacija koja se ujedno prenosi u sljedeći vremenski trenutak.

U ovakvoj izvedbi vektor memorije $\mathbf{C}_{t-1}$ ne utječe na funkcije propusnice $\mathbf{f}_t$, $\mathbf{i}_t$ i $\mathbf{o}_t$ zbog čega je gradijent ne eksplodira za taj vektor. Unatoč tome, ponekad je korisno uvesti tu ovisnost jer pomaže da se propusnice bolje izračunaju.

3.8. Sekvencijski modeli

Sekvencijski modeli (engl. sequence to sequence models [27]) su tip modela koji iz ulaznog niza generiraju novi izlazni niz. Ovakav tip modela je pogodan za prevođenje prirodnih jezika. U sklopu neuronskih mreža, model se oblikuje pomoću dvije povratne mreže. Prva povratna mreža predstavlja enkoder koji čita ulazni niz i generira skrivenu reprezentaciju cijelog niza. Druga povratna mreža predstavlja dekoder koji iz skrivene reprezentacije nastoji generirati izlazni niz.

Teoretski, problem predstavlja kako procijeniti uvjetnu vjerojatnost izlaznog niza. Formalno $(x_1, ..., x_T)$ predstavlja ulazni niz, a $(y_1, ..., y_{T'})$ izlazni niz. Skrivenu reprezentaciju možemo predstaviti vektorom c. Traži se najbolja procjena uvjetne vjerojatnosti:

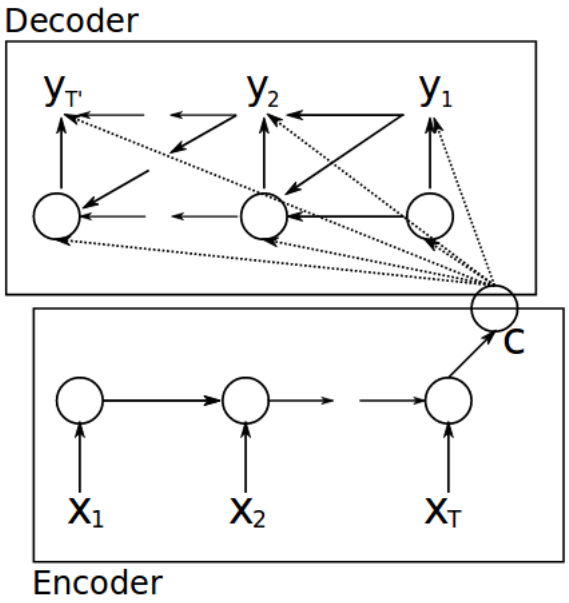
$$p(y_1, ..., y_{T'}|x_1, ..., x_T) = \prod_{t=1}^{T'} p(y_t|c, y_1, ..., y_{t-1}) \tag{3.7}$$

$$p(y_t|c, y_1, ..., y_{t-1}) = g(y_{t-1}, s_t, c) \tag{3.8}$$

U formuli 3.8 funkcija g predstavlja nelinearnu funkciju izlaza povratne mreže dekodera koji predstavlja kolike su vjerojatnosti da jedan od mogućih znakova nastavlja niz. Kako bi se vjerojatnost mogla izračunati na nizovima proizvoljne duljine, na kraju

ulaznog i izlaznog niza se dodaje oznaka kraja niza. Model se nadalje optimizira maksimizirajući izglednost izlaznih nizova.

Model se može opisati slikom 3.12. Enkoder prolazi ulaznim nizom i generira skrivenu reprezentaciju koja zatim ulazi u dekoder koji prima prethodni član dekodiranog niza, skriveno stanje dekodera i vrijednost reprezentacije kako bi generirao novi znak.



Slika 3.12: Sekvencijski model [8]

Za evaluaciju rezultata kod prirodnih jezika se obično koriste mjere kao *BLEU* (*engl. bilingual evaluation understudy* [22]) koja procjenjuje kvalitetu rezultata na razini cijelog korpusa kada postoje nekoliko različitih referentnih prijevoda. To u ovom radu nije slučaj pošto postoji samo jedan odgovarajući izlazni niz, pa su odgovarajuće mjere preciznost i odziv koje računaju koliko je međusobno poklapanje između točnog i generiranog niza.

3.8.1. Mehanizam pažnje

Mehanizam pažnje (*engl. Attention mechanism* [15] [3]) je tehnika koja se koristi u sekvencijskim modelima kako bi se dekoderu naznačilo na koje dijelove ulaznog niza treba obratiti pažnju pri generiranju sljedećeg znaka niza. Mehanizam se može prikazati slikom 3.13. Na slici su s h označena skrivena stanja enkodera, a sa s skrivena stanja dekodera. Kako bi model obratio pažnju na specifičan dio ulaznog niza, umjesto

jednog vektora skrivene reprezentacije, generira se novi vektor skrivene reprezentacije za svaki znak koji se dekodira. Model je opisan sljedećim formulama:

$$p(y_i|y_1, ..., y_{i-1}, \mathbf{x}) = g(y_{i-1}, s_i, c_i) \tag{3.9}$$

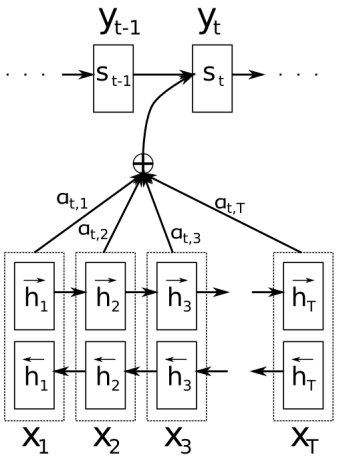
$$s_i = f(s_{i-1}, y_{i-1}, c_i) \tag{3.10}$$

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j \tag{3.11}$$

$$\alpha_{ij} = \frac{exp(e_{ij})}{\sum_{k=1}^{T_x} exp(e_{ik})} \tag{3.12}$$

$$e_{ij} = a(s_{i-1}, h_j) \tag{3.13}$$

U formuli 3.9 vjerojatnost sljedećeg izlaza sada ovisi o cijelom ulaznom vektoru to jest o vektoru konteksta, a ne samo o jednoj skrivenoj reprezentaciji. Na isti način ovisi i sljedeće skriveno stanje dekodera u formuli 3.10. Formula 3.11 prikazuje kako se izračunava vektor konteksta c_i. Vektor h_j predstavlja konkatenirana skrivena stanja dvosmjerne povratne mreže u trenutku j, to jest za j-ti ulaz. Vektor c_i je stoga težinska suma skrivenih stanja enkodera. Težine se računaju tako da se normaliziraju vektori e_{ij} koji predstavljaju odnos između trenutačnog skrivenog stanja s_{i-1} dekodera i svih vektora skrivenih stanja enkodera h_j. Odnos se izračunava pomoću nelinearne funkcije modela a (formula 3.13) koja se uči zajedno sa modelom.



Slika 3.13: Mehanizam pažnje [3]

Mehanizam pažnje je jako koristan jer omogućava modelu da probire informaciju o skrivenoj reprezentaciji enkodera kako bi generirao sljedeći izlaz, to jest omogućava da se model fokusira na specifičan dio ulaznog niza koji je bitan za dekodiranje sljedećeg izlaza.

4. Eksperimenti

Sva programska podrška korištena za izradu ovog rada dostupna je na Web stranici `https://github.com/mkucijan/Java-Bytecode-Analysis`. Za procesiranje podataka korišten je python paket *javalang* i modul za parsiranje Java *class* datoteka u python objekte. Za implementaciju modela neuronske mreže korišten je alat *Tensorflow* [1] u Python programskom jeziku. Prednosti korištenja ovog alata je što omogućava jednostavno građenje neuronskih mreža tako da se željeni čvorovi i operacije dodaju u model, te se bira optimizacijski postupak za koji se gradijent automatski izračunava iz strukture grafa. Uz to, koristio se Google radni okvir za sekvencijske modele koji se najčešće koriste za prevođenje prirodnih jezika [7].

Za ovaj rad definirana su dva zadatka:

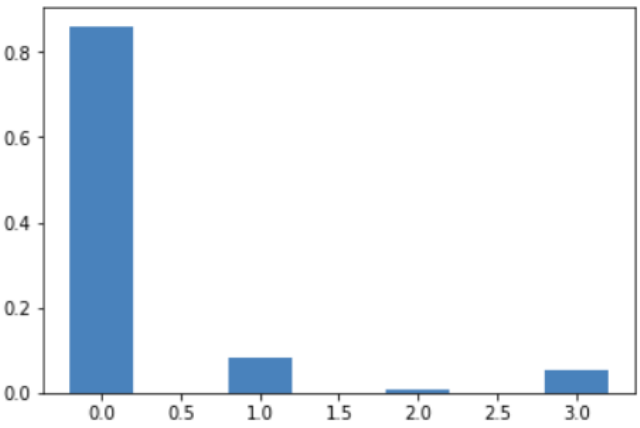
- klasifikacija Java instrukcija koje pripadaju određenom izrazu
- sekvencijski model koji prevodi instrukcije u oznake izraza izvornog koda

U nastavku je opisano koji su podaci korišteni, kako su označeni, koja je reprezentacija podataka, te kakav se model koristi za klasifikaciju.

4.1. Podatci

Za provođenje ovog eksperimenta koristile su se programske knjižice otvorenog koda *Apache Commons*. Otvoreni kod se prevodio pomoću *Javac* prevodioca s uključenim *debug* simbolima kako bi se u metodama izvršnog koda nalazio atribut *LineNumberTable*. Ta tablica se koristila kako bi se instrukcije označile kojem izrazu u izvornom kodu pripadaju, no u tablici ne postoje zapisi za svaku pojedinu liniju i izraz pa nije moguće savršeno povezati instrukcije u izvorni kod. Za pojedine instrukcija kao naredbe skoka se zaključilo da pripadaju odreženom izrazu tako da se sagledalo kako je označena instrukcija na koju se skače. Za klasifikaciju izraza i ugnježđenja koristili su se tri izraza: *IF* oznakom 1, *WHILE* oznakom 2 i *FOR* oznakom 3, ostale instrukcije su označene oznakom 0.

Distribucija tako označenih instrukcija se može vidjeti na slici 4.1. Daleko najviše instrukcija je neoznačeno to jest reprezentirano oznakom 0. Ukupna veličina baze podataka iznosi preko dvanaest tisuća metoda s preko četiristo tisuća instrukcija.



Slika 4.1: Postotak instrukcija po definiranoj klasi

Za svaki pojedini razred, parsiraju se instrukcije iz njegovog izvršnog koda, koje se nalaze u metodama tog razreda. Instrukcije se odvajaju od njezinih operanada te su zasebno označene da pripadaju određenom izrazu. Kod instrukcija čiji se operand referencira na poziciju u tablicu konstanti, umjesto pozicije dohvaća se vrijednost iz tablice, te se potom označuje.

Za drugi zadatak povezane su instrukcije s čvorovima apstraktnog sintaksnog stabla *javalang* parsera. U nastavku se može vidjeti primjerak dijela stabla u *json* formatu (kod 4.1). Svaki čvor sadrži 4 atributa:

- *name*: Ime izraza
- *pos*: Početna i završna pozicija izraza
- *children*: Djeca čvora
- *instructions*: Instrukcije i operandi generirane od tog izraza s brojem koji označava pomak za broj okteta od početka metode.

Ovakvo stablo se za svaki čvor koji sadrži instrukcije pretvara oznaku ciljanog jezika. Ako čvor označava izraz izjave generira se oznaka za početak i kraj izraza. Oznake se zapisuju u listu rekurzivnim prolazom po stablu te se tako generira ciljani jezik.

```
{
"name": "IfStatement",
"pos": [12,15],
"children": [
    {
        "name": "BlockStatement",
        "pos": [12,15],
        "children": [
            {
                "name": "StatementExpression",
                "pos": [13,14],
                "children": [],
                "instructions": [
                    ["32","iload_1"],
                    ["33","iconst_1"],
                    ["34","iadd"],
                    ["35","istore_1"]
                ]
            }
        ]
    }
],
"instructions": [
    ["27","iload_1"],
    ["28","iconst_1"],
    ["29","if_icmple"],
    ["30","7"]
]
}
```

Kod 4.1: Primjer apstaktnog stabla

4.2. Pretprocesiranje

U sklopu preprocesiranja u izvornom kodu dan je niz metoda pomoću kojih se ulazni podaci prilagođavaju željenoj reprezentaciji. Osnovna reprezentacija instrukcija i operanada je rijetki vektor duljine veličine vokabulara koji ima jednu jedinicu na odgovarajućoj poziciji za tu instrukciju ili operand. Vokabular se generira na skupu za treniranje i sadrži posebnu oznaku za nepoznate ulazne vrijednosti. Korištenje rijetke reprezentacije stvara veliku memorijsku kompleksnost, zato se često koriste modeli guste reprezentacije. Jedan od takvih je *word2vec* model, koji stvara vektor reprezentacije fiksne veličine, uspoređujući kontekst u kojem se nalazi. Osim manje dimenzionalnosti vektora, riječi koje imaju slične kontekste i značenja imaju sličnu vektorsku reprezentaciju [18]. Kod prirodnih jezika ovakav pristup postiže dobar uspjeh, no na ovom problemu izravan izračun gustog vektora unutar modela povratne mreže je imalo puno više uspjeha.

U nastavku slijedi popis metoda za pretprocesiranje ulaznih podataka:

– Izvlačenje imena metoda iz Java potpisa metoda.

```
Ljava/lang/String/substring(II)Ljava/lang/String;
-> substring
```

Ovakvo pretprocesiranje može biti korisno za smanjenje veličine vokabulara koji je potreban za označavanje instrukcija i operanada. Uz to, s time se pretpostavlja da metode s istim imenom dijele semantičku sličnost.

– Filtriranje metoda bez instrukcije skoka. Instrukcija skoka je nužna za stvaranje složenijih izraza u izvornom kodu i ugnježđenja, pa se na ovakav način jednostavno odstranjuju nebitni dijelovi izvršnog koda koji mogu otežati učenje.

– Odvajanje operanada u posebni ulazni skup. Za operande se može koristiti njihov binarni zapis iz izvršnog koda, te spojiti s reprezentacijom instrukcija u modelu.

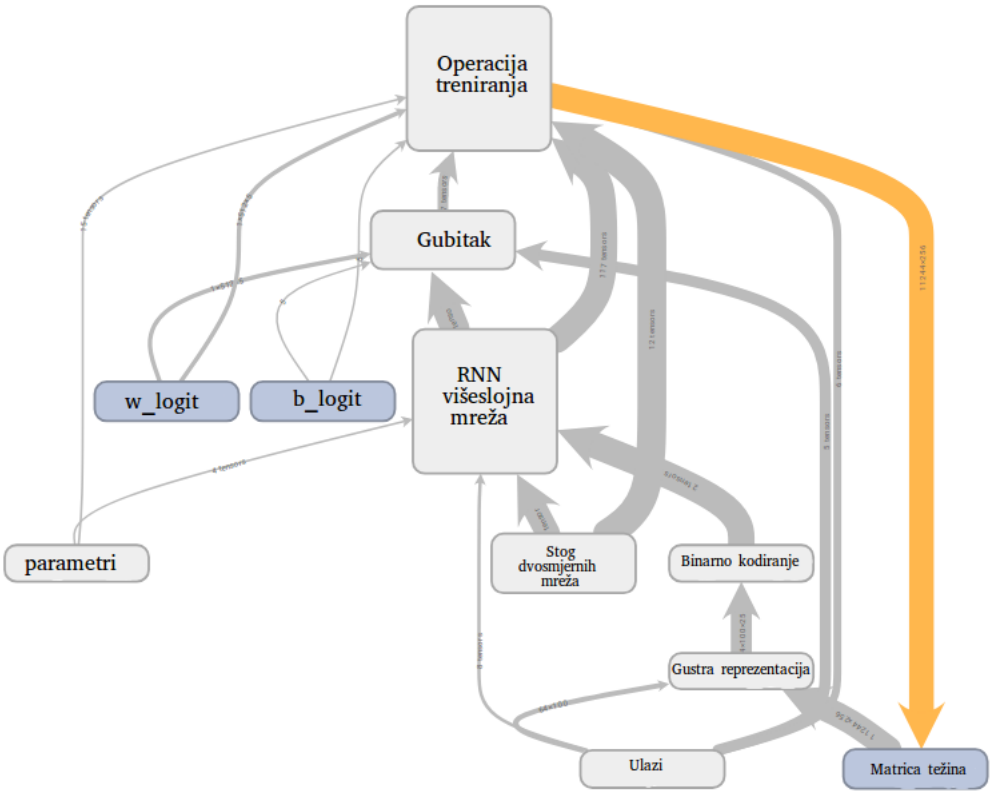
– Binarna reprezentacija operanda čija je vrijednost cijeli broj. Većina operanada koji se koriste za kontrolu toka i podataka su cjelobrojni brojevi, pa kako bi neuronska mreža imala bolji uvid u njihovu vrijednost, reprezentiraju se vektorom koji predstavlja njihov binarni zapis.

4.3. Model

4.3.1. Model povratne mreže

Konačan model za klasificiranje instrukcije ovisno o pripadnosti izrazu se sastoji od dvije ulazne reprezentacije. Prva je ulazni niz instrukcija i operanada iz jedne metode koji čini jedan vremenski niz povratne mreže. Svaka instrukcija i operand je reprezentirana brojem koji predstavlja poziciju u tablici vokabulara. Zatim se taj broj zamjenjuje vektorom guste reprezentacije koji se izvlači iz matrice težina koja se trenira zajedno sa ostatkom modela. Druga ulazna reprezentacija su cjelobrojni operanadi koji su poredani u niz jednake duljine kao i prva ulazna reprezentacija te se svaki operand nalazi na poziciji koja odgovara njegovoj pripadnoj instrukciji. Ulazni niz se definira fiksno, te se višak dopunjuje nulama, a ako nedostaje mjesta ostatak se odvaja u sljedeći ulazni niz. Reprezentacije se potom konkateniraju te se prosljeđuju u dvosmjernu povratnu mrežu s LSTM ćelijom. Povratna mreža se može sastojati od više uzastopnih slojeva gdje sljedeći sloj prima novu izlaznu reprezentaciju niza, a broj slojeva se uzima kao parametar modela. Izlazi povratne mreže zatim u svakom trenutku prolaze linearnu

transformaciju koja reducira dimenzionalnost izlaznog vektora na broj klasa. Izlazni vektor se normalizira *softmax* operacijom te se zatim računa gubitak unakrsne entropije. Predikcije se izračunavaju odabirom indeksa s najvećom vrijednošću izlaznog vektora koji se dobije iz *softmax* operacije. Model se zatim optimizira algoritmom stohastičkog gradijentog spusta koji pokazuje bolje performanse dugoročno nego neka alternativna rješenja [30]. Model se može opisat grafom na slici 4.2.



Slika 4.2: Model za klasificiranje instrukcija

4.3.2. Model prevođenja sekvence

Model prevođenja sekvence instrukcija se sastoji od enkodera, koji stvara skrivenu reprezentaciju instrukcija, i dekodera, koji generira slijed oznaka početaka i kraja izraza Java izvornog jezika. Enkoder se sastoji od višeslojne dvosmjerne povratne mreže s LSTM ćelijom. Ulazni niz instrukcija i operanada se predstavlja rijetkim vektorom pomoću vokabulara čiju gustu reprezentaciju model potom uči. Nakon toga enkoder generira skrivena stanja za svaki vremenski trenutak niza koje potom koristi dekoder. Dekoder se također sastoji od višeslojne povratne mreže. U svakom trenutku generira oznaku izraza koristeći informacije o prijašnjem trenutku te mehanizam pažnje s kojim producira skrivenu reprezentaciju iz koje dekodira informacije. Model se optimizira algoritmom stohastičkog gradijentnog spusta.

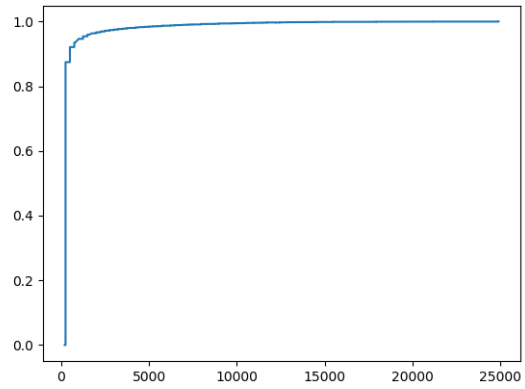
5. Rezultati

Svi eksperimenti su provođeni pomoću *Google Colaboratory* sustava koji nudi besplatno procesiranje pomoću *Tesla K80* grafičke kartice.

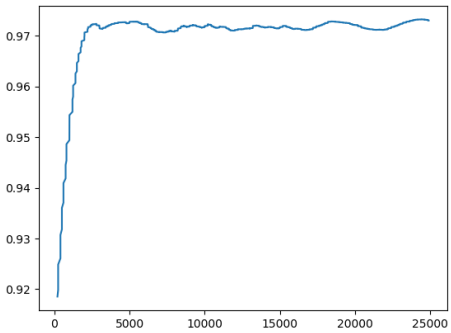
5.1. Klasifikacija izraza po instrukciji

Zadatak klasifikacije svake instrukcije se provodi dvosmjernom višeslojnom povratnom mrežom koja u svakom vremenskom trenutku klasificira ulaznu instrukciji ovisno o njezinoj pripadnosti jednoj od četiri klasa: *NULL*, *IF*, *WHILE* i *FOR*. *NULL* označava zadanu klasu u slučaju da instrukcija ne pripada nijednoj drugoj.

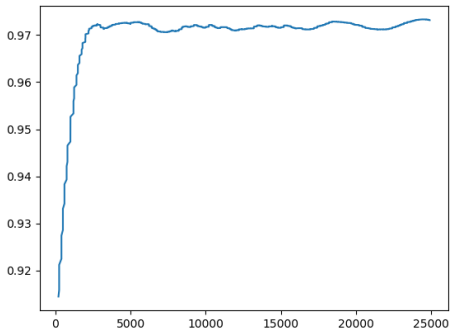
Parametri modela su odabrani pretraživanjem pomoću skupa za validaciju. Njihove vrijednosti su 300 za veličinu skrivenog sloja ćelija, zatim 3 sloja dvosmjerne mreže te 80 instrukcija za maksimalnu duljinu ulaznog niza. Na slici 5.1 može se vidjeti rast točnosti na skupu za učenje kroz iteracije. Broj metoda za učenje i evaluaciju iznosi 12103. Rezultat se može usporediti s maks klasifikatorom koji bi sve instrukcije označio klasom *NULL* te bi njegova točnost iznosila 85.39%. Na slikama 5.2 mogu se vidjeti rezultati na skupu za evaluaciju. Na tom skupu maks klasifikator ima točnost 82.87%.



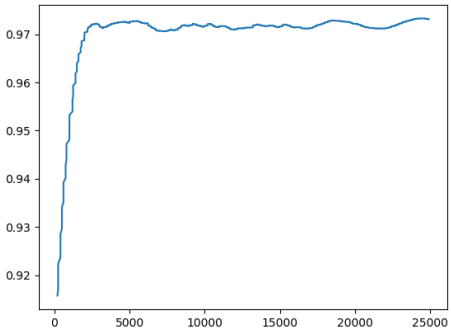
Slika 5.1: Točnost na skupu za učenje



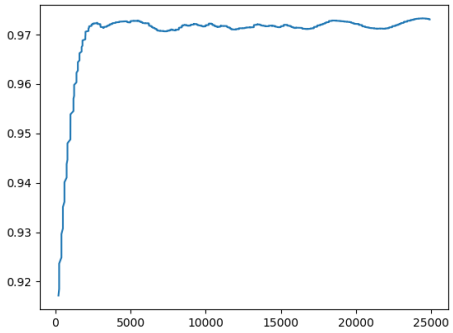
(a) Točnost



(b) f1 mjera



(c) Preciznost



(d) Odziv

Slika 5.2: Rezultati na skupu za evaluaciju

Unatoč visokoj točnosti model ima problema s prepoznavanjem više izraza u jednom slijedu što ćemo opisati pomoću primjera u kodu 5.1.

```
public class Test1 {

        private static int parametar;

        public static void sayHello() {
                System.out.println("Hello.");
        }

    public static void main(String[] args) {
        int mylocal = 1;
        System.out.println("" + mylocal);
        if(mylocal>1) {
                mylocal = mylocal + 1;
        }
        while(mylocal<10) {
                System.out.println(".." + mylocal);
                mylocal = mylocal + 1;
        }
        for(int i=0; i<2; i++) {
                sayHello();
        }
    }
}
```

Kod 5.1: Testni primjer

Za dani primjer programa u kodu 5.2 može se vidjeti niz instrukcija koji je generiran pomoću prevodioca. Instrukcije i operandi su označeni bojom ako su generirani od traženih izraza i to plavom za *IF*, zelenom za *WHILE*, žutom za *FOR*. Ukoliko je model krivo označio dio slijeda, u crvenoj boji je označena klasa koju je model generirao. Za dani primjer model je krivo označio određene instrukcije i operande. Početni izraz *IF*, je dobro označio nakon čega je instrukcije i operande krenuo označavati oznakom za *FOR* izraz umjesto *WHILE* oznakom. Nadalje, sljedeću *FOR* petlju je označio sa *WHILE* oznakom potpuno zamjenjujući izraze. Mogući razlog takvog ponašanja je specifičnost danog primjera. Izraz *IF* i *WHILE* petlja koriste zajedničku varijablu za provjeravanje uvijeta te izraz *IF* povećava varijablu za jedan. Ukoliko se koriste različite varijable za uvijet, model označi pravilne instrukcije, ali još uvijek s krivom oznakom *FOR* petlje za *WHILE* petlju. U slučaju da se *WHILE* petlja izdvoji u posebnu metodu, model je pravilno klasificira.

U bazi primjera predominantne su kraće metode bez puno ugnježđenja i različitih izraza što utječe na sposobnost modela da nauči obrazac kompliciranijih primjerka. Dio metoda se može ignorirati pri ovakvom zadatku, s obzirom da su sva 3 izraza koja tražimo ovisna o instrukciji skoka. Kada se filtriraju sve metode bez instrukcije skoka

ostaje samo trećina skupa za učenje. Točnost takvog modela je približno ista kao i prošlog te iznosi 97.14%, ali razlika je što je dani primjer bolje označen. Mjesta izraza na instrukcijama su dobro pogođena samo što su i *WHILE* i *FOR* instrukcije označene kao *WHILE*.

```
iconst_1
istore_1
getstatic
java/lang/System.out:Ljava/io/PrintStream;
new
java/lang/StringBuilder
dup
invokespecial
java/lang/StringBuilder.<init>:()V
ldc
invokevirtual
java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
iload_1
invokevirtual
java/lang/StringBuilder.append:(I)Ljava/lang/StringBuilder;
invokevirtual
java/lang/StringBuilder.toString:()Ljava/lang/String;
invokevirtual
java/io/PrintStream.println:(Ljava/lang/String;)V
iload_1
iconst_1
if_icmple
7
iload_1 -> FOR
iconst_1 -> FOR
iadd -> FOR
istore_1 -> FOR
iload_1 -> FOR
bipush -> FOR
10 -> FOR
if_icmpge -> FOR
35 -> FOR
getstatic
java/lang/System.out:Ljava/io/PrintStream;
new
java/lang/StringBuilder
dup
invokespecial
java/lang/StringBuilder.<init>:()V
ldc
..
invokevirtual
java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
iload_1
invokevirtual
java/lang/StringBuilder.append:(I)Ljava/lang/StringBuilder;
invokevirtual
java/lang/StringBuilder.toString:()Ljava/lang/String;
invokevirtual
java/io/PrintStream.println:(Ljava/lang/String;)V
iload_1
iconst_1
iadd
istore_1
goto
-35
iconst_0 -> WHILE
istore_2 -> WHILE
iload_2 -> WHILE
```

```
iconst_2 -> WHILE
if_icmpge -> WHILE
12 -> WHILE
invokestatic
Test1.sayHello:()V
iinc
#2
-89
goto -> WHILE
-11 -> WHILE
return
```

Kod 5.2: Klasifikacija primjera

5.2. Model generiranja sekvence

Model generiranja sekvence izraza je puno složeniji i opsežniji problem nego prijašnji zadatak. Koristio se višeslojni dvosmjerni povratni model za enkoder, te povratna mreža s mehanizmom pažnje kao dekoder. Parametri enkodera su 600 jedinica za dimenziju reprezentacije ulaznih znakova, isto tako za veličinu ćelije dekodera i enkodera kao i veličinu kontekstnog vektora mehanizma pažnje. Enkoder se proteže kroz 3 sloja povratne mreže. Za optimizaciju se koristio Adam (*engl. Adaptive Moments*) [13] jer je brže prilazio minimumu od stohastičkog gradijentnog spusta.

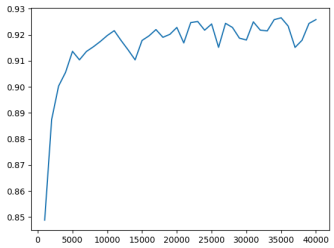
Za primjer *main* funkcije u kodu 5.1 očekivani izlaz ovog zadatka se može vidjeti u kodu 5.3.

```
LocalVariableDeclaration
StatementExpression
END_STATEMENT
IfStatement
StatementExpression
END_STATEMENT
END_STATEMENT
LocalVariableDeclaration
WhileStatement
StatementExpression
END_STATEMENT
StatementExpression
END_STATEMENT
END_STATEMENT
ForStatement
StatementExpression
END_STATEMENT
END_STATEMENT
SEQUENCE_END
```

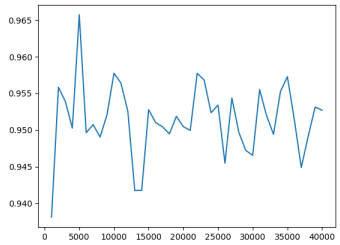
Kod 5.3: Primjer očekivanog izlaza sekvencijskog modela

Rezultati na skupu podataka za evaluaciju su prikazani na slici 5.3. Dani rezultati su dobiveni koristeći metodu ranog zaustavljanja kako bi se spriječila prenaučenost.

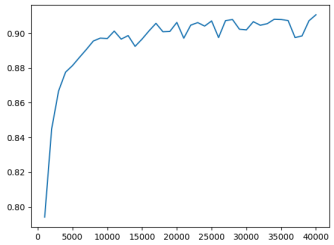
Odziv stabilno raste kontinuiranim učenjem što znači da se sve veći postotak potrebnih izraza prepoznaje, no preciznost dobivenog niza varira, s obzirom da model neke obrasce ponavlja te sa manje informacija nastoji zaključiti koji izraz je potrebno generirati.



(a) f1 mjera



(b) Preciznost



(c) Odziv

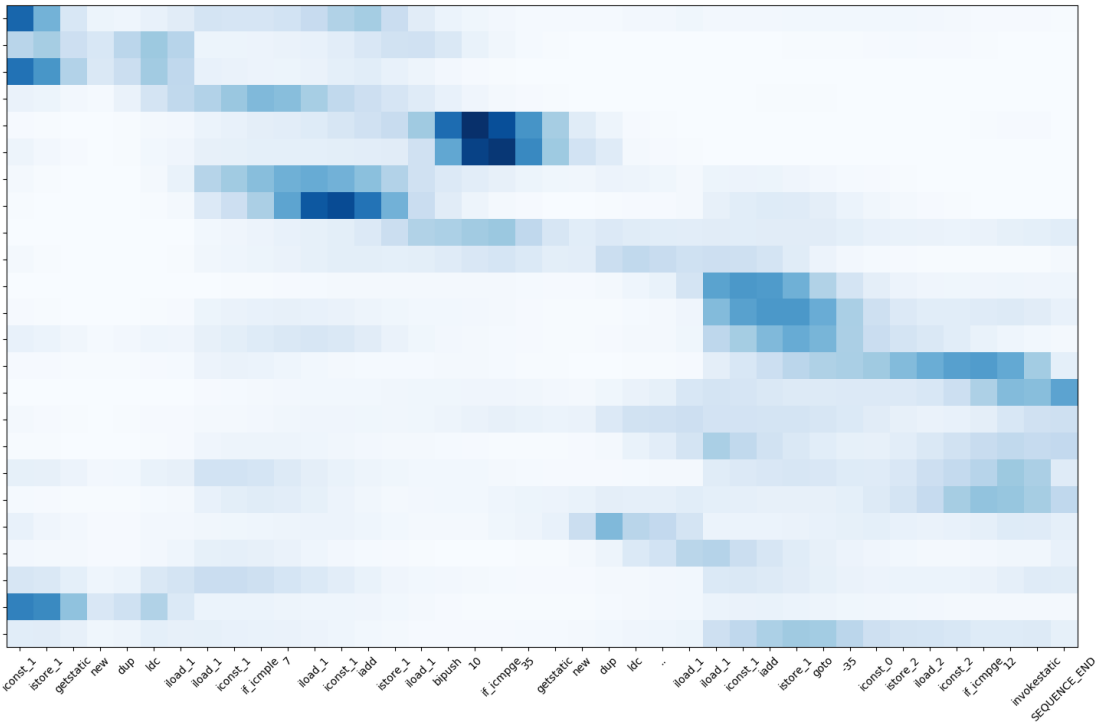
Slika 5.3: Rezultati na skupu za evaluaciju

Tako za primjer u kodu 5.1 dobiveni izlaz se nalazi u kodu 5.4. Početak metode je točno prepoznat, zatim slično se, kao sa prvim modelom, *WHILE* petlja zamjenjuje s *FOR* petljom, potom slijedi još jedna *FOR* petlja i pri kraju netočno nadodani izrazi. Radi boljeg razumjevanja izvođenja modela, na slici 5.4 mogu se vidjeti upaljene težine mehanizma pažnje. U svakom retku se nalazi prikaz težina vektora konteksta za generirani izraz, gdje zadnji izraz (redak) je oznaka kraja niza. U svakom stupcu je oznaka jačine koeficijenta koji se množi sa skrivenim stanjem enkodera za određenu instrukciju ili operand. Svaka instrukcija ili operand je reprezentirana zadnjim slojem skrivenog stanja dvosmjernog enkodera, stoga reprezentacija sadrži informaciju o cijelom nizu to jest metodi. Iz slike 5.4 se vidi kako je dekoder za generiranje izraza uvjeta (*IfStatement*, četvrti redak) najviše pažnje posvetio instrukcijama koje su generirane iz tog izraza, a to su ujedno instrukcije koje su bile označene u modelu u prvom zadatku (kod 5.2). Slično je više pažnje obraćeno pri generiranju oznake kraja tog izraza (sedmi redak). Zatim, model je napravio istu grešku kao model u prvom zadatku te generirao oznaku *FOR* petlje na temelju istog područja instrukcija kao što je prvi model ozna-

čio. U prvom zadatku model je učio na temelju označenih instrukcija, dok je u ovom zadatku bilo potrebno generirati oznake izraza cijele metode, a označene instrukcije je sam naučio pomoću mehanizma pažnje model kako bi generirao izraze. Na kraju se pojavljuju dodatni izrazi koji su generirani ponovnim sagledavanjem ulaznog niza, moguće zbog prenaučenog dekodera koji loše određuje oznaku kraja.

```
LocalVariableDeclaration
StatementExpression
END_STATEMENT
IfStatement
StatementExpression
END_STATEMENT
END_STATEMENT
ForStatement
StatementExpression
END_STATEMENT
END_STATEMENT
StatementExpression
END_STATEMENT
ForStatement
StatementExpression
END_STATEMENT
END_STATEMENT
IfStatement
StatementExpression
END_STATEMENT
END_STATEMENT
ReturnStatement
END_STATEMENT
SEQUENCE_END
```

Kod 5.4: Primjer dobivenog izlaza sekvencijskog modela



Slika 5.4: Upaljene težine mehanizma pažnje, u svakom retku za jedan generirani znak (dio operanada je uklonjen radi preglednosti). Tamnija plava boja označava veći iznos koeficijenta.

6. Zaključak

Duboke neuronske mreže zadnjih godina su se naglo počele razvijati, od mogućnosti izgradnje sve dubljih mreža, do novih prilagodljivih arhitektura. U ovom radu primijenile su se neke od tehnika dubokog učenja u svrhu analize i dekompilacije Java bajtkoda. Povratne neuronske mreže postigle su zanimljiv razvoj i primjenu u domeni prevođenja prirodnih jezika, no kako bi se sličan pristup mogao primijeniti na prevođenje umjetnih jezika sa striktnom strukturom potrebno je prilagoditi postojeće metode.

Ovaj rad je sagledao i testirao kako se postojeće metode mogu koristiti da bi se iz izvršnog koda analizirale informacije o postojećoj strukturi i poretku izvornog koda u svrhu dekompajliranja. Izrađena su dva modela koji indiciraju kakva je podležeća struktura izvornog koda. U prvom modelu koristile su se povratne neuronske mreže kako bi se označile instrukcije koje je prevodioc generirao iz izraza uvjeta i petlji. Model nije bio u mogućnosti savršeno naučiti obrasce takvih izraza, te je izraze ponekad zamjenjivao ili bi povezao niz instrukcija koje bi se nalazile u blizini traženog izraza. Drugi model se sastojao od povratnih mreža koje su sačinjavale enkoder i dekoder. Enkoder je bio zadužen za stvaranje vektorske reprezentacije ulaznih instrukcija koje su se potom prevodile u oznake početka i kraja izraza Java izvornog koda. Model je uspješno prepoznavao kraće slijedove izraza, ali dugoročno nije imao kapacitet da prepozna složeniji i duži niz. Pomoću mehanizma pažnje moguće je označiti instrukcije i operande koji su najznačajniji za generiranje trenutnog izraza te se uviđa se djelomično preklapanje označenih instrukcija s prvim modelom.

Umjetne neuronske mreže se intenzivno istražuju, te se konstantno nadograđuju i nadodaju koncepti koji povećavaju mogućnosti mreža da kvalitetno izvlače informacije. Nove arhitekture, poput neuronskih Turingovih strojeva i neuronskog programiranja, povećavaju širinu mogućnosti koje je moguće izvesti s neuronskim mrežama te tako mogu pomoći i za daljnje razvijanje ovog područja.

LITERATURA

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, i Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

[2] Mayank Agarwal. Back propagation in convolutional neural networks – intuition and code, 2017. URL `https://bit.ly/2s6lZto`.

[3] Dzmitry Bahdanau, Kyunghyun Cho, i Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014. URL `http://arxiv.org/abs/1409.0473`.

[4] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, i Martin T. Vechev. Statistical deobfuscation of android applications. U *ACM Conference on Computer and Communications Security*, stranice 343–355. ACM, 2016.

[5] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 0387310738.

[6] Denny Britz. Recurrent neural networks tutorial, part 1 – introduction to rnns, 2015. URL `https://bit.ly/1PbAMJ4`.

[7] Denny Britz, Anna Goldie, Thang Luong, i Quoc Le. Massive Exploration of Neural Machine Translation Architectures. *ArXiv e-prints*, mar 2017.

[8] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, i Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[9] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, i Zhenkai Liang. Neural nets can learn function type signatures from binaries. U *USENIX Security Symposium*, stranice 99–116. USENIX Association, 2017.

[10] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, Dec 1989. ISSN 1435-568X. doi: 10.1007/BF02551274. URL `https://doi.org/10.1007/BF02551274`.

[11] Xavier Glorot, Antoine Bordes, i Yoshua Bengio. Deep sparse rectifier neural networks. U Geoffrey Gordon, David Dunson, i Miroslav Dudík, urednici, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, svezak 15 od *Proceedings of Machine Learning Research*, stranice 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR. URL `http://proceedings.mlr.press/v15/glorot11a.html`.

[12] Robert Grosse. Krakatau - java decompiler, assembler, and disassembler. URL `https://github.com/Storyyeller/Krakatau`.

[13] Diederik P Kingma i Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[14] Tim Lindholm, Frank Yellin, Gilad Bracha, i Alex Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st izdanju, 2014. ISBN 013390590X, 9780133905908.

[15] Thang Luong, Hieu Pham, i Christopher D. Manning. Effective approaches to attention-based neural machine translation. U *EMNLP*, stranice 1412–1421. The Association for Computational Linguistics, 2015.

[16] Warren S. McCulloch i Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, Dec

1943. ISSN 1522-9602. doi: 10.1007/BF02478259. URL https://doi.org/10.1007/BF02478259.

[17] Jerome Miecznikowski i Laurie J. Hendren. Decompiling java bytecode: Problems, traps and pitfalls. U *CC*, svezak 2304 od *Lecture Notes in Computer Science*, stranice 111–127. Springer, 2002.

[18] Tomas Mikolov, Kai Chen, Greg Corrado, i Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[19] Marvin Minsky i Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, USA, 1969.

[20] Christopher Olah. Neural networks, types, and functional programming, 2015. URL http://colah.github.io/posts/2015-09-NN-Types-FP/.

[21] Christopher Olah. Understanding lstm networks, 2015. URL http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[22] Kishore Papineni, Salim Roukos, Todd Ward, i Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. U *Proceedings of the 40th annual meeting on association for computational linguistics*, stranice 311–318. Association for Computational Linguistics, 2002.

[23] Se Hoon Park. Understanding jvm internals, 2017. URL https://www.cubrid.org/blog/understanding-jvm-internals/.

[24] Raúl Rojas. *Neural Networks - A Systematic Introduction*. Springer, 1996.

[25] Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, i Alexey Loginov. Evolving exact decompilation. U *Binary Analysis Research (BAR), 2018*, 2018.

[26] Eui Chul Richard Shin, Dawn Song, i Reza Moazzezi. Recognizing functions in binaries with neural networks. U *24th USENIX Security Symposium (USENIX Security 15)*, stranice 611–626, Washington, D.C., 2015. USENIX Association. ISBN 978-1-931971-232. URL https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/shin.

[27] Ilya Sutskever, Oriol Vinyals, i Quoc V Le. Sequence to sequence learning with neural networks. U *Advances in neural information processing systems*, stranice 3104–3112, 2014.

[28] Avinash Sharma V. Understanding activation functions in neural networks, 2017. URL https://bit.ly/2wv3kcd.

[29] P.J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Harvard University, 1975. URL https://books.google.hr/books?id=z81XmgEACAAJ.

[30] Ashia C. Wilson, Rebecca Roelofs, Mitchell Stern, Nati Srebro, i Benjamin Recht. The marginal value of adaptive gradient methods in machine learning. U Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, i Roman Garnett, urednici, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, stranice 4151–4161, 2017. URL https://bit.ly/2yE6K12.

Dekompajliranje Java aplikacija potpomognuto metodama strojnog učenja

Sažetak

Razvojem strojnog učenja i povećanjem proceduralne moći računala, otvorio se novi način za pristupanju problemima. Metode strojnog učenja, kao što su neuronske mreže, omogućavaju iskorištavanje velike količine podataka kako bi se u sisteme ugradilo znanje o obrascima i pravilima bez direktnog programiranja, a u svrhu rješavanje problema. U ovom će se radu razmotrit moguća primjena umjetnih neuronskih mreža za rješavanje problema dekompajliranja Java bajtkoda. Prezentirana su dva modela koji nastoje otkriti strukturu apstraktnih izraza izvornog jezika iz niza instrukcija koje su zapisane u izvršnom formatu. Prvi model koristi povratne neuronske mreže kako bi označio instrukcije koje su generirane iz izraza uvjeta i petlja u izvornom kodu. Drugi model koristi povratne mreže kako bi iz niza instrukcija generirao oznake početaka i kraja izraza izvornog jezika. Modeli postižu minimalnu funkcionalnost prepoznavajući najjasnije obrasce izraza. Kod dužih i kompliciranijih nizova otkrivaju tek dio strukture te nisu u mogućnosti naći jasne granice izraza.

Ključne riječi: Java, Strojno učenje, Neuronske mreže, Dekompajliranje

Decompilatoin of Java Application Supported by Machine Learning Algorithms

Abstract

By development of machine learning techniques and increasing computer process abilities, new approach to solving problems has been presented. Machine learning methods like neural networks allow for analysis of large amounts of data to include knowledge of patterns and rules without usage of direct programming for solving problems. This paper will consider application of artificial neural networks for the problems of decompiling Java bytecode. Two models have been introduced to find the structure of abstract expressions of the original language from a series of instructions from a compiled code. The first model uses recurrent neural network to classify the instructions that were generated from conditional and loop expressions in the source code. The other model uses a recurrent neural networks to generate start and end labels of source expressions. Models achieve minimal functionality by recognizing the clearest patterns of expressions. On longer and more complicated array of instructions, model reveal only a part of the structure and is not able to find clear boundaries of expressions.

Keywords: Java, Machine Learning, Neural networks, Decompiling