SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1676

Dekompajliranje aplikacija pisanih u programskom jeziku C/C++ potpomognuto metodama strojnog učenja

Fran Varga

Zagreb, rujan 2018.

SADRŽAJ

1. Uvod

Svake godine nastaje velika količina novog zloćudnog koda. Izvori navode da se na mjesečnoj bazi stvori preko 5 milijuna novih zloćudnih programa [4]. U borbi protiv zabrinjavajuće statistike je ključna analiza binarnih datoteka. Analiza se koristi jer izvorni kod za zloćudne programe gotovo uvijek nije dostupan. Bez izvornog koda teško je shvatiti logiku programa jer prevoditelji ne čuvaju informacije jezika više razine, poput tipova podataka i drugih apstrakcija. Reverzno inženjerstvo stoga je nužan korak za određivanje funkcionalnosti binarnih datoteka. Dostupni dekompajleri koriste specifično domensko znanje prevoditeljskih konvencija i specijalizirane tehnike za binarnu analizu. Proces dekompajliranja ovisi o arhitekturi, prevoditelju, instrukcijskom setu i drugim varijablama. Dekompajleri se također moraju nadograđivati za svaku novu arhitekturu koja se pojavi na tržištu, ali i važnije za svaku novu verziju prevoditelja.

Kako bi se smanjila količina napora potrebnog za rekonstrukciju logike programa bilo bi zanimljivo metodama strojnog učenja automatizirati dio posla nalaženja logičkih dijelova programa. Iako je ovo područje istraživanja još u povojima postoje neki zanimljivi radovi koji bi mogli ukazati na put do rješenja. Chua et al. [8] su konstruirali sustav strojnog učenja koji s točnošću 81% do 84% može naučiti funkcijske potpise, odnosno tip povratne vrijednosti i tipove i broj argumenata. Svoj sustav su nazvali `EKLAVYA`, a temelji se na povratnim mrežama koje provode učenje nad čitkim instrukcijama strojnog koda, odnosno vrše jezično procesiranje i grupiranje instrukcija u logičke skupine poput memorijske instrukcije ili aritmetičke instrukcije. Iz takvih nizova zatim provode odluku na temelju konteksta instrukcija. Na temelju rukovanja memorijskim adresama odlučuje se o tipu i broju argumenata.

Drugi zanimljivi program je `ByteWeight` kojeg su konstruirali Bao et al. [5]. Metodama strojnog učenja konstruirali su alat koji nalazi početke i krajeve funkcija u binarnim datotekama s većim postotkom uspješnosti od dosadašnjih alata koji se koriste, poput `IDA`-e. Zanimljivije je da alat sam uči koje su glavne značajke i informacije bitne za problem, pa se može primijeniti na nove platforme, prevoditelje i optimizacijske postupke. Potrebno ga je samo trenirati na novim skupovima podataka.

Još jedan zanimljiv rad se više bavi točnošću dekompilacije. Schulte et al. [21] su konstruirali evolucijski algoritam koji kao uvjet ima jednakost okteta. Alat evoluira izvorni kod koji se dobije izlazom nekog dekompajlera, ili se nađe nasumično, dok se ne zadovolji određeni postotak jednakosti okteta prevedenog evoluiranog koda i ciljne datoteke koja se dekompajlira. Ovaj postupak daje čitak i prevodljiv izvoran kod, ali zahtjeva velike baze koda.

Ovaj rad će se fokusirati na traženje načina automatizacije konstrukcije baze podataka za strojno učenje nad binarnim datotekama generiranih iz izvornog koda C programskog jezika. Nama nije poznato da postoje baze binarnih datoteka s označenim instrukcijama u ovu svrhu. Dobar alat za označavanje binarnih datoteka bi puno pomogao u daljnjim razmatranjima ovakvih problema. Prvi dio rada će se fokusirati na konstrukciju alata za označavanje objektnog koda programa pisanih u C programskom jeziku.

Također istražiti će se kako iskoristiti konvolucijske mreže u svrhu dekompajliranja. Konvolucijske mreže su dobar kandidat za ovaj problem jer traže uzorke niže razine koji predstavljaju neki *razred* u kontekstu strojnog učenja. Pretpostavka je da razredi poput `FOR` imaju specifičan niz instrukcija na početku i na kraju svog bloka koji predstavljaju naredbu `for`. Konvolucija bi trebala moći naći početke i krajeve ovakvih razreda te označiti dijelove strojnog koda koji im pripadaju.

Rad je strukturiran na sljedeći način. U drugom poglavlju je opisan programski jezik C. Treće poglavlje opisuje `ELF` i `DWARF` format objektnih datoteka. U sljedećem poglavlju je opisan program za označavanje objektnog koda koji je dio sustava za strojno učenje. U petom poglavlju opisana je teoretska podloga strojnog učenja i konvolucijskih mreža. Šesto poglavlje sadrži rezultate i opis eksperimenta koji je proveden. Rad završava zaključkom i popisom literature.

2. Programski jezik C

Ovaj rad će proučavati programe napisane u programskom jeziku C. Programski jezik C je razvio Dennis Ritchie od 1969. do 1973. u Bell laboratorijima. Razvijen je za reimplementaciju `Unix` operacijskog sustava. Nije velik i kompaktno je opisan. Pogodan je za prevođenje jednostavnim prevoditeljima. Apstrakcije koje pruža su bazirane na konkretnim tipovima podataka i operacijama koje provode konvencionalna računala. Ulazno-izlazne operacije su operacije komunikacije s okolinom. Jezik se oslanja na knjižnice za ulazno-izlazne operacije. U isto vrijeme apstrakcije su dovoljno visoke razine da je omogućena prenosivost među računalima [19].

2.1. Kratki opis programskog jezika C

U nastavku je opisan programski jezik C po uzoru na knjigu Dennis Ritchiea i Brian Kernighana "The C programming language" [20].

Programski jezik C je jezik opće namijene i nije specijaliziran za neki poseban zadatak. On se još naziva i sustavski jezik jer je pogodan za kodiranje operacijskih sustava i prevoditelja, ali je korišten za pisanje programa iz različitih domena. Pruža osnovne konstrukte za kontrolu toka izvođenja dobro strukturiranih programa: grupiranje izraza, odlučivanje, biranje događaja, petlje i rani izlaz iz petlje.

Razni tipovi podataka su definirani programskim jezikom C. Osnovni tipovi su znakovi, cijeli brojevi i realni brojevi raznih preciznosti. Uz to postoji i skup tipova stvorenih pokazivačima, nizovima, strukturama i unijama. Izrazi se grade operatorima i operandama. Pokazivači pružaju strojno neovisnu aritmetiku adresa.

Funkcije mogu vraćati primitivne osnovne tipove, strukture, unije ili pokazivače. Svaka funkcija se može pozvati rekurzivno. Varijable se mogu deklarirati u blokovima. Funkcije programa pisanih u programskom jeziku C se mogu nalaziti u posebnoj izvornoj datoteci koja se prevodi zasebno. Varijable mogu biti dio funkcije, vidljive cijelom programu ili postojati u samo jednoj izvornoj datoteci.

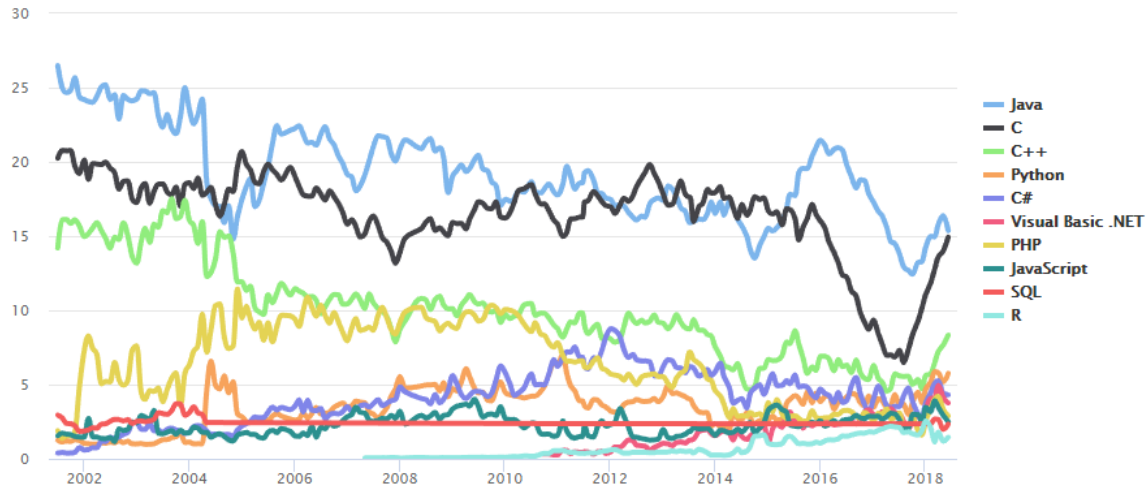
Pretprocesor provodi zamjenu makro definicija u programskom tekstu, uključivanje

drugih datoteka i uvjetno prevođenje.

Jezik je relativno niske razine. Ovo znači da najčešće koristi tipove podataka koje koriste i računala te vrši nad njima operacije koje implementiraju procesori. Programski jezik C ne pruža operacije za direktnu manipulaciju s nizovima, skupovima i listama, osim što se strukture mogu kopirati kao jedinice. Ne postoji sakupljač smeća osim za lokalne varijable i funkcije te nije definiran nikakav upravljač memorijom osim stoga. Ne postoje niti ulazno-izlazne metode, niti metode za rad s datotekama.

U jeziku je pružena samo najosnovnija kontrola toka jedne dretve bez podrške za paralelne operacije, sinkronizaciju i korutine.

1988. je napisan `ANSI` (American National Standards Institute) standard C programskog jezika. Programski jezik C jezik je svojom svestranošću i jednostavnošću impresionirao te se i danas koristi za rješavanje stvarnih problema. Na slici 2.1 može se vidjeti učestalost korištenja različitih programskih jezika u zadnjih 16 godina. Iz slike je vidljivo da je programski jezik C i dalje jedan od najpopularnijih jezika.



Slika 2.1: Korištenje programskih jezika, preuzeto iz [22]

2.2. Dekompajliranje C programa

Dekompajliranje je proces generiranja izvornog koda iz prevedenih programa spremnih za pokretanje. Prevođenje je proces s kojim su gotovo svi programeri dobro upoznati, pomisao bi bila da je i suprotan proces jednako razvijen. Ovo nije istina jer je dekompajliranje i dalje u fazi razvoja i postoji cijeli niz problema koji nisu riješeni. Tijekom prevođenja programi se optimiziraju za ciljnu platformu te se gube mnoge apstraktne informacije programa više razine. Uz gubitak informacija tijekom procesa prevođenja postoje razne tehnike obfuskacije koje su namjenjene skrivanju logike programa. Ovakve tehnike programeri koriste da bi onemogućili rekonstrukciju programske logike. Iako imaju svoje probleme, dekompajleri su i dalje važan alat u reverznom inženjerstvu. Da bi se dekompajlirao jedan program potrebno je puno ručnog rada. Na isječku koda 2.1 se vidi način rada dekompajlera Snowman [26]. Dekompajler je uspio rekonstruirati algoritam rekurzivne funkcije za računanje nekog člana Fibonaccijevog niza. Nije uspio rekonstruirati broj argumenata funkcije, te je dodao puno novih varijabli jer svaki međurezultat tretira kao posebnu operaciju pridruživanja. Dekompajler ne može vjerno rekonstruirati složene aritmetičke operacije, pa ih razdvaja u niz jednostavnijih operacija.

```c
int rek_fib(int a){
  if ((a==0)||(a==1))
    return 1;
  else
    return rek_fib(a-1)+rek_fib(a-2);
}
```

⇒

```c
int32_t text(int32_t a1, int32_t a2) {
  int32_t eax3;
  int32_t v4;
  int32_t eax5;
  int32_t v6;
  int32_t eax7;

  if (!a1 || a1 == 1) {
    eax3 = 1;
  } else {
    eax5 = text(a1 - 1, v4);
    eax7 = text(a1 - 2, v6);
    eax3 = eax7 + eax5;
  }
  return eax3;
}
```

Isječak koda 2.1: Izvorni kod rekurzivnog programa za računanje Fibonaccijevog niza, i isti program dekompajliran alatom Snowman

3. Formati objektnih datoteka

Za razumjevanje ovog rada potrebno je upoznati se s informacijama sadržanim u objektnim datotekama. Objektne datoteke su datoteke koje sadrže objektni kod. Objektni kod je prenosivi strojni kod (eng. *relocatable machine code*) koji često nije izvršiv. Opis objektnog koda mora sadržavati i informaciju kako se program pokreće. Objektne datoteke sadrže i upute operacijskom sustavu kako program učitati u memoriju i pokrenuti proces. Ovaj rad će se ograničiti na proučavanje objektnih datoteka `Unix` i sličnih operacijskih sustava prevedenih iz izvornog koda napisanog u programskom jeziku C. Neki od poznatih formata objektnih datoteka su `ELF` i `COFF`.

3.1. ELF format

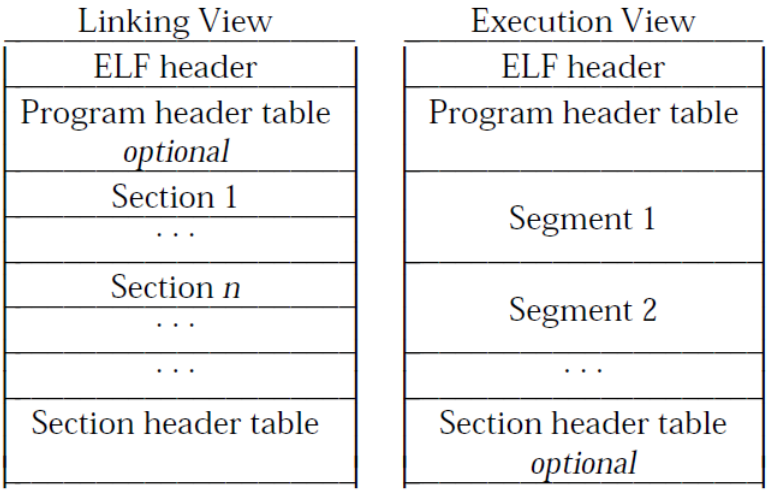
Jedan od danas najkorištenijih formata objektnih datoteka je `ELF` (eng. *Executable Linkable Format*) koji je postao standard na `*nix` operacijskim sustavima.

`ELF` je u svom začeću bio definiran kao standardni zapis objektnih datoteka za 32-bitnu Intel arhitekturu [23]. Razvijen je za Unix System V operacijski sustav. Kroz godine je prilagođen i upotrebljavan na mnogim drugim sustavima. Zamišljeno je da standard pruža skup definicija binarnih sučelja koja se mogu upotrijebiti na različitim računalnim platformama [23].

Postoje 3 vrste `ELF` datoteka:

- izvršna datoteka (eng. *executable file*)- sadrži objektni kod i podatke te specificira memorijski raspored procesa
- relokacijska datoteka (eng. *relocatable file*)- sadrži objektni kod i podatke pogodne za povezivanje s drugim objektnim datotekama, sadrži informacije o relokacijama, a relokacija je proces koji obavlja povezivač kada učitava instrukcije ovisne o smještaju u radnu memoriju
- dijeljeni objekt ili knjižnica (eng. *shared object file*) - sadrži objektni kod i podatke pogodne za povezivanje dinamičkim povezivačem

ELF formatom je definirana struktura datoteke kao niz zaglavlja i sekcija. Zaglavlja opisuju strukturu ostalih zaglavlja i sekcija, dok sekcije sadrže podatke. Tijekom procesa pokretanja datoteke sekcije se preslikavaju u memoriju procesa i u tom smislu se još nazivaju i segmenti. ELF format radi jednostavnosti omogućuje čitanje objektne datoteke na jedan od dva načina prikazana na slici 3.1. Ovi načini čitanja datoteke reflektiraju različite potrebe povezivača i operacijskog sustava. Način čitanja za povezivanje koriste prevoditelji i povezivači, dok način čitanja za pokretanje iskorištava operacijski sustav i programi za disasembliranje ili traženje pogrešaka u kodu.



Slika 3.1: Dva načina čitanja objektne datoteke, preuzeto iz [23]

Kako bi se pronašla neka informacija potrebno je poznavati cijelu strukturu datoteke jer su svi dijelove međuovisni. Na početku objektne datoteke nalazi se ELF zaglavlje. U ELF zaglavlju je zapisana većina informacije o datoteci i upute operacijskom sustavu kako pokrenuti program koji datoteka sadrži. Informacije koje su dostupne uključuju upute o učitavanju u memoriju, detalje o relokacijama, instrukcijski set koji se koristi, duljina riječi instrukcije i ostalo.

Svaka sekcija ima definirano zaglavlje u kojem pišu informacije o toj sekciji. Zaglavlja se ne nalaze u samoj sekciji kojoj pripadaju nego su pohranjena u tablici zaglavlja sekcija koja se nalazi na kraju datoteke. Korištenjem informacija iz zaglavlja sekcije moguće je pronaći sav objektni kod koji se nalazi u sekciji, ako ona sadrži objektni kod.

U zaglavlju sekcije pišu minimalno sljedeće informacije:

- `sh_name` - indeks znakovnog niza u zaglavlju imena sekcija
- `sh_type` - definira sadržaj sekcije

- `sh_flags` - sadrži zastavice koje opisuju dozvole sekcije, a najbitnija je `SHF_EXECINSTR` koja govori sadrži li sekcija strojne instrukcije
- `sh_addr` - sadrži adresu na koju bi se trebao alocirati prvi oktet sekcije u memoriji procesa
- `sh_offset` - sadrži udaljenost prvog okteta u sekciji u odnosu na početak datoteke
- `sh_size` - sadrži broj okteta u sekciji
- `sh_link`, `sh_info` - pomoćne varijable koje pobliže opisuju sekciju ovisno o tipu sekcije
- `sh_addralign` - ograničenja na poravnanje sekcije, veličina lokacije koja se adresira
- `sh_entsize` - neke sekcije su tablice zapisa jednakih veličina, `sh_entsize` definira veličinu jednog zapisa tablice u oktetima

Čitanjem `sh_name`, `sh_addr`, `sh_offset` i `sh_size` atributa može se direktno pronaći ime i veličina sekcije, njen odmak u datoteci i odmak u memoriji procesa za vrijeme pokretanja. Standardno je da se objektni kod sprema u sekciju imena `.text`. Tražeći sekcije imena `.text`, pronalazi se većina objektnog koda programa. Da bi se pronašao sav objektni kod potrebno je dohvatiti sve sekcije koje imaju postavljenu zastavicu `SHF_EXECINSTR` u atributu `sh_flags`. Adresa sekcije u memoriji procesa je bitna jer će kod koraka disasembliranja sve instrukcije biti označene po svojoj heksadekadskoj adresi u memoriji procesa, a ne odmakom u datoteci. Ovo je bitno jer će se kasnije vidjeti da su informacije o vezi izvornog koda i objektnog koda zapisane u odnosu na adrese u memoriji procesa, a ne u datoteci, osim ako je riječ o relokacijskim datotekama.

3.2. DWARF format

Informacije o vezi izvornog i objektnog koda su osnova rada programa za označivanje objektnog koda oznakama razreda za strojno učenje. Informacije su zapisane u dijelovima objektne datoteke strukturiranih `DWARF` formatom. `DWARF` (eng. *Debugging Information Format*) je format za zapisivanje informacija za traženje pogrešaka u programima [23]. `DWARF` informacije se zapisuju u sekcijama objektne datoteke čija imena imaju oblik `.debug_<name>`, gdje je `<name>` varijabla ovisna o tipu informacija koji se zapisuje. Neke od mogućih varijacija su `.debug_line`, `.debug_info`

i .debug_types. Čitanje DWARF informacija omogućuje dublji uvid u strukturu ELF objektne datoteke. Informacije DWARF formata pohranjene su u stablu DIE-va (eng. *Debugging Information Entry*). DIE-vi u stablu su čvorovi koji mogu posjedovati proizvoljan broj djece. Jedan DIE i njegova djeca potpuno opisuju jedan entitet u programu poput funkcije, varijable ili kompilacijske jedinice. DIE sadrži identifikacijsku oznaku i može imati niz atributa, a svaki atribut ima pridruženu vrijednost koja ga određuje. Stablo je zapisano obrnutom Poljskom notacijom (eng. *postfix order*).

Informacije o povezanosti izvornog i objektnog koda nalaze se u DIE-vima koji imaju oznaku DW_TAG_compilation_unit ili DW_TAG_subprogram. Ovi DIE-vi opisuju pojedinu funkciju ili kompilacijsku jedinicu programa. Kompilacijska jedinica (eng. *Compilation Unit*, CU) sadrži informacije o svom izvornom kodu koji se prevodi i tretira se kao jedna logička jedinica. Za svaku funkciju u objektnoj datoteci postoji DIE u kojem pridruženo atributu DW_AT_name piše njeno ime, a ima oznaku DW_TAG_subprogram. Par atributa DW_AT_low_pc i DW_AT_high_pc ili atribut DW_AT_ranges sadrže informaciju o kontinuiranom ili nekontinuiranom slijedu strojnih instrukcija koje su generirane iz izvornog koda pojedine funkcije. DIE-vi koji opisuju neku funkciju pretežno su djeca DIE-va označenih s DW_TAG_compilation_unit. Iteriranjem po svim CU-ovima i traženjem svih DIE-va označenih s DW_TAG_subprogram moguće je brzo pronaći sve funkcije sadržane u objektnoj datoteci i objektni kod koji je iz njih generiran.

Povezivanje objektnog koda s funkcijom međutim nije dovoljno, pa je potrebno pronaći DWARF informacije o linijama izvornog koda iz koje su neke strojne instrukcije generirane. Ove informacije su obično sadržane u .debug_line sekciji, a zapisane su u *matrici informacija o linijama*.

U matrici linijskih informacija su zapisane sljedeće informacije:

- address - vrijednost programskog brojila PC (eng. *program counter*) za pojedinu instrukciju
- op_index -jako duge instrukcije VLIW (eng. *Very Long Instruction Word*) zapisuju se kao niz instrukcija, ako je instrukcija dio VLIW niza u op_index je zapisan njen indeks, VLIW instrukcije su pogodne za paralelno izvođenje
- file - identifikator datoteke s izvornim kodom
- line - broj linije u datoteci iz koje je generirana instrukcija, numeriranje počinje s 1, a 0 je zapis kada je nemoguće odrediti liniju iz koje je instrukcija generirana
- column - broj stupca iz linije, numeriranje također počinje s 1

- `is_stmt` - indicira preporučeno mjesto za prekidnu točku te bi trebao reprezentirati liniju, izraz ili semantički jedinstven dio izraza
- `basic_block` - označava početak osnovnog bloka
- `end_sequence` - označava kraj povezanog niza strojnih instrukcija
- `prologue_end` - označava da je instrukcija kraj funkcijskog prologa i pogodna je za postavljanje prekidne točke za ulaz u funkciju
- `epilogue_begin` - označava da je instrukcija početak funkcijskog epiloga i pogodna je za postavljanje prekidne točke za izlaz iz funkcije
- `isa` - identifikator primjenjivog instrukcijskog seta
- `discriminator` - identifikator osnovnog bloka kojem instrukcija pripada

Matrica informacija o linijama zauzima veliki memorijski prostor, pa je radi sažimanja stvoren poseban kod širine jednog okteta za kompresiju te matrice. Taj kod se naziva *program za brojeve linija* (eng. *Line Number Program*). Program za brojeve linija treba interpretirati konačnim automatom stanja koji generira matricu linijskih informacija iz koda. Konačni automat stanja koji se koristi u svrhu generiranja matrice linijskih informacija je definiran u specifikaciji `DWARF` formata. Program za brojeve linija ima svoje zaglavlje u kojem je definirano optimalno kodiranje ovisno o arhitekturi.

Nakon što se generira matrica linijskih informacija dovoljno je proći kroz matricu i provjeriti za svaku memorijsku adresu postoji li u matrici. Nakon toga iščitati sve zanimljive informacije postaje trivijalno. Ovakve matrice su sadržane u sekciji jedanput za svaki `CU` iz kojeg je objektna datoteka generirana.

4. Program za označavanje objektnog koda

U ovom poglavlju bit će objašnjena konstrukcija programa za označavanje koda korištenjem isključivo informacija sadržanih u objektnoj i izvornoj datoteci. Za problem označavanja objektnog koda oznakama razreda korištenih u strojnom učenju neprakti-čno bi bilo zaposliti ljude da ručno označe cijeli skup podataka, pa je odlučeno napraviti automatizirani označivač. Automatsko označavanje podataka je povoljno za ovaj problem zbog velike količine javno dostupnog izvornog koda na internetu. Javni repozitoriji poput GitHub-a sadrže velike količine javnog izvornog koda. Program je u potpunosti ostvaren u programskom jeziku Python. Kod programa dostupan je na GitHub repozitoriju rada [24].

4.1. Sakupljanje podataka

Koliko je nama poznato ne postoje javno dostupne baze označenog objektnog koda programa pisanih u programskom jeziku C pa je iznimno bitno automatizirati generiranje baze podataka iz velike količine programa s javno dostupnim izvornim kodom. Izvorni kodovi programa koji će biti dio skupa podataka su odabrani ručno.

Program za označavanje koda mora moći primiti objektnu datoteku i iz nje generirati niz označenih strojnih instrukcija. Svaku instrukciju mora označiti sa svim naredbama izvornog koda u koje je enkapsulirana. Radi lakšeg parsiranja baze podataka, instrukcije se trebaju grupirati po funkcijama.

Program je ostvaren tako da radi na `git` repozitorijima koda pisanog u programskom jeziku C koji se mogu prevesti alatom `make` i `gcc`. Alat `make` služi za praćenje promjena u direktorijima izvornog koda radi smanjenja potrebe za prevođenjem velikog broja nepromijenjenih datoteka, a alat `gcc` je jedan od mogućih prevoditelja za programski jezik C. Alatu `make` je moguće zadavati `gcc` zastavice za cijeli projekt postavljanjem varijabli okoline, dodavanjem zastavica u `./configure` skriptu

ili dodavanjem zastavica prilikom pokretanja prevođenja. Zastavice koje je potrebno postaviti da bi se generirala baza podataka su zastavica `-g` i jedna od zastavica `-O0, -O1, -O2, -O3`. Zastavica `-g` govori prevoditelju da generira `DWARF` informacije i ugradi ih u objektne datoteke, dok zastavice `-O0, -O1, -O2, -O3` označuju koja razina optimizacije se treba koristiti. Zastavica `-O0` označava optimizacijsku razinu 0 odnosno da se optimizacija ne primjenjuje. To je i standardno ponašanje prevoditelja [1].

4.2. Modul za povezivanje objektnog i izvornog koda

Korištenjem navedenih informacije dostupnih iz `DWARF` i `ELF` formata, moguće je napisati program koji iz objektne datoteke izdvaja sve funkcije te zatim označava svaku instrukciju iz funkcija s imenom datoteke i linijom koda iz koje je generirana.

Alat `addr2line`, dostupan na `*nix` sustavima kao dio `GNU binutils` paketa, obavlja zadatak određivanja linije, iz datoteke izvornog koda, za proizvoljnu adresu memorijske lokacije u programu. Alat radi sporo i teško ga je automatizirati. Zbog toga je odlučeno da se izradi nova verzija koristenjem Python programskog jezika i modula `pyefltools`.

Modul programskog jezika Python `pyelftools` [7] sadrži sve potrebne metode za parsiranje i razumijevanje `ELF` datoteka i `DWARF` informacija sadržanih u njima. Modul služi za gradnju alata koji izvlače informacije iz `ELF` datoteka. `pyelftools` je pogodan za lakšu manipulaciju s podatcima i izgradnju alata za rad s objektnim datotekama.

Informacije koje su zanimljive za problem označavanja objektnog koda razredima za strojno učenje su linije izvornog koda iz kojih je neka strojna instrukcija generirana. Program koji rješava ovaj problem sastoji se od 3 glavne funkcije `main`, `decode_funcname` i `decode_file_line`. Funkcija `main` čita `ELF` i `DWARF` informacije te ih prosljeđuje funkcijama `decode_funcname` i `decode_file_line`. Funkcija `decode_funcname` pronalazi sve memorijske adrese koje obuhvaćaju funkcije definirane u programu. Funkcija `decode_file_line` za svaku memorijsku adresu koja je pronađena funkcijom `decode_funcname` pronalazi izvornu datoteku i liniju koda kako bi omogućila povezivanje izvornog koda s objektnim kodom.

Kod za funkciju `decode_funcname` prikazan je u isječku koda 4.1. Funkcija prima `DWARF` informacije i listu valjanih adresa `.text` sekcije. Funkcija `decode_funcname` prolazi listom svih `CU`-ova nađenih u programu. Svaki `CU` ima proizvoljan broj djece, od kojih su neka potprogrami sadržani u `CU`. Svaki `DIE` potpro-

grama označen je oznakom DW_TAG_subprogram. Za svaki potprogram definirana je početna i završna memorijska adresa u koju će se alocirati potprogram u vrijeme pokretanja procesa. Zadatak funkcije decode_funcname je za svaki potprogram naći početnu i završnu memorijsku adresu u kojoj se nalazi i da iz svih adresa koje se nalaze u sekciji .text nađe sve one koje su generirane iz izvornog koda. Adrese koje su vezane za izvorni kod sprema u listu. Lista sadrži najvišu i najnižu memorijsku adresu za svaki potprogram.

```
1  def decode_funcname(dwarf_info, addresses):
2
3    func_l=list()
4    # za svaku kompilacijsku jedinicu
5    for CU in dwarf_info.iter_Cus():
6
7      # za svaki logički član kompilacijske jedinice
8      for DIE in CU.iter_DIEs():
9
10       #pronađi sve potprograme
11       if DIE.tag == 'DW_TAG_subprogram':
12         #nađi najvišu adresu potprograma
13         lowpc = DIE.attributes['DW_AT_low_pc'].value
14         #nađi najnižu adresu potprograma
15         highpc = DIE.attributes['DW_AT_high_pc'].value
16         #dodaj važeće adrese u listu
17         func_l.append( addresses.in(lowpc, highpc) )
18
19   return func_l
20
```

Isječak koda 4.1: decode_funcname definicija

Funkcija decode_file_line traži izvornu datoteku i liniju izvornog koda za svaku instrukciju. Funkcija prima DWARF informacije i listu pronađenih valjanih adresa svakog potprograma. Funkcija generira matricu linijskih informacija za svaki CU. Za svaki CU se gradi matrica te se zatim kroz nju prolazi sa svim adresama proslijeđenim funkciji. Proces se provodi zato što sve instrukcije koje neka funkcija obuhvaća nisu nužno iz iste izvorne datoteke niti se generiraju iz slijednih linija izvornog koda. Primjer za ovaj problem su inline funkcije. Inline funkcije su obično kratke funkcije čiji se objektni kod direktno kopira u objektnog koda pozivne funkcije, ali označene su kao da dolaze iz druge funkcije. Objektni kod inline funkcija se ug-

nježđuje u objektni kod pozivne funkcije. Zbog ovog i sličnih problema potrebno je za svaku adresu proći kroz svaku matricu linijskih informacija. Najviše vremena se troši na generiranje matrica, pa je bitno da se za svaki `CU` matrica linijskih informacija generira samo jedanput. Također potrebno je uočiti da se adresa ne mora nalaziti u matrici informacija o linijama. Matrica je zapisana tako da za sve adrese koje se nalaze unutar pojasa adresa dva susjedna zapisa u matrici imaju jednake podatke kao zapis za najnižu adresu tog pojasa. Kod za `decode_file_line` funkciju nalazi se u isječku koda 4.2.

```
1  def decode_file_line(dwarf_info, addressses):
2    list_files = []
3
4    # za svaku kompilacijsku jedinicu
5    for CU in dwarf_info.iter_Cus():
6
7      #generiraj matricu linijskih podataka
8      lineprogram = dwarf_info.line_program_for_CU(CU)
9
10     #za svaku adresu
11     for address in addresses:
12
13       #prođi kroz matricu
14       for state in lineprogram.states:
15         #ako je adresa dio matrice
16
17         if prevstate and prevstate.address <= address < entry.state.
       address:
18           filename = lineprog['file_entry'][prevstate.file - 1].name
19           line = prevstate.line
20           #učitaj sve podatke i dodaj ih u rječnik
21           list_files.append(address,line,filename)
22       prevstate = state
23   return list_files
24
```

Isječak koda 4.2: `decode_file_line` definicija

4.3. Disasembliranje

Nakon izdvajanja funkcija i označavanja adresa s imenima izvorne datoteke i linijama iz koda, potrebno je obaviti disasembliranje. Disasembliranje je proces translacije

strojnog koda u asemblerski jezik. Ovo je bitan korak jer su do sada samo označene sve memorijske adrese u funkciji. Instrukcije mogu biti varijabilne duljine, što znači da jedna memorijska adresa ne sadrži jednu instrukciju, pa je potrebno grupirati memorijske adrese po instrukcijama. Disasembler iz niza memorijskih adresa koje pripadaju jednoj funkciji generira niz mnemoničkih asemblerskih instrukcija, uz oznake toka procesa. Čak i ako se ne namjeravaju koristiti instrukcije u formatu čitkom čovjeku potrebno je izvršiti ovaj korak radi rješavanja problema varijabilne duljine instrukcije.

Alat korišten za disasembliranje je `radare2` [3]. On je jedan od najpoznatijih alata za analizu binarnih datoteka, računalnu forenziku i disasembliranje. `Radare2` podržava pisanje skripti za mnoge jezike uz korištenje `r2pipe` cjevovoda. Skripta `r2script.py` obavlja disasembliranje i koristi funkcionalnosti `radare2` sustava opisanih u [3]. Izvorni kod skripte `r2script.py` se nalazi na GitHub repozitoriju rada u poddirektoriju `src` [24].

Nakon završetka disasembliranja dobivena je jedinstvena reprezentacija objektne datoteke kao niza strojnih instrukcija grupiranih po funkcijama, s oznakama datoteka i linija izvornog koda. Reprezentacija je prikazana u tablici 4.1.

Memorijska adresa	Instrukcijski kod	Mnemonički oblik	Linija izvora	Datoteka izvora
...				
0x40069e	0x31c0	xor eax, eax	12	A21.c
0x4006a0	0xc745e4000000	mov dword [rbp - 0x1c], 0	14	A21.c
0x4006a7	0x488d45d8	lea rax, [rbp - 0x28]	15	A21.c
0x4006ab	0x4889c6	mov rsi, rax	15	A21.c
0x4006ae	0xbf74084000	mov edi, 0x400874	15	A21.c
0x4006b3	0xb800000000	mov eax, 0	15	A21.c
...				

Tablica 4.1: Slijed instrukcija s označenom datotekom i linijama izvornog koda

4.4. Program za pridruživanje oznaka razreda

Program za pridruživanje oznaka razreda je dio pretprocesora koji označava podatak s nekom od proizvoljnog broja definiranih razreda za korištenje u algoritmima strojnog učenja. Odabrano je osam razreda za proučavanje: `IF`, `ELSE`, `FOR`, `WHILE`, `DO`, `SWITCH`, `CALL` i `FUNCTION`. Pretpostavka je da ovi razredi imaju distinktni uzorak niza strojnih instrukcija koji ih međusobno razlikuje. `IF`, `ELSE`, `FOR`, `WHILE`, `DO` i `SWITCH` razredi obuhvaćaju instrukcije koje su enkapsulirane u neku

od pripadnih naredbi programskog jezika C. `CALL` i `FUNCTION` razredi obuhvaćaju instrukcije generirane pozivom neke funkcije odnosno sami kod neke instrukcije. U tablici 4.2 može se vidjeti niz instrukcija koje obuhvaća razred `FOR`. Važno je primijetiti postavljanje brojača na memorijsku lokaciju `[rbp-4]` na početku, povećavanje brojaća i testiranje uvjeta uz uvjetni skok na kraju koji karakterizira sve `FOR` izraze.

```
0x400655        0xc745fc000000          mov dword [rbp - 4], 0
0x40065c        0xeb0e                  jmp 0x40066c
0x40065e        0x8b45f8                mov eax, dword [rbp - 8]
0x400661        0xfaf45f8               imul eax, dword [rbp - 8]
0x400665        0x8945f8                mov dword [rbp - 8], eax
0x400668        0x8345fc01              add dword [rbp - 4], 1
0x40066c        0x837dfc02              cmp dword [rbp - 4], 2
0x400670        0x7eec                  jle 0x40065e
```

Tablica 4.2: Sljed instrukcija generiranih iz `FOR` izraza

Do sada je opisan program koji nalazi objektni kod i linije izvornog koda. Zatim je potrebno naći nalazi li se na tim linijama neki od razreda. Svi razredi osim `CALL` i `FUNCTION` generiraju se iz pridruženih ključnih riječi koje se koriste kao imena razreda. U izvornome kodu nije teško pronaći ključne riječi `for, while, do, if, else` i `switch`. No, nalaženje ključnih riječi nije dovoljno. Linija kojom počinje niz instrukcija vezanih s npr. ključnom riječi `for` ne počinje na liniji gdje je `for`, nego na liniji gdje je prva vitičasta zagrada koja definira djelokrug `for`-a ili u slučaju kad `for` ne enkapsulira neki djelokrug tada niz instrukcija počinje na liniji gdje je točka-zarez koji završava izjavu. Instrukcije koje generira `for` nisu vezane uz ključnu riječ nego uz djelokrug koji ta naredba obuhvaća. Također bilo bi teško izdvajati razrede velikim brojem regularnih izraza zbog mogućnosti proizvoljnog korištenja razmaka i komentara. Zbog ovog odlučeno je koristiti leksički parser da bi se izdvojili svi leksikografski konstrukti iz izvornog koda. Pronaći konstrukt poput `for` i prvu lijevu vitičastu zagradu nakon njega je sada trivijalno, jer uključuje samo prolaz listom leksema.

Modul programskog jezika Python `pycparser` [6] je alat koji se koristio u svrhu leksičke analize. Svi prevoditelji vrše leksičku analizu, ali je nepraktično dohvatiti te podatke iz prevoditelja. Zato je odlučeno koristiti parser realiziran u sklopu `pycparser` modula. Parser nema mogućnost parsiranja komentara pa je stoga korišten prevoditelj `gcc` kao pretprocesor teksta da bi se iz datoteka izdvojili svi komentari. Također `pycparser` ima neke greške poput eksponencijalnog vremena

odlučivanja o pripadnosti nekog niza leksemu `string`, ako sadrži više predznačenih znakova (eng. *escaped character*). Ovakvi propusti su popravljeni preinakama u kodu označivača i modifikacijom poziva `pycparser` modula. Za konkretan problem stvoren je regularni izraz koji zamjenjuje svaki niz predznačenih znakova istim nizom razdvojenim zarezima ako je niz predznačenih znakova unutar navodnika.

Kada je konstruiran označivač moguće je naći koje su linije koda obuhvaćene kojim razredom. Sljedeći korak je povezivanje svake instrukcije s razredima kojima pripada. Ovo nije trivijalno jer instrukcija može pripadati većem broju razreda. Označivač konstruira listu ograda razreda, pa onda simultanim prolaskom kroz listu ograda i listu memorijskih adresa označava svaku adresu s potrebnim brojem razreda.

Ovime je zadatak programa koji priprema podatke završen. Primjer označenih instrukcija je prikazan na tablici 4.3. U datoteci označenih instrukcija nalazi se 6 stupaca koji pobliže opisuju niz instrukcija. Zapisana je memorijska adresa, kod instrukcije, mnemonički oblik instrukcije, linija izvornog koda, ime izvorne datoteke i niz oznaka. Broj kraj oznake označava indeks pojavljivanja jezičnog konstrukta, odnosno oznaka `FOR_2` označava drugo pojavljivanje `for` petlje u tijelu funkcije.

Memorijska adresa	Instrukcijski kod	Mnemonički oblik	Linija izvora	Datoteka izvora	Oznake
...					
0x40069e	0x31c0	xor eax, eax	12	A21.c	FUNCTION_2
0x4006a0	0xc745e4000000	mov dword [rbp - 0x1c], 0	14	A21.c	FUNCTION_2
0x4006a7	0x488d45d8	lea rax, [rbp - 0x28]	15	A21.c	FUNCTION_2 CALL_1
0x4006ab	0x4889c6	mov rsi, rax	15	A21.c	FUNCTION_2 CALL_1
0x4006ae	0xbf74084000	mov edi, 0x400874	15	A21.c	FUNCTION_2 CALL_1
0x4006b3	0xb800000000	mov eax, 0	15	A21.c	FUNCTION_2 CALL_1
0x4006b8	0xe873feffff	call sym.imp.__isoc99_scanf	15	A21.c	FUNCTION_2 CALL_1
0x4006bd	0xc745e8000000	mov dword [rbp - 0x18], 0	16	A21.c	FUNCTION_2 FOR_1
0x4006c4	0xeb1b	jmp 0x4006e1	16	A21.c	FUNCTION_2 FOR_1
0x4006c6	0xc745ec000000	mov dword [rbp - 0x14], 0	17	A21.c	FUNCTION_2 FOR_1 FOR_2
0x4006cd	0xeb08	jmp 0x4006d7	17	A21.c	FUNCTION_2 FOR_1 FOR_2
0x4006cf	0x8345e401	add dword [rbp - 0x1c], 1	18	A21.c	FUNCTION_2 FOR_1 FOR_2
0x4006d3	0x8345ec01	add dword [rbp - 0x14], 1	17	A21.c	FUNCTION_2 FOR_1 FOR_2
0x4006d7	0x837dec02	cmp dword [rbp - 0x14], 2	17	A21.c	FUNCTION_2 FOR_1 FOR_2
0x4006db	0x7ef2	jle 0x4006cf	17	A21.c	FUNCTION_2 FOR_1 FOR_2
0x4006dd	0x8345e801	add dword [rbp - 0x18], 1	16	A21.c	FUNCTION_2 FOR_1
...					

Tablica 4.3: Dio slijeda instrukcija koji generira pretprocesor

4.5. Mogućnosti alata za označavanje

Konstruirani alat ima svoja ograničenja. Alat se ne bavi makro definicijama kojima pretprocesor prevoditelja modificira izvorni kod. Velika količina informacija koje modificiraju izvorni kod poput uvjetnog definiranja varijabli, pa čak i definicije funkcija mogu biti u makro definicijama.

Alat se ne bavi povezanošću više izvornih datoteka. Ne traži postoji li neka varijabla ili funkcija definirana negdje drugdje nego prolazi kroz samo jednu datoteku. Ovo je problem jer će za bolje dekompajliranje biti potrebno izvlačiti veću količinu informacija poput tipova varijabli, tipova povratnih vrijednosti, definicija kompleksnih tipova i makro definicija. Radi konstrukcije ovakvog alata, potrebno bi bilo konstruirati algoritam dvostrukog prolaza kroz sve datoteke koji bi gradio bazu svih identifikatora, tipova itd.

Alat pronalazi razrede koji su definirani, ali ga nije jednostavno proširiti da prepoznaje konstrukte poput aritmetičkih operacija ili pridruživanja. Da bi se riješili ovakvi problemi trebalo bi provesti postupak sintaksne analize i tako konstruirati dobar skup razreda, te zatim izvlačiti linije koda koje su bitne. Radi potpunijeg razlaganja na razrede problem bi možda bilo bolje redefinirati kao problem prevođenja u neki međujezik. U alatu je već korištena leksička analiza da bi se našli jednostavni razredi.Kada bi se koristila i sintaksna analiza bilo bi moguće generirati potpuniji skup razreda. Sintaksna analiza bi naredbu `a=b+1;` istovremeno označila kao dio aritmetičkih operacija, operacija pridruživanja i naredbi. Kada bi se proveo postupak sintaksne analize samo bi ostala odluka na kojoj razini apstrakcije se želi odvajati razrede.

5. Teorijska podloga modelu konvolucijske mreže

Nakon što je konstruiran alat za označavanje objektnog koda oznakama razreda može se stvoriti skup podataka primjeren za algoritme strojnog učenja. Nad tim skupom podataka se može provesti strojno učenje u svrhu automatizacije utvrđivanja pripadnosti niza instrukcija nekoj logičkoj jedinici strojnog koda. Strojno učenje je način treniranja računala da riješi specifičan problem bez eksplicitnog kodiranja problema. Problem se rješava posredno, prezentiranjem računalu točnih rezultata za skup podataka i traženjem što bolje funkcije koja opisuje problem [16].

Strojno učenje se općenito koristi u 3 slučaja: kada je problem presložen i ljudi ne mogu ponuditi objašnjenje procesa, kada postoji ogromna količina podataka u kojima možda postoji neko znanje te kada postoji sustav koji se dinamički mijenja [16]. Sva tri slučaja su primjenjiva na problem dekompajliranja. Ako se problem redefinira kao problem traženja inverzne funkcije prevođenja izvornog koda nastaje prvi slučaj teškog problema za koji ne postoji algoritamsko rješenje. Postoje ogromne količine izvornog koda na raznim javnim repozitorijima iz kojih je moguće izvući znanje, a svaka nova verzija prevoditelja čini sustav dinamičnim.

5.1. Nadzirano učenje i unakrsna provjera

Proces treniranja nekog modela strojnog učenja na označenim uzorcima naziva se nadzirano učenje. Kada se provodi nadzirano učenje svaki primjer označen je pripadajućom oznakom razreda prije postupka treniranja. Kada uzorci nisu označeni radi se o nenadziranom učenju. Nadzirano učenje na temelju oznaka pokušava odrediti proces predviđanja koji može za svaki novi primjer odrediti pripadnost nekom od označenih razreda.

Unakrsna provjera (eng. Cross-validation) omogućuje provjeru rada modela nadziranog učenja. Unakrsna provjera se sastoji od izmjenjivanja postupka učenja na skupu

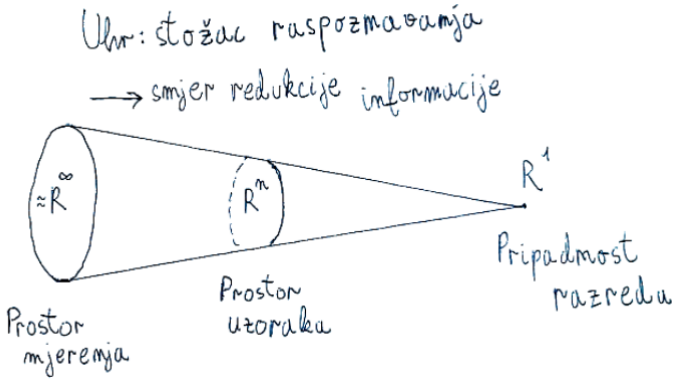
za treniranje i postupka provjeravanja na skupu za validaciju. Nakon nekoliko koraka procesa završna provjera provodi se na skupu za testiranje. Unakrsna provjera se koristi kako bi se rješili problemi poput šuma u podatcima i prenaučenosti. Ako se modelu da prevelika moć zaključivanja postoji mogućnost da nauči sve uzorke „napamet". Ako se to dogodi model savršeno predviđa sve uzorke koje je već vidio, ali nove uzorke predviđa sa smanjenom točnošću. U ovom slučaju može se reći da model ne generalizira. Bez generalizacije modeli su beskorisni u stvarnim uvjetima gdje se susreću s neviđenim primjerima.

5.2. Raspoznavanje uzoraka

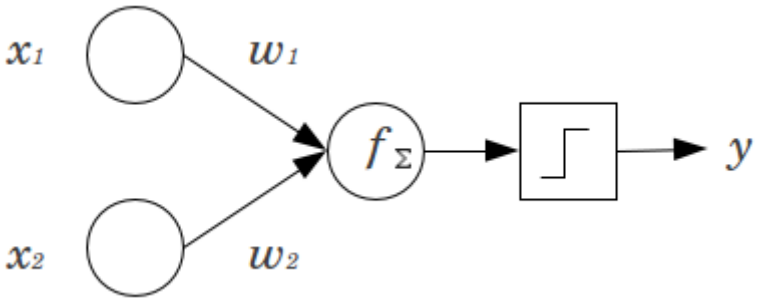
Strojno učenje nastalo je iz discipline raspoznavanja uzoraka [18]. Raspoznavanje uzoraka je disciplina računalne znanosti. Podskup je umjetne inteligencije i strojnog učenja, a bavi se metodama klasifikacije skupa uzoraka u proizvoljan broj razreda. Razred u kontekstu strojnog učenja i raspoznavanja uzoraka označava skup logički povezanih uzoraka i koristit će se u nastavku rada u tom smislu.

Prvotno se računalom pokušavalo odjeliti uzorke u neki skup razreda. Ljudi uče razlike između više uzoraka intuitivno i vrše redukciju informacije. Model redukcije naziva se Uhrov stožac raspoznavanja prikazan na slici 5.1. Slika pokazuje kako se redukcijom dimenzionalnosti može postići klasifikacija. Model redukcije informacija Uhrovog stošca može se objasniti raspoznavanjem koje vrše ljudi. Ljudski mozak je sastavljen od niza neurona. Na svaki neuron povezan je proizvoljan broj dendrita, a ima samo jedan izlazni akson. Kada oko vidi neku sliku u prostoru mjerenja R^{∞} ono ju transformira u niz impulsa u prostoru uzoraka ili prostoru značajki R^{n}. Impulsi putuju optičkim živcem do mozga te se sinapsama prenose na neki broj neurona. Neuroni vrše transformaciju impulsa i zatim prosljeđuju impuls sljedećem nizu neurona. Nakon dovoljnog broja transformacija, u mozgu nastaje odluka pripada li dio slike nekome razredu u prostoru R^{1}.

Na isti način modeliran je jedan od prvih računalnih sustava za raspoznavanje uzoraka. Taj model raspoznavanja uzoraka je bio model perceptrona. Umjetni model neurona prima proizvoljan broj ulaza i daje jedan binarni izlaz. Izlaz označava pripada li uzorak ili ne pripada razredu. Model perceptrona s dva ulaza prikazan je na slici 5.2. Slika opisuje model koji na temelju vektora uzorka s dvije značajke vrši linearnu transformaciju $f(\mathbf{x}, \mathbf{w})$ te na temelju predznaka rezultata zaključuje pripada li uzorak razredu.



Slika 5.1: Uhrov stožac



Slika 5.2: Model perceptrona

5.2.1. Reprezentacija uzoraka

Kako bi neki računalni model raspoznavanja uzoraka, poput perceptrona, mogao raditi potrebno je napraviti reprezentaciju uzorka. U računalu je nemoguće raditi s beskonačno dimenzionalnom slikom iz prostora mjerenja ($\approx R^{\infty}$). Transformacijom iz prostora mjerenja u prostor uzoraka se bave razni senzori. Uzorci se mogu predočiti u računalu kao jednodimenzionalni stupac vektori u prostoru uzoraka R^n. Svaki stupac vektor je jednakih dimenzija. Dimenzija stupac vektora uzorka označava broj značajki koje je moguće reprezentirati u računalnom obliku, brojevi koji pišu u vektoru sami po sebi ništa ne znače, nego im promatrač daje značenje. Primjer je vektor koji na indeksu i ima upisan broj 1, ali promatrač zna da je konstruiran takav skup značajki da je na i-tom mjestu u svakom vektoru upisana binarna vrijednost o tome posjeduje li uzorak ili ne posjeduje neku relevantnu značajku. Zadatak računala je iz skupa podataka odrediti koje su značajke bitne, a koje ne i na temelju njih napraviti klasifikaciju, dok je zadatak promatrača procijeniti i shvatiti rezultate, odnosno dati stvarni razlog zašto je računalo klasificiralo uzorke na određeni način. Iz jednostavnih modela je trivijalno vi-

djeti koja transformacija je dovela do klasifikacije, dok kompleksni modeli zahtijevaju dug proces razmatranja, da bi se shvatili mehanizmi koji su doveli do odluke.

5.2.2. Matematičko objašnjenje modela perceptrona

U nastavku poglavlja vrijedit će sljedeća notacija. Matrica $\mathbf{X}$ sadrži sve vektore uzoraka iz skupa podataka. Vektor $\mathbf{b}$ sadrži *odmak* uzoraka. Razlog imena odmak saznat će se u poglavlju gradijentni spust. Vektor $\mathbf{w}$ sadrži parametre funkcije odluke. Traži se $\mathbf{w}$ takav da vrijedi izraz (5.2.1). Radi pojednostavljenja u vektor $\mathbf{w}$ se kodira i odmak b na zadnje mjesto vektora $\mathbf{w}$ odnosno na w_{n+1}. Vektor $\mathbf{x}$ je jedan uzorak i na zadnjem mjestu x_{n+1} se postavlja 1 . Zadnji član vektora $\mathbf{x}$ se množi sa zadnjim članom vektora $\mathbf{w}$ odnosno sa b pa se tako b može izostaviti iz formule koja se optimira.Vektor $\mathbf{y}$ sadrži niz nula i jedinica koje označuju pripadnost razredu za svaki uzorak. Za jedan vektor dobiva se pojednostavljena formula (5.2.2). Funkcija $sign(x)$ (5.2.3) sažima kodomenu funkcije u skup $\{0, 1\}$.

$$sign(\mathbf{w}^T \cdot \mathbf{X} + b) = y \tag{5.2.1}$$

$$sign(\mathbf{w}^T \cdot \mathbf{x}) = y \tag{5.2.2}$$

$$sign(x) = \begin{cases} x \geqslant 0 & \to 1 \\ x < 0 & \to 0 \end{cases} \tag{5.2.3}$$

Ovime je potpuno opisan način klasifikacije ulaznih uzoraka modelom perceptrona.

5.2.3. Gradijentni spust

Decizijska funkcija je neka funkcija koja odvaja prostor značajki tako da su uzorci koji pripadaju razredu uvijek s pozitivne strane funkcije, a uzorci koji ne pripadaju razredu s negativne strane. Izraz (5.2.1) je primjer decizijske funkcije. Linearne decizijske funkcije se mogu zamisliti kao funkcije udaljenosti točke od neke plohe koja se naziva decizijska granica. Ovisno o dimenzionalnosti prostora decizijska granica može biti pravac ako je dimenzija prostora $D = 2$, ravnina za $D = 3$ ili hiperravnina za $D > 3$. Funkcija $d(\mathbf{X}, \mathbf{w})$ je decizijska funkcija koja daje udaljenost nekog uzorka od decizijske plohe i predznakom određuje s koje strane plohe se uzorak nalazi. Zadatak klasifikacije je optimiranje parametara vektora $\mathbf{w}$ tako da funkcija $d(\mathbf{X}, \mathbf{w})$ daje najbolje odvajanje razreda u prostoru značajki.

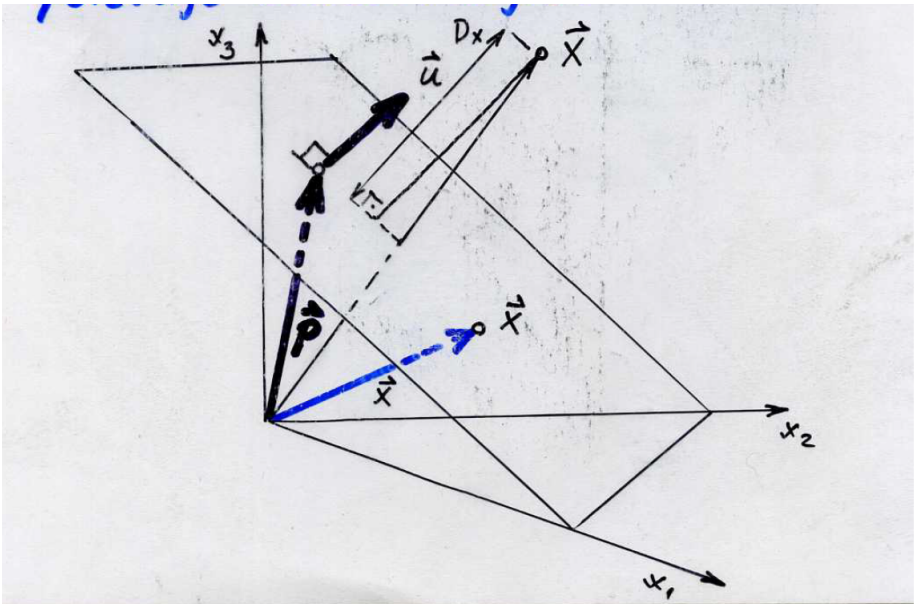
Da bi model mogao klasificirati skup podataka potrebno je naučiti parametre decizijske funkcije. Postupak učenja perceptrona naziva se algoritam gradijentnog spusta.

Na slici 5.3 je prikazana normala neke plohe u prostoru i vektori $\mathbf{p}$ i $\mathbf{x}$ točaka koje leže na plohi. Na trenutak se promatraju vektori bez pojednostavljenja i uzima se notacija $b = \mathbf{w}_{n+1}$. Poopće li se vektori iz slike 5.3 na R^N dobiva se zapis u formuli (5.2.4). Formula (5.2.5) opisuje decizijsku granicu modela perceptrona. Usporedbom tih dviju formula u jednadžbi (5.2.6) vidi se da je vektor parametara $\mathbf{w}$ uistinu normala decizijske granice. Stoga slijedi da je decizijska funkcija $d(\mathbf{x}, \mathbf{y})$ funkcija udaljenosti točke od plohe u prostoru.

$$\mathbf{u}^T \cdot (\mathbf{x} - \mathbf{p}) = 0 \Rightarrow \mathbf{u}^T \cdot \mathbf{x} = \mathbf{u}^T \cdot \mathbf{p} \tag{5.2.4}$$

$$\mathbf{w}^T \cdot \mathbf{x} + w_{n+1} = 0 \tag{5.2.5}$$

$$\mathbf{u} = \frac{\mathbf{w}}{||\mathbf{w}||} \quad , \quad \mathbf{u}^T \cdot \mathbf{p} = \frac{-w_{n+1}}{||\mathbf{w}||} \tag{5.2.6}$$



Slika 5.3: Ravnina iz prostora R^2 kao decizijska ploha, slika preuzeta iz [18]

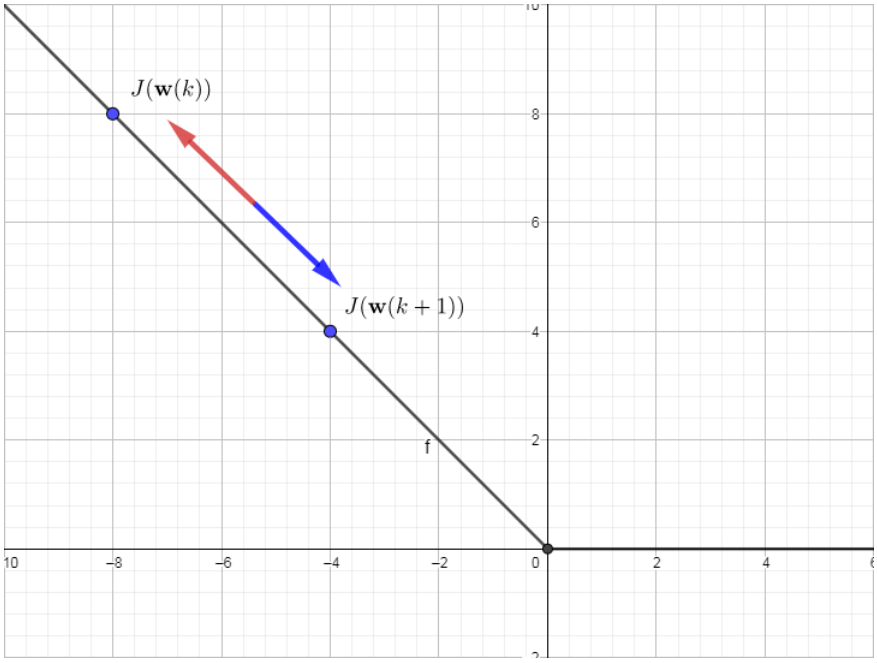
Optimirajući parametre $\mathbf{w}$ pomiče se ploha u prostoru koja odvaja prostor razreda. U primjeru klasifikacije uzorka, traži se takav $\mathbf{w}$ da za svaki $\mathbf{x}$ koji je element razreda vrijedi $\mathbf{w}^T \cdot \mathbf{x} > \mathbf{0}$. Potrebno je optimirati neku funkciju parametra $\mathbf{w}$ tako da postiže minimum kada je ispunjen ovaj uvjet. Za neki broj a uvjet $a > 0$ vrijedi kada je $|a| - a = 0$. Uvrštavanjem $\mathbf{w}^T \cdot \mathbf{x}$ umjesto a dobiva se izraz (5.2.7). Funkcija parametara i uzoraka klasifikacije koja postiže minimum kada je uvjet odlučivanja zadovoljen se naziva *kriterijska funkcija*.

Pošto su razredi označeni s 1 i 0 izraz (5.2.7) postiže minimum upravo kada je uvjet zadovoljen. Derivacijom izraza (5.2.7) po parametru $\mathbf{w}$ dobiva se gradijent koji

pokazuje smjer rasta funkcije. Pomakom u smjeru suprotnom od gradijenta funkcija se pomiče prema lokalnom minimumu za sve uzorke.

Iterativnim pomicanjem parametara u negativnom smjeru gradijenta dolazi se do globalnog minimuma funkcije (5.2.7) i najboljeg mogućeg razdvajanja u prostoru značajki. Slika 5.4 prikazuje korak k algoritma gradijentnog spusta primjenjenog na funkciju $J(\mathbf{w}, \mathbf{x})$, gdje $\mathbf{w}(k)$ označava vrijednost vektora parametara u k-tom koraku. Plava strelica pokazuje pomak, a crvena gradijent.

$$J(\mathbf{w}, \mathbf{x}) = |\mathbf{w}^T \cdot \mathbf{x}| - \mathbf{w}^T \cdot \mathbf{x} \tag{5.2.7}$$



Slika 5.4: Gradijentni spust funkcije $J(\mathbf{x}, \mathbf{w})$

5.3. Neuronske mreže s više potpuno povezanih slojeva

Neuronske mreže s više potpuno povezanih slojeva su logična nadogradnja modela perceptrona, s nekoliko izmjena. Više se ne promatra kriterijska funkcija nego funkcija gubitka. Funkcija gubitka određuje kolika je greška predviđanja. Jedna od najpopularnijih funkcija gubitka je *gubitak unakrsne entropije* koji definira stopu preklapanja distribucije točnih oznaka razreda $\mathbf{y}'$ i predviđenih oznaka $\mathbf{y}$. Gubitak unakrsne entropije dan je izrazom (5.3.1). Više se ne traži decizijska ploha niti se sustav optimira posredno kriterijskom funkcijom nego se direktno traži optimum funkcije gubitka. Funkcija $sign(x)$ ima gradijent koji uvijek iznosi 0 jer uvijek ima konstantnu

vrijednost osim u 0 gdje je gradijent nedefiniran. Zbog ovoga se ne može koristiti u funkcijama koje se optimiraju. Umjesto $sign(x)$ funkcije upotrebljava se niz funkcija poput $softmax(x)$ (5.3.2), $tanh(x)$ (5.3.3) ili $ReLU(x)$ (5.3.4). Ove funkcije se zovu aktivacijske funkcije. Funkcija $softmax(x)$ i $tanh(x)$ imaju problem nestajućeg gradijenta. Nestajući gradijent nastaje jer funkcije asimptotski u oba smjera teže konstantnoj vrijednosti te im je gradijent tada 0. Ove funkcije imaju gradijent različit od 0 samo u malom pojasu oko ishodišta. Funkcija $ReLU(x)$ nema problem nestajućeg gradijenta. Funkcija daje izlaz 0 kada je ulaz manji od 0, a kada je ulaz veći od nula samo ga propušta. Zbog ovoga je gradijent funkcije kada je ulaz veći od 0 uvijek 1.

Neuroni se slažu u sloj i nekoliko slojeva se slaže u slijed, tako da između svaka dva susjedna sloja postoji potpuna povezanost, odnosno svaki neuron je povezan sa svakim iz susjednog sloja.

$$\mathcal{L}(x) = \sum_{\mathbf{x}} \mathbf{y} \cdot log(\mathbf{y}') \tag{5.3.1}$$

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{k=0} e^{x_k}} \tag{5.3.2}$$

$$tanh(x_i) = \frac{e^{x_i} - e^{-x_i}}{e^{x_i} + e^{-x_i}} \tag{5.3.3}$$

$$ReLU(x_i) = max(0, x_i) \tag{5.3.4}$$

Višeslojne mreže su primjenjive na većem skupu problema od perceptrona jer uvode nelinearnu ovisnost između varijabli. Svaki sloj neurona ima svoju matricu parametara $\mathbf{W}$ koja je nastala slaganjem svih vektora $\mathbf{w}$ neurona u nizu. Ovaj korak se radi kako bi se transformacija u jednom sloju mogla proračunati u jednom koraku jednadžbom (5.3.5) u kojoj matrica $\mathbf{S}$ sadrži sve ulaze u sloj, a vektor $\mathbf{h}$ sadrži sve izračunate izlaze sloja.

$$\mathbf{h} = \mathbf{W}^T \cdot \mathbf{S} + \mathbf{b} \tag{5.3.5}$$

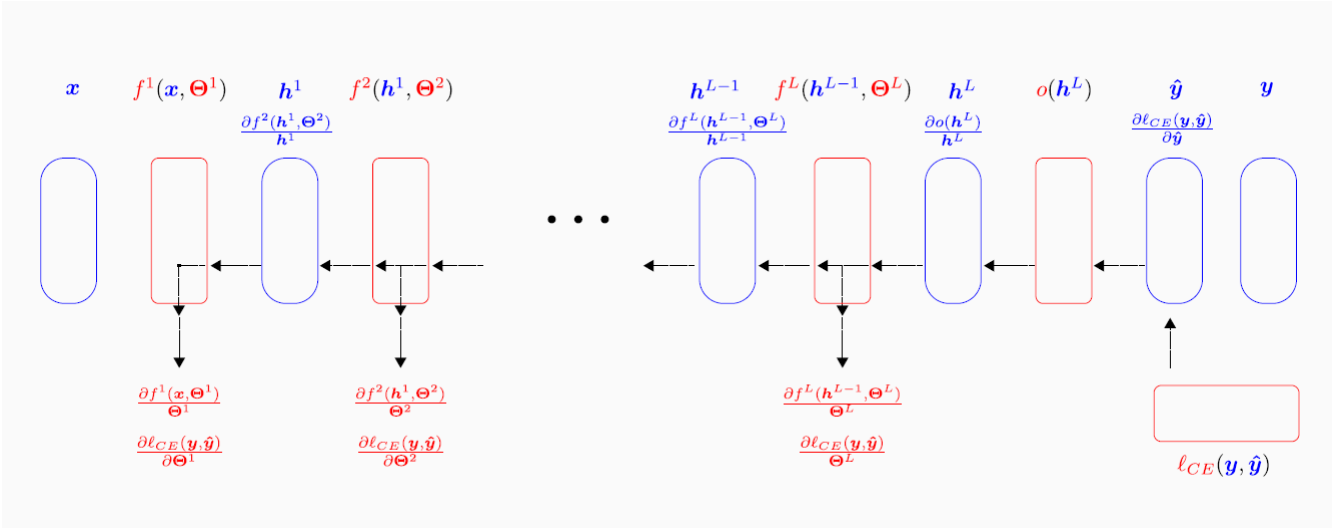
5.4. Algoritam propagacije pogreške unatrag

Neuronske mreže s više slojeva uče algoritmom propagacije pogreške unatrag (eng. *backpropagation*). Ovaj algoritam je često korišten u svim granama umjetne inteligencije. Algoritam propagacije pogreške unatrag je nadogradnja algoritma gradijentnog spusta generaliziranog na više slojeva. Izlaz neuronske mreže se može matematički

zapisati kao niz transformacija matrica. Svaka transformacija također može biti provedena kroz aktivacijsku ili prijenosnu funkciju. Izlaz neuronske mreže je niz ugniježđenih afinih transformacija matrica provedenih kroz aktivacijske funkcije. Koristeći pravilo ulančavanja derivirati se može svaka dobro definirana funkcija.

Algoritam propagacije pogreške unatrag provodi gradijentni spust u svakom neuronu lokalno počevši od izlaza mreže prema početku. Svi neuroni u jednom sloju se optimiraju u isto vrijeme. Dva različita sloja se ugođuju slijedno, jer gradijent u sloju ovisi o gradijentu svih sljedećih slojeva.

Algoritam se sastoji od prolaza unaprijed koji računa izlaze svih neurona mreže te funkciju gubitka mreže. Prolaz unatrag se sastoji od propagacije gradijenta gubitka unatrag kroz slojeve mreže te osvježavanja težina u slojevima u smjeru negativnog gradijenta funkcije gubitka. Vizualizacija algoritma može se vidjeti na slici 5.5. Na slici $f(\mathbf{x}, \theta)$ predstavlja funkciju sloja za uzorak $\mathbf{x}$ i parametre sloja θ dok h predstavlja aktivaciju sloja, a $\mathcal{L}$ predstavlja funkciju gubitka modela. Plavi čvorovi predstavljaju neuronske slojeve, a crveni čvorovi predstavljaju aktivacijske funkcije. Nakon što je izračunat izlaz mreže računa se gubitak $\mathcal{L}$. Ispod čvorova pišu gradijenti koje čvor izbacuje radi ugađanja težina, a to su gradijenti funkcije neuronskog sloja po parametrima. Iznad čvorova pišu gradijenti koje čvor šalje unatrag prethodnim slojevima, a to su gradijenti po ulaznim aktivacijama. Ovakvim prolazom može se izračunati cjelokupni gradijent gubitka po svim parametrima neuronske mreže.



Slika 5.5: Vizualizacija propagacije pogreške unatrag, slika preuzeta iz [10]

5.5. Konvolucijske mreže

Duboko učenje se zasniva na učenju dijelova uzorka i spajanja više odluka u jedinstvenu klasifikacijsku odluku [10]. Duboko učenje uči najbolju reprezentaciju uzorka.

Na primjer, konvolucijska mreža za prepoznavanje ljudskoga lica radi na principu traženja glavnih odlika lica. Konvolucijska mreža prvo traži oči, uši, nos i slične objekte u slici, koji su svi opet sastavljeni od drugačijeg spleta značajki, koje onda spaja u novi niz značajki na temelju kojeg odlučuje o pripadnosti slike skupu lica.

U problemu dekompilacije pretpostavka je da svaki razred ima niz instrukcija na početku i kraju koji je jedinstven za njega. Pristupom dubokog učenja model bi trebao moći naučiti uzorke početaka i krajeva razreda.

Konvolucija je jedna od metoda dubokog učenja. Prvi put su je definirali LeCun et al. u [14]. Kao što ime sugerira konvolucijske mreže koriste matematičku operaciju konvolucije. Konvolucija je dana izrazom (5.5.1).

$$conv(f(x), g(x)) = (f * g)(t) = \int_0^t f(t-\tau) \cdot g(\tau) d\tau \tag{5.5.1}$$

U matematičkom smislu konvolucija je transformacija funkcije $g(\tau)$. Funkcija $f(-\tau)$ je u ovom kontekstu filtar koji kliže cijelom domenom funkcije i množi funkciju $g(\tau)$. Konvolucijom se računa klizeća težinska suma $g(\tau)$, gdje je težinska funkcija $f(-\tau)$.

Konvolucijska mreža je bilo koja neuronska mreža koja u barem jednom od svojih slojeva umjesto operacije matričnog množenja koristi operaciju konvolucije, ali ne u strogom matematičkom smislu. Za dvije matrice $\mathbf{A}$ i $\mathbf{B}$ reda $m \times n$ i $n \times p$ operacija matričnog množenja $\mathbf{C} = \mathbf{A} \times \mathbf{B}$ je definirana izrazom (5.5.2). Operacija konvolucije $\mathbf{C} = conv(\mathbf{A}, \mathbf{B})$ je definirana izrazom (5.5.3), izostavljanjem sumacije formula se znatno računski pojednostavljuje. Zanimljivo je da za konvoluciju matrice ne moraju biti istog unutarnjeg reda, ali matrica $\mathbf{B}$ mora biti manja ili jednaka matrici $\mathbf{A}$. Ako je matrica $\mathbf{B}$ manja onda se formula provodi za svaku podmatricu matrice $\mathbf{A}$ koja je istog reda kao i matrica $\mathbf{B}$

$$\mathbf{A} \times \mathbf{B} = \mathbf{C} \quad \Leftarrow \quad c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} \cdot b_{k,j} \tag{5.5.2}$$

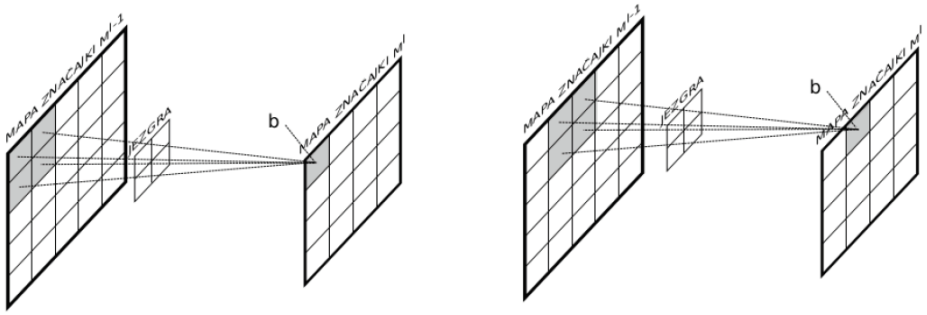
$$conv(\mathbf{A}, \mathbf{B}) = \mathbf{C} \quad \Leftarrow \quad c_{i,j} = a_{i,j} \cdot b_{i,j} \tag{5.5.3}$$

Rijetka interakcija, dijeljenje parametara i ekvivarijantna reprezentacija su tri motiva za korištenje konvolucijskih mreža [12]. Tradicionalne neuronske mreže koriste

velike matrice parametara $\mathbf{W}$ koje opisuju interakciju svakog ulaza sa svakim neuronom zasebno, ovakve matrice mogu postati memorijski prezahtjevne.

Konvolucijske mreže koriste rijetku interakciju. U literaturi se još može naći pod imenom rijetka povezanost ili rijetke težine. Matrica $\mathbf{W}$ u kontekstu konvolucije je jezgra ili filtar odnosno funkcija koja prolazi stacionarnom funkcijom u svakom vremenskom intervalu. Jezgra je nekoliko dimenzija manja od ulaznog podatka, a opisuje interakciju u nekom dijelu uzorka. Primjer zašto su jezgre jako korisne su slike. Slika može imati nekoliko milijuna piksela, dok je jezgra od nekoliko desetaka do nekoliko stotina piksela dovoljna za prepoznavanje neke bitne značajke slike poput ruba. Također ovo znači da računanje zahtjeva manje operacija, a prostorna složenost se značajno smanjuje.

Dijeljenje parametara je ostvareno zato što se operacija konvolucije s jezgrom provodi na svakoj poziciji ulaznog uzorka, kao što je prikazano na slici 5.6. Ovo svojstvo se još naziva svojstvo povezanih težina. Vrijednost težina konvolucije na nekoj poziciji ovisi o svim ostalim pozicijama u uzorku. Zbog ovoga nije nužno učiti težinu za svaku poziciju.



Slika 5.6: Vizualizacija konvolucije, slika preuzeta iz [10]

Ekvivarijantne funkcije su one funkcije koje imaju svojstvo da ako se ulaz promijeni, izlaz se jednako mijenja odnosno neovisne su o transformaciji. Općenito za ekvivarijantne funkcije vrijedi $f(g(x)) = g(f(x))$. Ovo je bitno jer se isti objekt može pronaći bilo gdje na slici, pa ima smisla jednom naučiti reprezentaciju npr. rubova i onda dijeliti tu funkciju sa svim neuronima. Konvolucija na žalost nije ekvivarijantna na transformacije poput rotacije i skaliranja, ali i ova svojstva se mogu postići uz neke preinake.

Konvolucijski sloj obično ima tri stadija. Stadij računanja konvolucije, stadij primjene nelinearnih aktivacijskih funkcija i stadij sažimanja. Kako se konvolucija provodi na svakoj poziciji ulaznog uzorka, a jezgra najčešće nije jednodimenzionalna, susjedni neuroni dobivaju veliku količinu redundantnih informacija. Sloj sažimanja

obično usrednjava informacije u nekom prostoru radi smanjenja dimenzionalnosti. Neke od funkcija sažimanja koje se koriste su $maxPool(\mathbf{x})$, $avragePool(\mathbf{x})$, $L^2normPool(\mathbf{x})$, $weightedAvrage(\mathbf{x})$. Sažimanje omogućuje konvolucijskom sloju da bude invarijantan na malu translaciju u ulaznom uzorku. Invarijantnost je jako bitno svojstvo kada je zanimljivija informacija postoji li neka značajka, nego gdje se nalazi.

Duboke mreže obično imaju prednost što se ne mora vršiti proces ekstrakcije značajki. Ekstrakcija značajki je pretproces koji se provodi prije metoda strojnog učenja koji iz uzorka izvlači bitne značajke i relacije koje će biti bitne za odluku klasifikacije. Ovo je često težak zadatak jer nije odmah očito koje informacije su bitne za neki problem. Razvoj dobrog modela ekstrakcije značajki zahtjeva puno ekspertnog domenskog znanja i duge cikluse razvoja i testiranja [9]. Duboke mreže će same provesti biranje bitnih značajki nad uzorkom i time znatno olakšati posao razvitka sustava.

Za potrebe učenja konvolucija ne unosi nikakve probleme. Ako konvoluciju promatramo kao množenje pojedinih ćelija matrica onda je gradijent po parametru skup ćelija, a gradijent po ulazu parametar. Treba samo paziti da se gradijent primijenjuje samo na one podatke koji su korišteni odnosno koje je sloj sažimanja odabrao, ako se koristi na primjer $maxPool(x)$. $maxPool(x)$ uzima samo jednu od vrijednosti cijelog filtra, pa se samo ti parametri mogu učiti.

5.6. Optimizacijski algoritam ADAM

Za optimizacijski postupak umjesto minimizacije funkcije gubitka algoritmom propagacije greške unazad odabran je nešto moderniji optimizator. `ADAM` (eng. *Adaptive Moment Estimation*)[13] je algoritam optimizacije koji koristi gradijente prvog reda. Temelji se na procjeni momenata prvog i drugog reda, odnosno srednje vrijednosti i varijance. Osim što provodi gradijentni spust također za svaki parametar čuva uprosječeni kvadrat gradijenta i uprosječeni gradijent. U koraku učenja gradijent se skalira dijeljenjem s korijenom kvadrata akumuliranog gradijenta. Parametri koji su imali velike vrijednosti u nekoliko iteracija neće se odmah jako povećati, nego tek kada se napuni propusnica kvadratnog gradijenta. `ADAM` postiže jako dobre rezultate bez puno namještanja, pa je dobar izbor za optimizacijski postupak. Algoritam je robustan i radi na širokom skupu parametara.

5.7. TensorFlow

`TensorFlow`[2] je Google-ov radni okvir za strojno učenje koji radi na različitim platformama. Koristi grafove toka podataka da bi reprezentirao računanje, zajedničko stanje i operacije koje mijenjaju stanja. Modul mapira čvorove grafa na različite uređaje unutar okoline poput centralne procesorske jedinice, grafičke procesorske jedinice i posebno dizajniranih procesorskih jedinica za proračune s tenzorima. Također sadrži implementacije svih važnijih metoda strojnog učenja i omogućuje jednostavniji pristup i konstrukciju modela za eksperimente. Zbog svojstva otvorenog koda i pristupačnosti postao je širom korišten za razne potrebe. U ovom radu svi modeli strojnog i dubokog učenja implementirani su korištenjem `TensorFlow` Python modula.

5.8. Google colab

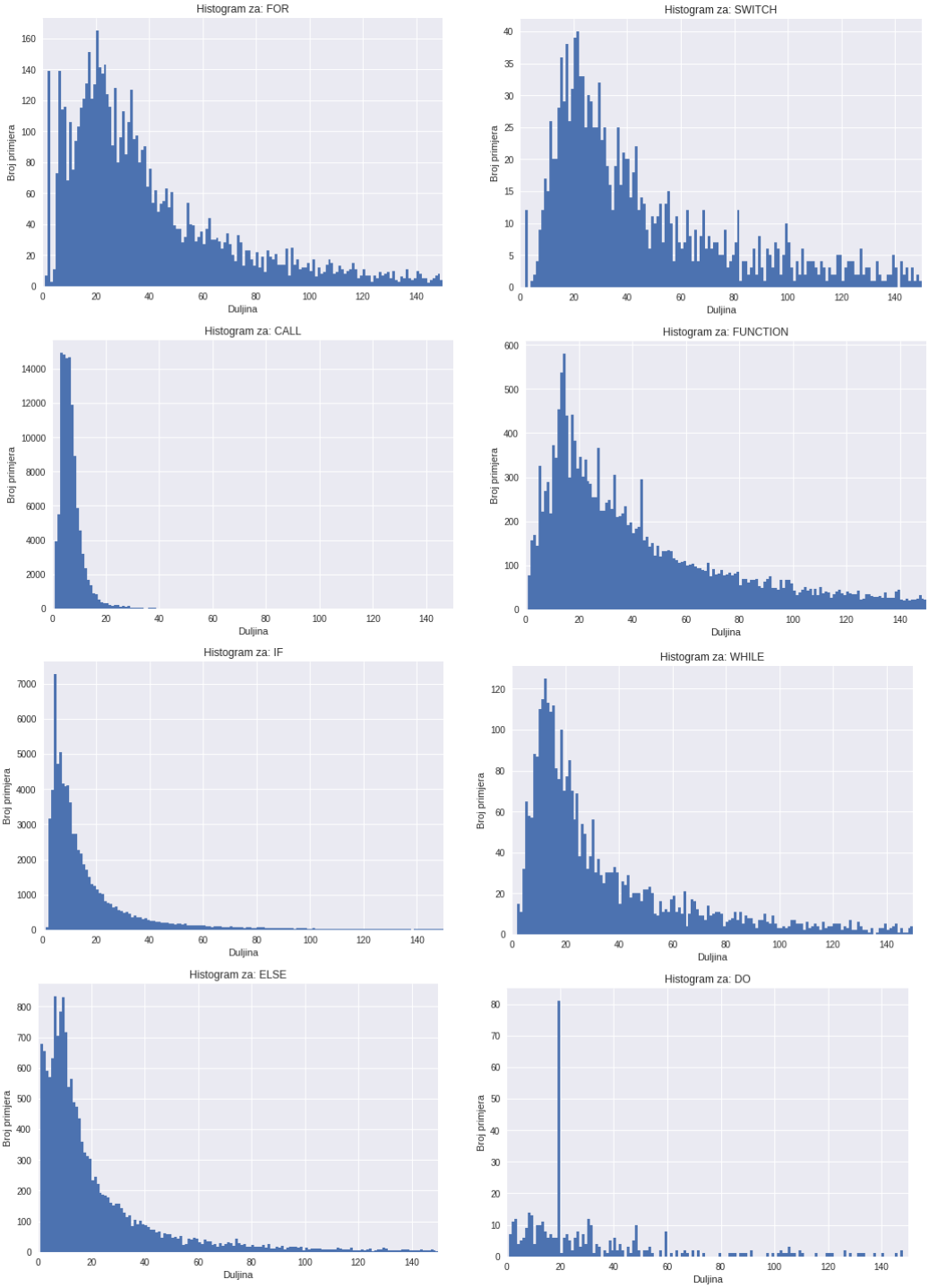
Google colab je Google-ova platforma koja pruža korisnicima besplatno korištenje relativno jakih računala. Ovo je jednostavan način da se iskoristi velika procesna moć Google-ovih računala. Dostupne su *Nvidia Tesla K80* grafičke kartice s 24 gigabajta video radne memorije što je više nego dovoljno za potrebe ovog problema. Računala također imaju dva *Intel(R) Xeon(R) CPU @ 2.30GHz* i 12 gigabajta radne memorije. Računala se daju na korištenje u sesijama od 6 sati, ali broj sesija koje korisnik može zatražiti nije ograničen. Ove specifikacije bi trebale biti dovoljne za bilo koji problem ove razine.

6. Eksperiment

U ovom poglavlju će biti razmotren eksperiment korištenja metoda strojnog učenja nad označenim objektnim kodom programa pisanih u programskom jeziku C. Kodove svih navedenih eksperimenata može se pronaći na `git` repozitoriju rada [25].

6.1. Skup podataka

Skup podataka za treniranje sastoji se od označenog objektnog koda programa `git`, `vim`, `curl` i `binutils` prevedenih programom `gcc`. Histogrami koji opisuju skup podataka mogu se vidjeti na slici 6.1. Histogrami pokazuju distribuciju broja instrukcija koje pripadaju jednom primjeru razreda. Najviši vrhovi uvijek se nalaze ispod 25 instrukcija. Sekvenca duljine do 20 znakova stoga bi trebala pokriti većinu niza instrukcija i uhvatiti sve najbitnije uzorke. Razredi `IF` i `ELSE` imaju znatno kraću duljinu, ali uzimajući u obzir domensko znanje programiranja da `ELSE` uvijek dolazi u paru sa `IF` i njihova duljina doseže oko 20 instrukcija. `CALL` ima znatno kraću duljinu, ali pretpostavlja se da uzimanje više instrukcija neće naštetiti točnosti modela. Histogram za razred `DO` ima oblik češlja i ovakvog je oblika jer za taj razred postoji najmanje primjera. Svi histogrami su nagnuti na desno zbog ograničenja da duljina niza instrukcija ne može biti manja od 0. U tablici 6.1 je zapisan broj pojavljivanja pojedinog razreda i ukupan broj instrukcija. Zbog velike razlike u brojevima razreda se u algoritmu treniranja modela mora obratiti pozornost na obuhvaćanje svih razreda podjednako.



Slika 6.1: Histogrami za klasifikacijske razrede

FOR	6704
CALL	113136
IF	77224
ELSE	16590
WHILE	3499
DO	475
SWITCH	1733
FUNCTION	21548
Broj instrukcija	1829678

Tablica 6.1: Broj uzoraka po razredu i ukupan broj instrukcija

6.2. Model i rezultati

Konstruiran je konvolucijski model koji gradi mrežu od niza konvolucijskih slojeva, pa zatim niza potpuno povezanih slojeva ovisno o ulaznoj konfiguraciji. Kod modela se može vidjeti na GitHub repozitoriju rada [25]. Konfigurabilna konstrukcija modela omogućuje testiranje većeg broja različitih modela. Konvolucija se provodi nad uzorcima jednakih dimenzija pa je potrebno odrediti širinu ulaznog vokabulara i duljinu sekvence. Ovaj pristup se ne temelji na jezičnom procesiranju mnemoničkih instrukcija nego na direktnom učenju iz okteta zapisa instrukcije. Stručnjaci preporučuju uzimanje prva 4 okteta svake instrukcije za provođenje učenja, jer će sve bitne informacije o instrukciji vjerojatno biti sadržane u tom dijelu [9]. Stručnjaci također preporučuju kodiranje instrukcija vektorima indikatorskih varijabli (eng. *one-hot vector*) koji su se pokazali kao dobar izbor za metode strojnog učenja na disasembliranim podatcima u svrhu traženja zloćudnog koda [9]. Vektor indikatorskih varijabli je rijetka reprezentacija dobra za korištenje s modelima gdje je potrebno napraviti veliku distinkciju između vektora uzoraka, što cjelobrojno kodiranje ne može postići. Postoji veća razlika između dva vektora ako su oni kodirani sa $(0, 1)$ i $(1, 0)$ umjesto 0 i 1. Potrebno je osigurati da dvije instrukcije koje mogu imati sličan kod budu više udaljene u prostoru značajki.

Hiperparametri modela strojnog učenja su oni parametri koji se postavljaju prije učenja i utječu na rezultate, ali se ne uče. Duljina sekvence je hiperparametar algoritma učenja, ali širina nije. Širina svakog uzorka ograničena je duljinom vektora indikatorskih varijabli. Ako se instrukcije skrate na 4 okteta to znači da su skraćene na 8 heksadekadskih znamenki. Ako se svaka znamenka kodira kao vektor indikatorskih varijabli njihova konkatenacija ima točno 128 članova jer svaki vektor indikatorskih varijabli heksadekadskog broja ima duljinu 16. Zbog ovoga je širina sekvence uvijek

128. Uz duljinu sekvence ostali hiperparametri algoritma uključuju veličinu konvolucijske jezgre, broj jezgri, širinu potpuno povezanog sloja, broj epoha učenja i mjesta uzorkovanja sekvence. Zbog pretpostavke da razredi imaju jedinstven niz na početku i na kraju niza instrukcija koji mu pripada testirano je utječe li uzimanje instrukcija za učenje samo s početka niza na problem klasifikacije.

Hiperparametri su određeni postupkom k-struke unakrsne provjere. Uzevši podskup objektnih datoteka iz programa `git` radi smanjenja vremena računanja proveo se postupak k-struke unakrsne provjere, gdje je $k = 26$. Za 26 kombinacija hiperparametara model se trenirao na skupu za treniranje. Skup za treniranje je za svaku kombinaciju uzet kao 80% cijelog skupa te zatim provjerio na skupu za validaciju. Dijeljenje skupa se radilo za svaku kombinaciju. U tablici 6.2 nalaze se postotci točnosti naučenih modela u 10 epoha učenja. Iz tablice je prema postotcima točnosti vidljivo da duljina sekvence i mjesto uzorkovanja instrukcija ne utječe na preciznost modela ako mu se da dovoljan kapacitet potpuno povezanog sloja. Veličina slojeva ima dramatičan utjecaj na točnost. Ovo je dobar pokazatelj da u podatcima postoje uzorci koje mreža može pronaći, no zahtjeva veći kapacitet da bi savršeno odvojila razrede. Također jedan bitan pokazatelj je da točnost klasifikacije ne varira jako između skupova za treniranje i validaciju. Ovo znači da je skup podataka dobro konstruiran i da model uistinu uči koje su instrukcije indikatori razreda, a ne da samo uči "napamet" uzorke zbog prevelikog kapaciteta. Modeli velikog kapaciteta imaju tendenciju prenaučiti skup podataka odnosno zapamtiti sve primjere. Ovo za posljedicu ima jako lošu generalizaciju na neviđenim podatcima.

Zatim se empirijski izabrao završni model. Završni model se sastoji od konvolucijskog sloja sa 16 jezgri veličine 128x6, bez sloja sažimanja, 2 potpuno povezana sloja veličine 10 i 8 te izlazni sloj s aktivacijskom funkcijom $softmax(x)$, a svi unutarnji slojevi imaju aktivacijsku funkciju $ReLU(\mathbf{x})$. Točnost se računa kao postotak točno klasificiranih primjera od ukupnog broja primjera. Točnost se na isti način računa i za pojedini razred. Na cijelom skupu podataka koji sadrži označene instrukcije programa `git`, `curl`, `binutils` i `vim`, model dostiže točnost od oko 81%. U nastavku su modeli trenirani na skupu `git`, `binutils` i `vim`, dok je `curl` skup korišten za validaciju. Odnos količine uzoraka skupova za treniranje i validaciju je otprilike 8 naprema 2. Na slici 6.2 se vidi kretanje točnosti kroz epohe. Model najbolje generalizira nakon 10 epoha učenja, a točnost na skupu za validaciju nakon toga trenutka više ne raste, odnosno oscilira oko te vrijednosti bez pravog pomaka. Zbog ovog je za treniranje svih modela odabrano 10 epoha.

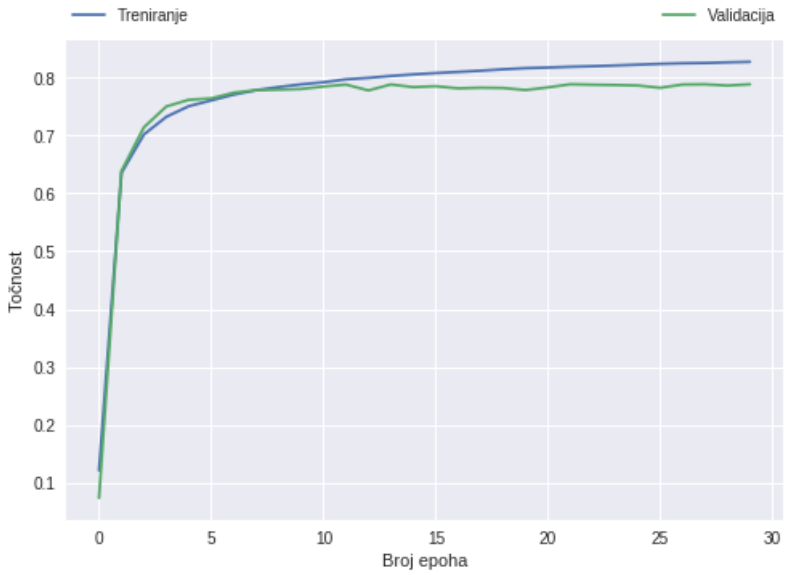
Skup za treniranje

Samo početne instrukcije					Početne i završne instrukcije				
Duljina sekvence: 6									
conv \ fc	2	5	8	11	conv \ fc	2	5	8	11
3	58.59	67.05	70.69	73.31	3	40.57	71.94	73.73	75.38
6	46.03	73.24	74.72	76.57	6	48.04	70.35	74.56	78.12
9	26.29	73.46	76.14	79.02	9	46.13	75.77	78.33	80.07
Duljina sekvence: 8									
conv \ fc	2	5	8	11	conv \ fc	2	5	8	11
3	49.45	68.68	72.05	71.50	3	45.90	68.32	73.97	75.55
6	48.92	71.15	75.23	77.00	6	56.37	73.53	75.34	78.40
9	61.22	75.80	77.99	78.50	9	54.31	75.29	78.40	79.73
Duljina sekvence: 10									
conv \ fc	2	5	8	11	conv \ fc	2	5	8	11
3	38.61	64.58	72.52	72.47	3	52.38	68.20	74.04	72.70
6	53.24	72.85	75.09	75.98	6	58.05	71.86	77.11	78.04
9	45.21	74.19	76.57	78.68	9	50.14	77.66	78.97	79.53

Skup za testiranje

Samo početne instrukcije					Početne i završne instrukcije				
Duljina sekvence: 6									
conv \ fc	2	5	8	11	conv \ fc	2	5	8	11
3	58.22	66.85	70.61	73.41	3	40.44	71.63	74.02	75.56
6	45.76	72.96	74.69	76.02	6	48.17	70.55	74.66	77.94
9	26.45	73.36	76.19	79.11	9	46.31	75.79	78.25	80.20
Duljina sekvence: 8									
conv \ fc	2	5	8	11	conv \ fc	2	5	8	11
3	49.50	68.53	71.65	71.29	3	45.61	68.37	73.98	75.91
6	48.92	70.88	74.84	76.80	6	56.09	73.20	74.74	78.33
9	61.25	75.87	77.85	78.21	9	54.23	75.35	78.49	79.86
Duljina sekvence: 10									
conv \ fc	2	5	8	11	conv \ fc	2	5	8	11
3	38.51	63.96	72.73	72.41	3	52.17	67.89	74.04	73.02
6	53.13	72.85	74.84	75.99	6	57.97	71.45	77.13	78.26
9	45.23	74.04	76.57	78.39	9	50.12	77.57	78.49	79.46

Tablica 6.2: k-struka unakrsna provjera



Slika 6.2: Točnost na skupu za učenje i na skupu za validaciju kroz epohe

Kako bi se protumačio rad modela dublje od same točnosti potrebno je proučiti konfuzijsku matricu. Konfuzijska matrica sadrži podatke za računanje svih mjera klasifikacije. U stupcima se nalaze svi primjeri nekog razreda podijeljeni po razredima koji su im predviđeni. U redcima se nalaze svi primjeri klasificirani u neki razred odjeljeni po pravom razredu kojem pripadaju. Uzimajući ovo u obzir u matrici na dijagonali uvijek pišu pravi pozitivi. U nekom retku svi članovi osim dijagonalnog su lažni pozitivi. U nekom stupcu svi članovi osim dijagonalnog su lažni negativi. Svi članovi dijagonale su pravi negativi za svaki drugi razred. Ovime se mogu izračunati sve mjere klasifikacije.

Konfuzijska matrica za razrede je prikazana na tablici 6.3. Iz konfuzijske matrice se vidi da matrica teži dijagonalnoj matrici što znači da model može dobro odvojiti gotovo sve razrede. `ELSE` je razred koji je najlošije klasificiran. `ELSE` razred je specifičan jer je kraći u odnosu na ostale razrede i relativno je zamjenjiv s razredom `IF`. `ELSE` također nema mnogo jedinstvenih značajki koje ga određuju nego se samo pojavljuje kao još jedna instrukcija skoka nakon `IF`-a.

Kod `WHILE` i `FOR` razreda najčešća je greška zamjena ta dva razreda. `DO` razred ima najveću točnost, ali to je zato što se najmanje pojavljuje. Zbog svojstva učenja da se u svakoj mini grupi nalazi podjednak broj primjera svakog razreda, `DO` se najčešće pojavljivao, pa ga je model imao priliku prenaučiti. Također `DO` je poseban razred jer je pretpostavka da se njegov jedinstven niz instrukcija ne pojavljuje na početku nego na kraju razreda, pa je moguće zato dobro odvojen. `FUNCTION` se često zamjenjuje s `CALL` jer se specifičan uzorak za oba razreda sastoji od instrukcija za rukovanje sto-

gom odnosno memorijskih instrukcija. SWITCH je dobro odvojen razred jer njegov specifični uzorak ima mnogo instrukcija skoka te nije sličan niti jednom drugom razredu.

točno / predviđeno	FOR	CALL	IF	ELSE	WHILE	DO	SWITCH	FUNCTION
FOR	10.043	0.054	0.105	0.522	0.758	0.	0.034	0.091
CALL	0.079	9.642	1.179	1.619	0.133	0.074	0.014	0.408
IF	0.033	1.032	11.865	4.904	0.129	0.049	0.102	0.073
ELSE	0.357	0.612	2.130	3.776	0.361	0.	0.082	0.328
WHILE	0.831	0.163	0.112	0.232	10.210	0.	0.088	0.163
DO	0.126	0.068	0.148	0.340	0.	11.576	0.	0.030
SWITCH	0.072	0.0240	0.252	0.224	0.112	0.	11.448	0.052
FUNCTION	0.203	0.064	0.126	0.351	0.197	0.	0.020	12.178

Tablica 6.3: Konfuzijska matrica

6.3. Usporedba optimizacijskih razina

Zanimljivo je ispitati kako optimizacija utječe na model jer se u stvarnom svijetu programi optimiziraju ovisno o arhitekturi. Točnost klasifikacije u odnosu na razinu optimizacije prikazana je u tablici 6.4. Očito je iz tablice da se točnost klasifikacije smanjuje, što je i bilo očekivano. Razina optimizacije 1 gotovo da nema utjecaj na točnost, dok razina 2 drastično smanjuje točnost. U tablici 6.5 se nalazi konfuzijska matrica za razinu optimizacije 3. Problem sličnosti FOR i WHILE razreda dolazi do većeg izražaja.

Optimizacijska razina 2 uključuje optimizator petlji koji uzrokuje ovu pojavu. Gotovo svi ELSE primjeri nisu prepoznati nego se klasificiraju u IF, CALL i FUNCTION razrede. ELSE razred je najteže klasificirati jer nema poseban uzorak, a optimizacijom se gotovo gubi. SWITCH i DO razredi su još uvijek vjerojatno prenaučeni jer ih ima najmanje, a u mini grupu su stavljeni podjednak broj puta kao ostali razredi. FUNCTION razred isto ima smanjenu točnost. Kada model uči FUNCTION razred uči uzorak za postavljanje i čišćenje stoga na početku odnosno kraju razreda. Optimizator pretvara neke funkcije u `inline` funkcije koje nemaju ovaj uzorak pa ih se ne može pronaći.

Optimizacija uzrokuje gušći strojni kod jer briše sve redundantne instrukcije i optimizira za najveću funkcionalnost. Zbog gušćeg strojnog koda razredi su međusobno blisko u objektnom kodu. Ako neki razred često počinje odmah nakon drugog ta dva

razreda će biti teško odvojiva jer će model naučiti oba uzorka zajedno. Primjer su razredi `IF` i `ELSE`

Razina:	Točnost treniranja:	Točnost testiranja:
O0	77.693	76.385
O1	79.801	78.564
O2	59.511	58.016
O3	55.654	55.655

Tablica 6.4: Točnost klasifikacije optimizacijskih razina

točno / predviđeno	FOR	CALL	IF	ELSE	WHILE	DO	SWITCH	FUNCTION
FOR	6.130	0.230	1.335	1.239	4.414	0.000	0.000	1.082
CALL	0.168	7.182	1.594	2.089	0.166	0.085	0.036	2.562
IF	1.991	2.163	10.222	5.722	2.495	0.000	0.090	2.387
ELSE	0.255	0.172	0.327	0.403	0.140	0.000	0.081	0.338
WHILE	2.270	0.095	1.004	0.707	3.553	0.000	0.009	0.353
DO	0.028	0.137	0.123	0.066	0.009	11.051	0.009	0.137
SWITCH	0.180	0.057	0.425	0.277	0.068	0.000	10.919	0.345
FUNCTION	0.845	1.078	1.318	2.009	0.581	0.000	0.054	7.197

Tablica 6.5: Konfuzijska matrica za optimizacijski nivo 3

6.4. Klasifikacija cijele funkcije

Radi korištenja modela u stvarnim uvjetima također je napravljen test klasifikacije cijelih funkcija. Cilj klasifikacije cijele funkcije je označiti niz instrukcija oznakama poput onih u tablici 4.3. Prvi korak bio bi odvojiti niz instrukcija na one koje treba klasificirati u 8 definiranih klasa i na skup onih instrukcija koje ne pripadaju niti jednoj klasi. Kada bi se ovaj korak uspio odraditi bilo bi moguće s točnošću 81% pronaći sve razrede u funkciji. Na žalost podatci pokazuju da je ovo puno teži problem od klasifikacije i da konvolucijske mreže možda nisu najbolji izbor za ovaj dio zadatka.

Konstruiran je jednak model kao u prethodnom poglavlju, ali su sada svi razredi spojeni u jedan razred `razredi`, a sve ostale instrukcije stavljene u razred `ne razredi`. Problem je postao binaran i nastoje se odvojiti sve instrukcije koje je moguće klasificirati od onih koje ne pripadaju razredima. Također uvedeno je da je svaki niz od 16 instrukcija jedan uzorak uz dozvoljena preklapanja nizova. Ovo se dopustilo jer konvolucijske mreže ne mogu primiti proizvoljnu duljinu sekvence uzorka. Za

svaki takav niz traži se predikcija odnosno traže se instrukcije koje započinju sekvencu koja je početak nekog razreda. Sada se koristi druga metrika, odaziv, koja označava koliko je primjera dohvaćeno od ukupnog broja primjera. Ova metrika se koristi jer 90% nizova ne treba oznaku, odnosno ima oznaku `ne razred`, a ako model da svim nizovima oznaku `ne razred` postići će točnost od 90% što ne odražava stvarnu uspješnost modela.

Konvolucija može postići do 83% odaziv instrukcija koje ne treba klasificirati, i do 76% odaziv instrukcija koje treba klasificirati, vidljivo u tablici 6.6. Ovo nisu najbolji rezultati jer oko 13% skupa su one instrukcije koje treba klasificirati, a model može naći 76% tih instrukcija. Također 82% točnost nalaženja `ne razreda` znači da se oko 100,000 instrukcija označuje kao razred iako ne bi trebao. Ovaj broj je usporediv s brojem uzoraka za razrede koji je 250,909. Ovo znači da je oko 25% predikcija modela lažan pozitiv.

	Točnost treniranja:	Točnost testiranja:
Točnost	83.871	82.246
Odaziv razreda	62.500	76.063
Odaziv ne razreda	91.304	82.586

Tablica 6.6: Točnost klasifikacije optimizacijskih razina

Konvolucija vjerojatno nije najbolji izbor za rješavanje ovog problema. Konvolucija provjerava sekvence od određenog broja instrukcija što znači da se dvije susjedne sekvence razlikuju samo u jednoj instrukciji. Sekvenca koja slijedi ne sadrži prvu instrukciju prethodne sekvence i dodaje jednu instrukciju na kraj. Posljedica ovoga je to da su skup `razredi` i `ne razredi` teško odvojivi. Bolja opcija bi bila neka vrsta povratnih mreža. Povratne mreže se lakše nose s ovakvim problemima jer u trenutku klasifikacije funkcije znaju cijeli kontekst dosadašnje klasifikacije te prostornu udaljenost informacija jer koriste gustu reprezentaciju. Povratnom mrežom bi se mogli pronaći svi početci razreda, pa zatim te instrukcije klasificirati konvolucijskom mrežom koja dobro razdvaja razrede.

7. Zaključak

U ovom diplomskom radu obavljena su dva različita zadatka. Konstruiran je alat u programskom jeziku Python za automatsko označavanje disasembliranih instrukcija objektnog koda programa pisanih u programskom jeziku C. Alat prolazi kroz niz objektnih datoteka i čitajući njihove `ELF` i `DWARF` informacije gradi poseban zapis objektnog koda. Alat generira listu instrukcija s adresama, mnemoničkim zapisom, linijom i datotekom izvornog koda te oznakom pripadnosti svim razredima odnosno naredbama u čijem djelokrugu postoji.

Drugi zadatak je ispitivanje mogućnosti korištenja tog alata za konstrukciju skupa podataka iz kojeg bi bilo moguće naučiti model strojnog učenja kako dekompajlirati neki program. Za ovo su iskorištene duboke konvolucijske mreže. U eksperimentima je pokazano da konvolucijske mreže mogu dobro odvojiti skup od 8 razreda logičkih konstrukata izvornog koda. Model postiže točnost preko 80% na skupu za učenje i skupu za validaciju. Ispitano je kako točnost pada sa stupnjem optimizacije programa, te može li model naučiti koje dijelove treba klasificirati u razrede, a koje ne. Pokazano je da konvolucijske mreže nisu najbolji izbor za traženje gdje u kodu postoje razredi, ali ako bi neki drugi model riješio problem bile bi dobar izbor za klasifikaciju tih razreda.

Pokazano je da konvolucijske mreže mogu dobro odvojiti razrede, odnosno da postoje stvarni uzorci koje mreža može naučiti. U nastavku rada bilo bi potrebno konstruirati model koji može dobro naći gdje u kodu se uzorci nalaze. Moguća je upotreba `Single Shot Multi Box` [15] detektora ili alata `YOLO` [17]. Jednostavno objašnjenje je da ti alati postavljaju granice razreda u sekvence te zatim uče te granice.

Guste reprezentacije poput `word2vec` mogle bi se konkatenirati na dosadašnju reprezentaciju da se izluče informacije o sličnosti ili razlikama instrukcija [11].

LITERATURA

[1] Using the gnu compiler collection (gcc): Optimize options. URL https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[2] 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.

[3] Anton Kochkov, aoighost, Austin Hartzheim, David Tomaschik, DZ-ruyk, Grigory Rechistov, hdznrrd, Jeffrey Crowell, John, Judge Dredd, jvoisin, Kevin Grandemange, muzlightbeer, Peter C, sghctoma, SkUaTeR, TDKPS, Thanat0s, maijin, pancake. Radare2 book. https://legacy.gitbook.com/book/radare/radare2book/details, 2018.

[4] AV-TEST. Av-test – the independent it-security institute, 2017. URL https://www.av-test.org/en/statistics/malware/.

[5] Tiffany Bao, Johnathon Burket, Maverick Woo, Rafael Turner, i David Brumley. Byteweight: Learning to recognize functions in binary code. USENIX, 2014.

[6] Eli Bendersky. Pycparser. https://github.com/eliben/pycparser, 2018.

[7] Eli Bendersky. Pyelftools. https://github.com/eliben/pyelftools, 2018.

[8] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, i Zhenkai Liang. Neural nets can learn function type signatures from binaries. U *Proceedings of the 26th USENIX Conference on Security Symposium, Security*, svezak 17, 2017.

[9] Andrew Davis i Matt Wolff. Deep learning on disassembly data. *Black Hat, USA*, 2015.

[10] Siniša Šegvić. Duboko Učenje. Predavanja iz kolegija Duboko učenje, 2018. URL http://zemris.fer.hr/~ssegvic/du/.

[11] Yoav Goldberg i Omer Levy. word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*, 2014.

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

[13] Diederik P. Kingma i Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL http://arxiv.org/abs/1412.6980.

[14] Yann LeCun, Bernhard E Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne E Hubbard, i Lawrence D Jackel. Handwritten digit recognition with a back-propagation network. 1990.

[15] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, i Alexander C Berg. Ssd: Single shot multibox detector. U *European conference on computer vision*, stranice 21–37. Springer, 2016.

[16] Jan Šnajder i Bojana Dalbelo Bašić. Strojno učenje. Predavanja iz kolegija Strojno učenje, 2016. URL http://www.fer.unizg.hr/predmet/su.

[17] Joseph Redmon, Santosh Divvala, Ross Girshick, i Ali Farhadi. You only look once: Unified, real-time object detection. U *Proceedings of the IEEE conference on computer vision and pattern recognition*, stranice 779–788, 2016.

[18] Slobodan Ribarić. Raspoznavanje uzoraka. Predavanja iz kolegija Raspoznavanje uzoraka, 2016. URL http://www.fer.unizg.hr/predmet/rasuzo_a.

[19] Dennis M. Ritchie. The development of the c language. *SIGPLAN Not.*, 28(3): 201–208, Ožujak 1993. ISSN 0362-1340. doi: 10.1145/155360.155580. URL http://doi.acm.org/10.1145/155360.155580.

[20] Dennis M Ritchie, Brian W Kernighan, i Michael E Lesk. *The C programming language*. Prentice Hall Englewood Cliffs, 1988.

[21] Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, i Alexey Loginov. Evolving exact decompilation.

[22] TIOBE. Tiobe programming community index, 2018. URL https://www.tiobe.com/tiobe-index/. [Online; pristupljeno 1 Srpanj, 2018].

[23] *Tool Interface Standard (TIS), Portable Formats Specification*. TIS Committee, 1993. URL http://www.rcollins.org/intel.doc/Tools.html.

[24] Fran Varga. C bytecode analysis. https://github.com/fvarga94/C_bytecode_analysis, 2018.

[25] Fran Varga. C deep learning. https://github.com/fvarga94/C_deeplearning, 2018.

[26] Yegor Derevenets. Snowman, 2018. URL https://github.com/yegord/snowman.

Dekompajliranje aplikacija pisanih u programskom jeziku C/C++ potpomognuto metodama strojnog učenja

Sažetak

Svake godina raste količina zloćudnog koda. U borbi protiv te zabrinjavajuće statistike se troši puno truda i novca. Postoje alati za dekompajliranje koji pomažu stručnjacima za reverzno inženjerstvo u rekonstrukciji izvornog koda i proučavanju zloćudnog koda. Ti alati su ograničeni, a postupak analize je i dalje dug i naporan. Postavlja se pitanje može li se automatizirati dijelove ovog procesa. Strojno učenje je metoda koja se pokazala uspješnom u zadatcima identifikacije zloćudnog koda i rekonstrukcije nekih dijelova koda. U ovom radu pokazano je kako iskoristiti duboke konvolucijske mreže za odjeljivanje razreda naredbi kontrole toka, odlučivanja i poziva funkcija. Konvolucijske mreže mogu s preko 81% točnošću predvidjeti koja naredba obuhvaća neki niz instrukcija, a mogu pronaći početke tih nizova s odazivom od preko 76% za instrukcije koje jesu početci nizova.

Ključne riječi: C/C++, dekompajliranje, strojno učenje, konvolucijske mreže, ELF format, DWARF format, disasembliranje, reverzno inženjerstvo)

Decompilation of aplications written in C/C++ programming language based on machine learning methods

Abstract

Every year the amount of malware increases. In the fight against these worrying statistics a lot of money and effort is spent. There are decompilation tools that help reverse engineering experts to reconstruct source code and analyze the malware. These tools are stil limited in their capabilities and the process is tedious and requires a lot of man hours. The question arises: can the process be automated. Machine learning has been shown to be successful in malware detection and partial code reconstruction. In this paper it is shown how to use deep convolution networks to guess the control flow statements, decision making statements, function starts and calls. Convolution networks can predict the correct class with a 81% accuracy rate. They can also find the sequences with the recall rate of 72%.

Keywords: C/C++, decompilation, machine learning, convolution networks, ELF format, DWARF format, code disassembly, reverse engineering