Eötvös Loránd Tudományegyetem

Informatikai Kar

Programozáselmélet és Szoftvertechnológiai Tanszék

Kódvisszafejtés mélyhálókkal

Témavezető:
Dr. Várkonyi Teréz Anna
egyetemi adjunktus

Szerző:
Csertán András
programtervező informatikus BSc

Budapest, 2022

EÖTVÖS LORÁND TUDOMÁNYEGYETEM
INFORMATIKAI KAR

SZAKDOLGOZAT TÉMABEJELENTŐ

Hallgató adatai:
Név: Csertán András
Neptun kód: NDLG3A

Képzési adatok:
Szak: programtervező informatikus, alapképzés (BA/BSc/BProf)
Tagozat : Nappali

Belső témavezetővel rendelkezem

Témavezető neve: *Dr. Várkonyi Teréz Anna*
munkahelyének neve, tanszéke: ***ELTE-IK, Programozáselmélet és Szoftvertechnológia Tanszék***
munkahelyének címe: ***1117, Budapest, Pázmány Péter sétány 1/C.***
beosztás és iskolai végzettsége: ***adjunktus, PhD***

A szakdolgozat címe: Kódvisszafejtés mélyhálókkal

A szakdolgozat témája:
(A témavezetővel konzultálva adja meg 1/2 - 1 oldal terjedelemben szakdolgozat témájának leírását)

A kódvisszafejtő program feladata egy futtatható fájlból előállítani az annak megfelelő, magasszintű forrásfájlt. Működése a fordítóprogramok működésével ellentétes, amik egy magasszintű forrásfájlból állítják elő a futtatható állományt. Segítségével megismerhetjük a program valódi működését, ezáltál többek között kártékony szoftverek detektálására is használhatjuk.
Szakdolgozatom célja egy olyan alkalmazás készítése, ami a fenti problémát mély neurális hálók segítségével oldja meg. A program bemenete az alacsony szintű Assembly kód, kimenete pedig a magas szintű, egy átlagos programozó által is értelmezhető C nyelvű kód. A hatékonyság növelése érdekében az Assembly kód először szegmentálásra kerül, így blokkokat kapunk, amik nagyjából egy C utasításnak felelnek meg. A szegmentálást és a megfelelő C utasítás meghatározását egy-egy neurális háló fogja végezni. Ezek után a kapott C utasításokat összefűzve, valamint az esetleges hibákat korrigálva megkapjuk a várt kimenetet.
A felhasználónak lehetősége lesz saját adatokkal a háló tanítására is, valamint különböző példákon keresztül a működés kipróbálására. Az implementáció Python nyelven fog történni.

Budapest, 2021. 11. 30.

Tartalomjegyzék

1. fejezet

Bevezetés

A fordítóprogramok feladata a magas szintű programkód átalakítása a számítógép számára értelmezhető formára. Céljuk, hogy a programozó sokkal magasabb absztrakciós szinten fejezhesse ki szándékát, ezáltal megkönnyítve a szoftverfejlesztés folyamatát. A kódvisszafejtő programok működése ezzel ellentétes, az alacsony szintű, gépközeli kódot alakítják át magas szintű kóddá. Segítségével beleláthatunk a program forráskódjába akkor is, ha csak egy futtatható állomány áll rendelkezésre, ezáltal könnyebben megvédhetjük gépünket a kártékony szoftverektől.

A fenti feladat megoldására már léteznek különböző kódvisszafejtő programok [1, 2], ugyanakkor ezek legnagyobb hátránya, hogy a fordítóprogramokhoz hasonlóan bonyolult szabályok alapján működnek. Ez azért probléma, mert ezen szabályokat minden programozási nyelv esetén külön meg kell fogalmazni, ami egy bonyolult és időigényes feladat.

Dolgozatomban egy olyan programot mutatok be, ami a fenti problémát a természetes nyelvfeldolgozásban használt gépi tanulási eszközökkel oldja meg. A neurális gépi fordítás az utóbbi években hatalmas fejlődésen ment keresztül[bert], így könnyen adódik, hogy ezen eredményeket fel lehetne használni a kódvisszafejtéshez, annyi különbséggel, hogy nem angolról németre, hanem például assembly-ről C-re fordítunk.[1] Ezen módszer előnye, hogy nem szükséges hozzá programozók hosszú ideig munkája a különböző szabályrendszerek megalkotásához, valamint egy másik nyelvre való áttérés sem okoz különösebb nehézséget. Továbbá a különböző programkód generáló szoftvereknek[**??**] hála, lényegében korlátlan mennyiségű adat áll rendelkezésre.

[1] Majd látni fogjuk, hogy a programozási nyelvek sajátos szerkezete miatt sajnos nem alkalmazhatók egy az egyben a természetes nyelvek fordítása során elért eredmények.

1.1. A kódvisszafejtés lépései

A kódvisszafejtés három fő lépésben történik, ezek közül az első kettőben használtam neurális hálókat. Nulladik lépésként az Assembly kód kinyerése történik a futtatható állományból, erre az `objdump` eszközt használtam.

1.1.1. Assembly szegmentálás

A visszafejtés első lépésében történik az Assembly kód szegmentálása, ahol a cél, hogy úgy daraboljuk fel a kódot egymás utáni blokkokra, hogy egy-egy ilyen blokk egy sor C kódnak feleljen meg. Ehhez egy `LSTM`[**??**] cellákból álló rekurrens neurális hálót használok. (TODO: Ábra) A modell bemenete az Assembly sorok, a kimenete pedig minden sorra 1, ha ott kezdődik egy blokk, 0 egyébként. Ez egy bináris klasszifikációs probléma, melyet a modell könnyedén megtanult.

1.1.2. Maszkolt C kód előállítása

A második lépésben minden Assembly blokkot egy sor C kódnak próbálunk megfeleltetni. Fontos, hogy ebben a lépésben nem várjuk el a változók és a számliterálok pontos visszaállítását. Ezért az eredeti C kódban minden változó előfordulást `VAR`-ra, minden számliterál előfordulást `NUM`-ra cserélünk. A konkrét értékek visszahelyettesítése majd a harmadik lépésben fog megtörténni.

Ezen probléma megoldására egy enkóder-dekóder modellt használtam. Az enkóder első lépésben a bemeneti Assembly blokk sorait kódolja egy vektorba, majd a dekóder ezen vektorból állítja elő a kimeneti tokenek listáját. A lehetséges kimeneti tokeneket a ?? táblázat tartalmazza.

1.1.3. Változók és számliterálok rekonstruálása

A harmadik lépésben minden előállított maszkolt C sorra megpróbáljuk visszaállítani, hogy konkrétan milyen változók[2] és milyen számliterálok szerepeltek ott. Ezeket soronként, a hozzá tartozó Assembly blokk segítségével próbáljuk rekonstruálni, az alábbi lépésekben:

1. Változók és számliterálok kinyerése a megfelelő Assembly blokkból

[2] A konkrét változónevek elvesznek a fordítás során, így ezeket nem lehet visszafejteni, ezért itt annyit követelünk meg, hogy két változó ugyanaz-e vagy sem.

2. Ezek permutálása és behelyettesítése a `VAR` és `NUM` tokenek helyére

3. Az eddig rekonstruált és a mostani behelyettesítésből kapott kódrészlet lefordítása

4. Ha a lefordított Assembly megegyezik[3] az eredeti Assembly-vel, akkor lépünk a következő blokkra, különben visszalépünk a 2. pontra.

Így sorról sorra behelyettesítjük a változókat és számliterálokat, így végül visszanyerjük az eredeti[4] C kódot.

Probléma a szorzással, osztással és modulozással

Az `x0 = x1 + 5 ;` kifejezésnek megfelelő Assembly blokkban megjelenik az 5-ös számliterál, ugyanakkor ez nem minden műveletnél van így. Hatékonysági okokból a számmal való osztás során a fordító nem osztást fog generálni, hanem szorzást és bitshifteléseket. Ezért például a 17-tel való osztás során a vonatkozó Assembly blokkban megjelenő számliterálok a [...]. Láthatjuk, hogy 17 ezek között nincs ott, ezért hiába próbáljuk végig a behelyettesískor az összes számliterált, soha nem fogjuk a megfelelő Assembly blokkot visszakapni.

Ezen probléma megoldására előre regeneráltam, hogy egy adott számmal való osztás során milyen "varázsszámok" jelennek meg az Assembly-ben. Ezután a fenti algoritmus még kiegészítésre kerül annyiban, hogy nem csak az Assembly-ben megjelenő számliterálokat próbáljuk behelyettesíteni a `NUM` tokenek helyére, hanem ha pl. a [...] számok mind szerepelnek, akkor ehhez a listához a 17-et is hozzávesszük.

A szorzás és modulozás során hasonló probléma lép fel, például a [...] számokkal való szorzás esetén egyáltalán nem jelenik meg számliterál az Assembly-ben. Így ezeket mindig hozzá kell venni a lehetséges számliterálok listájához, ha van a kifejezésben `*` token.

A hátránya ennek a módszernek, hogy ha egy kifejezésben több `/`, `%` vagy `*` token van, akkor a lehetséges számliterálok száma nagyon megnő, ezáltal a program futási ideje is sokkal nagyobb lesz, mivel minden lehetséges behelyettesítés kipróbálása során le kell fordítani a kapott programot.

[3] Teljes egyezést nem várhatunk el, hiszen lehet, hogy pl. más regiszterekre hivatkozunk, ezért csak "szerkezeti" egyezést követelünk meg.

[4] A változónevek $x0, x1, \ldots$ lesznek

1.1.4. Assembly beágyazás

Az első két lépésben használt modellek valójában nem a nyers Assembly sorokat kapják meg bemenetként, hanem az azokból előállított kontextusvektorokat. Ez a trükk már régóta ismert a neurális gépi fordítási problémák megoldása során, a legismertebb ilyen megoldás a `Word2Vec`[??]. Ahhoz hasonlóan az Assembly beágyazásnál is cél, hogy a hasonló soroknak közeli vektorokat feleltessünk meg, ezzel megkönnyítve a későbbik során a modell tanulási folyamatát. Én erre a problémára a `Palmtree`[3] eszközt használom. Ez minden sor Assembly-t egy `N` dimenziós vektorba képez le. Működési elve a `BERT`[??]-höz hasonló.

2. fejezet

Felhasználói dokumentáció

A program `Ubuntu 20.04` operációs rendszer alatt fut. A futtasához szükség van `Python 3.10.4`-re, valamint több könyvtárra, ezeket a `requirements.txt` fájl tartalmazza, melyeket a `pip` csomagkezelő segítségével könnyen feltelepíthetünk az alábbi paranccsal: `pip install -r requirements.txt`

A programnak alapvetően két felhasználási módja van, egyrészt lehet saját adatokkal tanítani a szegmentálást és a maszkolt C kód előállítását végző modelleket, másrészt két, nagyszámú adaton előre betanított modellt használva egy grafikus felület segítségével van lehetőség a lefordított C kódok visszafejtésére.

2.1. Tanítás

2.1.1. Adatok

A tanítási adatok egyszerű C fájlok, melyeket a `model/train/raw_data` mappába kell elhelyezni. Ezekből a nyers adatokból még ki kell nyerni a tanításhoz szükséges információkat, ezt a `model/train/features.py` script futtatásával tudjuk megtenni. Ez az előfeldogozott adatokat egy `json` fájlba menti ki, ezt fogjuk a tanítások során majd használni

2.1.2. Szegmentálás és fordítás tanítása

A tanítási folyamatot a `model/train/segmentation_train.py`, illetve `model/train/translation_train.py` program futtatásával tudjuk elindítani. Ha nincs a gépünkön TPU vagy GPU, akkor először egy figyelmeztető üzenetet ír ki a

`Flax` könyvtár, hogy CPU-n fog futni a program. Ezt követően a `Jax` könyvtár ír ki egy figyelmeztető üzenetet, ez a könyvtár belső működésére vonatkozik, nyugodtan figyelmen kívül hagyhatjuk.

Ezután megjelenik egy folyamatjelző sáv, ami a felhasználó számára jelzi, hogy az adott epoch-ban az adatok hányadrészét dolgozta fel eddig a modell. Az epoch végén kiírja, hogy mennyi volt a hibája és a pontossága a modellnek a tanító adatokon, majd a ugyanezt a tesztadatokra.

A program futása végén egy új ablakban megnyílik egy grafikon, ami külön a tanító és teszt példákra szemlélteti, hogy a tanítás során mekkora volt a modell hibája és pontossága. A felugró ablakot bezárva megjelenik egy üzenet, hogy a betanított modell milyen néven került elmentésre. A betanított modellek a `model/train/runs` mappában érhetőek el.

Szegmentálás konfigurálása

A szegementálás tanítása során használt paramétereket a `model/train/segmentation_train.yaml` fájlban lehet konfigurálni. A felhasználó az alábbi beállításokat tudja módosítani:

- `num_epochs`: A tanulási iterációk száma, azt adja meg, hogy a modell összesen hányszor "látja" a teljes tanítóhalmazt. az alapértelmezett érték 10.
- `optimizer`: A tanítási folyamat során milyen optimalizálót használjon a modell, az `optax`[**TODO**] könyvtár alapértelmezett optimalizálói közül lehet választani. Az alapértelmezett érték `"adam"`[**TODO**].
- `learning_rate`: A tanulási ráta, azt szabályozza, hogy a gradiens módszer során milyen mértékben változtassa paramétereit a modell. Az alapértelmezett érték 0.003.
- `hidden_size`: Az `LSTM` által használt rejtett vektor mérete, az alapértelmezett érték 256.
- `batch_size`: A részmintaméret, azt lehet beállítani vele, hogy egy tanítási lépés során egyszerre hány adattal dolgozzon a modell. Az alapértelmezett érték 4.

- `max_len`: A bemenet, vagyis, hogy hány sor Assembly-ből áll a program maximális hossza. Az alapértelmezett érték 90.
- `embedding_size`: A begyazás után hány dimenziós vektort kapunk. Alapvetően a beágyazást egy előre betanított `Palmtree` modell végzi, ami 128 dimenziós vektorokkal dolgozik, így az alapértelmezett érték 128, amit csak a `Palmtree` módosítása esetén változtassunk.
- `test_ratio`: Az összes adat hányadrésze legyen a teszthalmaz (a maradék adat adja a tanítóhalmazt). Az alapértelmezett érték $0,2$.

Fordítás konfigurálása

A fordítás tanítása során használt paramétereket a `model/train/translation_train.yaml` fájlban lehet konfigurálni. A felhasználó az alábbi beállításokat tudja módosítani:

- `num_epochs`: Ld. szegmentálás konfigurálása.
- `optimizer`: Ld. szegmentálás konfigurálása.
- `learning_rate`: Ld. szegmentálás konfigurálása.
- `hidden_size`: Ld. szegmentálás konfigurálása.
- `batch_size`: Ld. szegmentálás konfigurálása.
- `max_input_len`: A bemenet, vagyis, hogy hány sor Assembly-ből áll egy visszafordítandó blokk, maximális hossza. Az alapértelmezett érték 40.
- `max_output_len`: A kimenet, vagyis, hogy hány token-ből állhat a visszafordítás során kapott maszkolt C kód maximális hossza. Az alapértelmezett érték 40.
- `test_ratio`: Ld. szegmentálás konfigurálása.

2.2. Betanított modell használata

2.2.1. Grafikus interfész

A `main.py` programot futtatva egyrészt megjelennek a korábban írt figyelmeztető üzenetek a konzolon, másrészt egy új ablakban a program grafikus felülete. Itt a `Browse` gombra kattintva kiválaszthatjuk a visszafejteni kívánt fájlt, majd ezt a `Submit` gombra kattintva tudjuk betölteni. Ekkor a baloldai szövegdobozban megjelenik a kiválasztott futtatható állományból kinyert Assembly kód. Ezután a `Decompile` gombra kattintva megkezdődik a kód visszafordítása, ami az eredeti C kód bonyolultságától függően néhány másodperctől fél percig tart. Ezután megnézhetjük, hogy a bevezetésben leírt lépesek hogy zajlottak. a `Segmentation` gombra kattintva a jobboldali szövegdobozban megjelenik a feldarabolt Assembly, a blokkokat egy-egy üres sor választja el. A `Masked C` gombra kattintva megjelenik az eredeti C kód "sablonja", itt még az összes változó helyén a `VAR` token, a számliterálok helyén pedig a `NUM` token szerepel. A `Reconstructed C` gombra kattintva pedig az iteratív rekonstruálás során visszafejtett C kód jelenik meg a szövegdobozban, ami szerkezetében megegyezik[1] az eredeti C kóddal.

2.2.2. Konzolos interfész

Ha a főprogramot a `-gui=False` argumentummal futtatjuk, akkor egy konzolos interfész segítségével is kipróbálhatjuk a program működését. Először megjelennek a szokásos figyelmeztető üzenetek, majd a program bekéri a visszafejtendő fájl nevét. Ide ha az `exit` parancsot írjuk, akkor leáll a program futása. Ha nem létező fájlnevet adunk meg vagy nem megfelelő formátumű a fájl, akkor a program egy `TODO` hibaüzenetet ír ki. Ha létezik a fájl és megfelelő formátumú, akkor a program rögtön elkezdi a visszafordítást, ez általában egy pár másodperces folyamat, bonyolultabb programok esetén kicsivel több.

2.2.3. A modell konfigurálása

A modell különböző paramétereit a `model/model_config.yaml` fájlban lehet konfigurálni. A felhasználó az alábbi beállításokat tudja módosítani:

[1] A betanított modellek nem 100% pontosságúak, így nem garantált, hogy mindig sikeres lesz a visszafordítás.

- `segmentation/model_path`: A szegmentálás során használt modell relatív elérési útvonala. Az alapértelmezett érték `"flax_models/segmentation.params"`, ez egy előre, sok adaton betanított modell. A felhasználónak lehetősége van a tanítási folyamat során elmentett modellek betöltésére is, ekkor annak a relatív elérési útvonalát kell megadni.
- `segmentation/hidden_size`: Ld. szegmentálás konfigurálása.
- `segmentation/max_len`: Ld. szegmentálás konfigurálása.
- `translation/model_path`: A fordítás során használt modell relatív elérési útvonala. Az alapértelmezett érték `"flax_models/translation.params"`, ez egy előre, sok adaton betanított modell. A szegmentáláshoz hasonlóan itt is van lehetőség a felhasználó által betanított modellek betöltésére az útvonal módosításával.
- `translation/vocab_path`: Ide egy json formátumú fájl elérési útvonalát kell megadni, ami egy szótárat tartalmaz, melyben szerepel, hogy a lehetséges kimeneti tokeneket milyen számra képezzük. Az alapértelmezett érték `"flax_models/vocab.json"`.
- `hidden_size`: Ld. szegmentálás konfigurálása.
- `max_input_len`: Ld. fordítás konfigurálása.
- `max_output_len`: Ld. fordítás konfigurálása.

3. fejezet

Fejlesztői dokumentáció

A főprogram (betanított modell használata) alapvetően nézet-modell architektúrát követ.

3.1. Felhasználói felület

A grafikus és a konzolos felhasználói felületek a `ui` könyvtárban érhetőek el. A konzolos interfész egy nagyon egyszerű kommunikációt biztosít: bekéri a felhasználótól a fájl nevét, majd az eredményt struktúrálva kiírja a konzolra.

A grafikus felület ezzel szemben komplexebb. Az implementálás során a `PySimpleGUI`[**TODO**] könyvtárat használtam, melynek segítségével könnyen lehet grafikus felületű alkalmazásokat készíteni Python nyelven. A `GUI` osztály konstruktorában történik meg a felhasználói felület elemei

3.2. Jax & Flax

Ma már rengeteg különböző magas szintű könyvtár érhető el a `Python` nyelvhez, melyek megkönnyítik a mélytanulási algoritmusok implementálást. Én ezek közül a Google által fejlesztett `Jax`-et[4] és az arra épülő `Flax`-et[5] használtam.

3.2.1. Jax

A Jax könyvtár két legfőbb komponense a gyorsított lineáris algebra (`XLA`) és az automatikus differenciálás. Mindkettő nagyban hozzájárul ahhoz, hogy könnyeb-

ben és gyorsabban lehessen különböző gépi tanulási algoritmusokat és modelleket implementálni, ezzel gyorsítva az ezirányú kutatásokat.

A gépi tanulási algoritmusok során nagyon sok lineáris algebrai műveletet (pl. mátrixszorzás) kell végezni, így elenghetetlen, hogy ezeket minél gyorsabban elvégezzük. Ebben segít az `XLA`, ami egy doménspecifikus fordító, célja, segítségével a lineáris algebrai műveletek elvé

3.2.2. Flax

[3]

4. fejezet

Összegzés

Irodalomjegyzék

[1] National Security Agency. *Ghidra*. `https://ghidra-sre.org/`. 2019.

[2] Peter LaFosse Jordan Wiens Rusty Wagner. *Binary Ninja*. `https://binary.ninja/`. 2016.

[3] Xuezixiang Li, Yu Qu és Heng Yin. „Palmtree: learning an assembly language model for instruction embedding". *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021, 3236–3251. old.

[4] James Bradbury és tsai. *JAX: composable transformations of Python+NumPy programs*. 0.2.5. verzió. 2018. URL: `http://github.com/google/jax`.

[5] Jonathan Heek és tsai. *Flax: A neural network library and ecosystem for JAX*. 0.4.1. verzió. 2020. URL: `http://github.com/google/flax`.

Ábrák jegyzéke

Táblázatok jegyzéke