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, ami nem igényel olyan bonyolult szabályokat. A neurális gépi fordítás az utóbbi években hatalmas fejlődésen ment keresztül[3], 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, mint említettük, hogy nem szükséges programozók hosszú ideig tartó 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.

A dolgozat 2. fejezete bemutatja, hogy hogyan lehet neurális hálók segítségével megoldani a kódvisszafejtés problémáját, valamint hogyan lehet ezt részfeladatokra bontani. A 3. fejezetben található a program használatának leírása. Egyrészt lehetőség van előre betanított hálókkal a kódvisszafejtés kipróbálására, másrészt saját adatokkal a neurális hálók tanítására. A 4. fejezetben szerepel a kódbázis felépítése, az egyes modulok működése és felelősségi köre, valamint a tesztelés folyamata.

2. fejezet

A kódvisszafejtés lépései neurális hálók segítségével

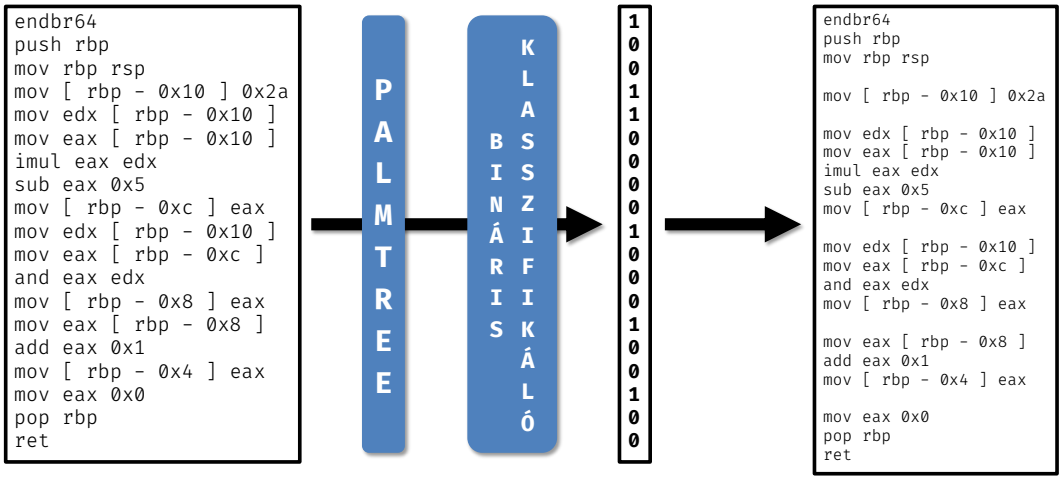
A kódvisszafejtés neurális hálók segítségével három fő lépésben történik. Előkészítésként az Assembly kód kinyerése történik a futtatható állományból, például az `objdump`[4] eszköz segítségével. Ezután az Assembly kódot szegmentáljuk, így olyan blokkokat kapunk, amik egy sor C kódnak feleltethetőek meg. Ezek után ezekből a blokkokból előállítjuk a megfelelő C sorokat, először csak egy sablont, ahol a változók és számliterálok még maszkolva vannak. Végül pedig ezen C kód és az Assembly kód segítségével előállítjuk az eredeti C kódot.

```
Assignments := Assignments Assignment | Assignment
 Assignment := Var = Expr
        Var := x0, …, x19
       Expr := Var | Num | BinaryExpr | UnaryExpr
        Num := -100, …, 100
  UnaryExpr := UnaryOp Var | Var UnaryOp
    UnaryOp := ++ | -- | -
 BinaryExpr := Expr BinaryOp Expr
   BinaryOP := + | - | * | / | % | & | | | ^ | >> | <<
```

2.1. ábra. A modell tanítása során használt C kódok alapját képező nyelvtan.

2.1. 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. Ezt a feladatot egy `LSTM`[5] cellákból álló rekurrens neurális háló felhasználásával tudjuk megoldani. A modell bemenetét az Assembly sorok képezik, a kimenete pedig minden sorra 1, ha ott kezdődik egy blokk, 0 egyébként. Ez egy bináris klasszifikációs probléma, melyet a modell könnyedén megtanulhat.



2.2. ábra. A szegmentálás folyamata.

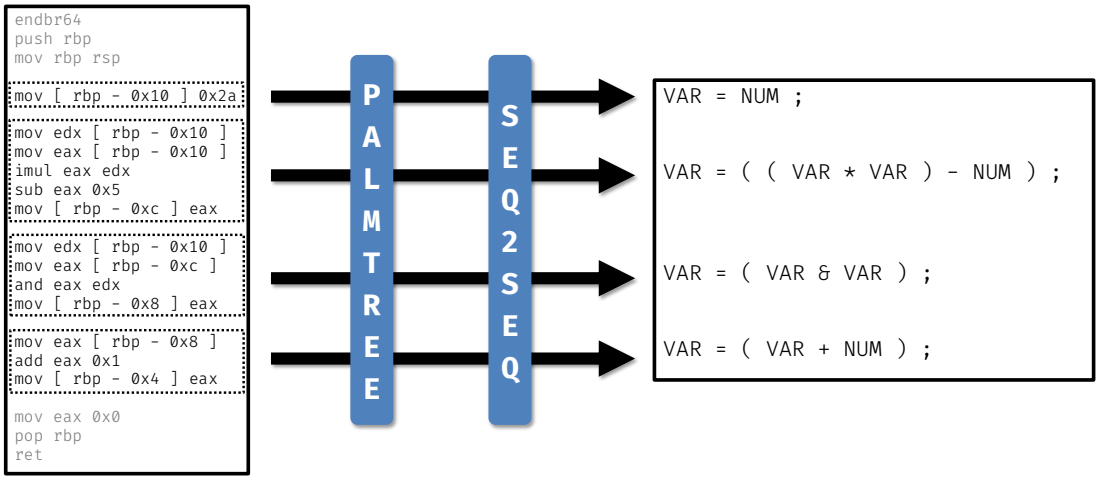
2.2. Maszkolt C kód előállítása

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.

A fenti problémát egy enkóder-dekóder modell segítségével oldhatjuk meg. 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 és a hozzájuk tartozó alapértelmezett indexeket az alábbi ábra tartalmazza.

• <PAD>	0	• &	9
• <SOS>	1	• /	10
• VAR	2	• -	11
• =	3	• *	12
• NUM	4	• ^	13
• ;	5	• +	14
• (	6	• >>	15
• %	7	• |	16
•)	8	• <<	17

2.3. ábra. A lehetséges kimeneti tokenek és a hozzájuk tartozó alapértelmezett indexek.



2.4. ábra. A maszkolt C kód előállításának folyamata.

2.3. Változók és számliterálok rekonstruálása

A harmadik lépésben minden előállított maszkolt C sorra megpróbáljuk visszaállítani, hogy konkrétan milyen változók[1] 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. Ezek permutálása és behelyettesítése a `VAR` és `NUM` tokenek helyére.

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

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

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

Így sorról sorra behelyettesítjük a változókat és számliterálokat, míg végül visszanyerjük az eredeti[3] C kódot.

2.4. 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 $(30841, 32, 3, 31)$. Láthatjuk, hogy 17 ezek között nincs ott, ezért hiába próbáljuk végig a behelyettesítéskor az összes számliterált, soha nem fogjuk a megfelelő Assembly blokkot visszakapni.

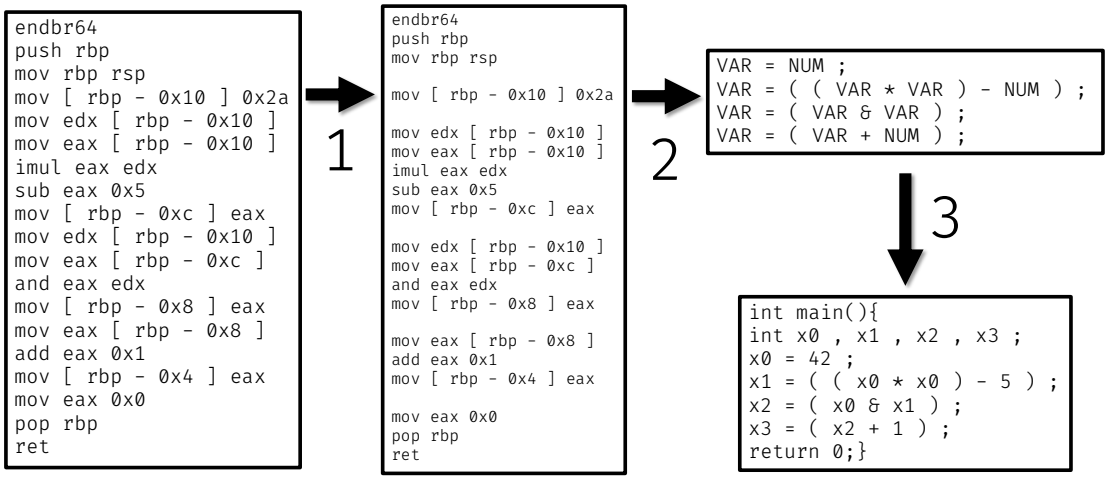
Ezen probléma megoldására előre legenerálható, hogy egy adott számmal való osztás során milyen "varázsszámok" jelennek meg az Assembly-ben. Ezután a fenti algoritmus még kiegészült annyiban, hogy nem csak az Assembly-ben megjelenő számliterálokat próbáljuk behelyettesíteni a `NUM` tokenek helyére, hanem, ha pl. a $(30841, 32, 3, 31)$ számok mind szerepelnek, akkor ehhez a listához a 17-et is hozzávesszük.

A szorzás és modulozás során hasonló probléma lép fel, például a $(2, 3, 6)$ 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.

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

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



2.5. ábra. A kódvisszafejtés lépései.

2.5. 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`[6]. Ahhoz hasonlóan az Assembly beágyazásnál is cél, hogy a hasonló soroknak közeli vektorokat feleltessünk meg, ezzel megkönnyítve a későbbiek során a modell tanulási folyamatát. Ezen feladat megoldására jelenleg a `Palmtree`[7] az egyik legjobb eszköz. Ez minden sor Assembly-t egy `N` dimenziós vektorba képez le, működési elve a `BERT`[3]-höz hasonló.

3. fejezet

Felhasználói dokumentáció

A program `Ubuntu 20.04` operációs rendszer alatt fut. A futtatáshoz 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.

3.1. Tanítás

Az Assembly kód szegmentálását és a maszkolt C kód előállítását két külön neurális háló végzi, a felhasználónak ezek tanítására van lehetősge.

3.1.1. Adatok

A tanítási adatok egyszerű C fájlok, melyeket a `model/train/raw_data` mappában 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.

3.1.2. Szegmentálás és fordítás tanítása

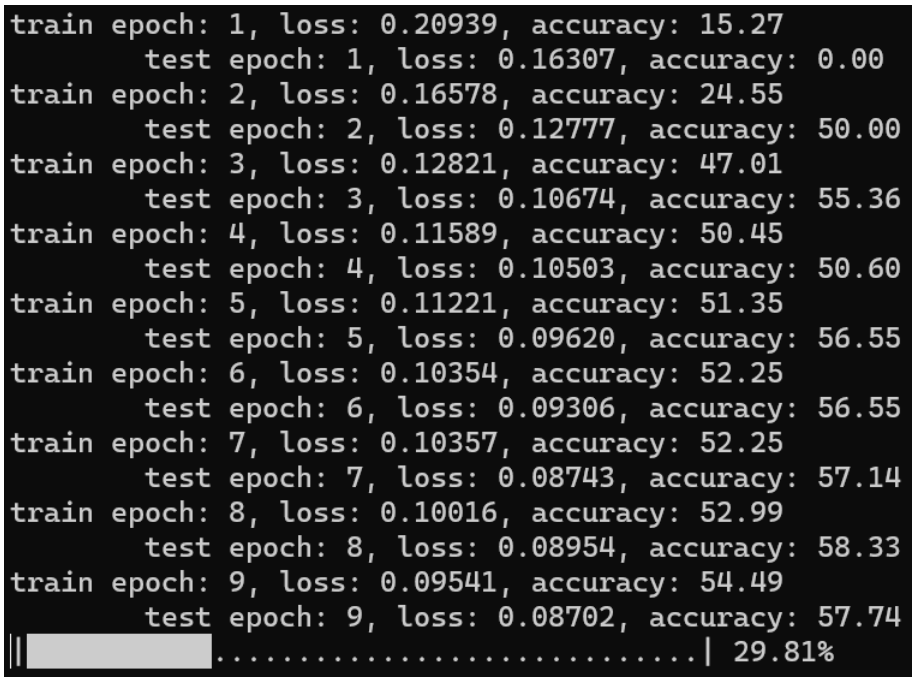
A tanítási folyamatot a `model/train/segmentation_train.py`, illetve `model/train/translation_train.py` program futtatásával tudjuk elindítani. Ha nincs a gépünkön TPU vagy GPU, akkor először egy figyelmeztető üzenetet ír ki a `Flax` könyvtár, hogy CPU-n fog futni a program. Ezt követően a `Jax` könyvtár ír ki egy figyelmeztető üzenetet, ez a könyvtár belső működésére vonatkozik, nyugodtan figyelmen kívül hagyhatjuk.

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and
 rerun for more info.)
/home/csandras/venvs/decompiler/lib/python3.10/site-packages/jax/_src/tree_util.py:
188: FutureWarning: jax.tree_util.tree_multimap() is deprecated. Please use jax.tre
e_util.tree_map() instead as a drop-in replacement.
  warnings.warn('jax.tree_util.tree_multimap() is deprecated. Please use jax.tree_u
til.tree_map() '
```

3.1. ábra. A futtatás során kapott figyelmeztető üzenetek.

Ezután megjelenik egy folyamatjelző sáv, ami a felhasználó számára jelzi, hogy az adott tanítási iterációban az adatok hányadrészét dolgozta fel eddig a modell. A tanítási iteráció végén kiírja, hogy mennyi volt a hibája és a pontossága a modellnek a tanító adatokon, majd 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 mentődik el. A betanított modellek a `model/train/runs` mappában érhetőek el.



3.2. ábra. A fordítás tanítása. A felhasználó számára egy folyamatjelző sáv mutatja, hogy az adott tanítási iterációban a tanítópéldák hány százalékát dolgozta fel eddig a modell. Ezen kívül látszik minden tanítási iterációra a modell hibája és pontossága a tanító-, valamint a tesztpéldákon.

Szegmentálás konfigurálása

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

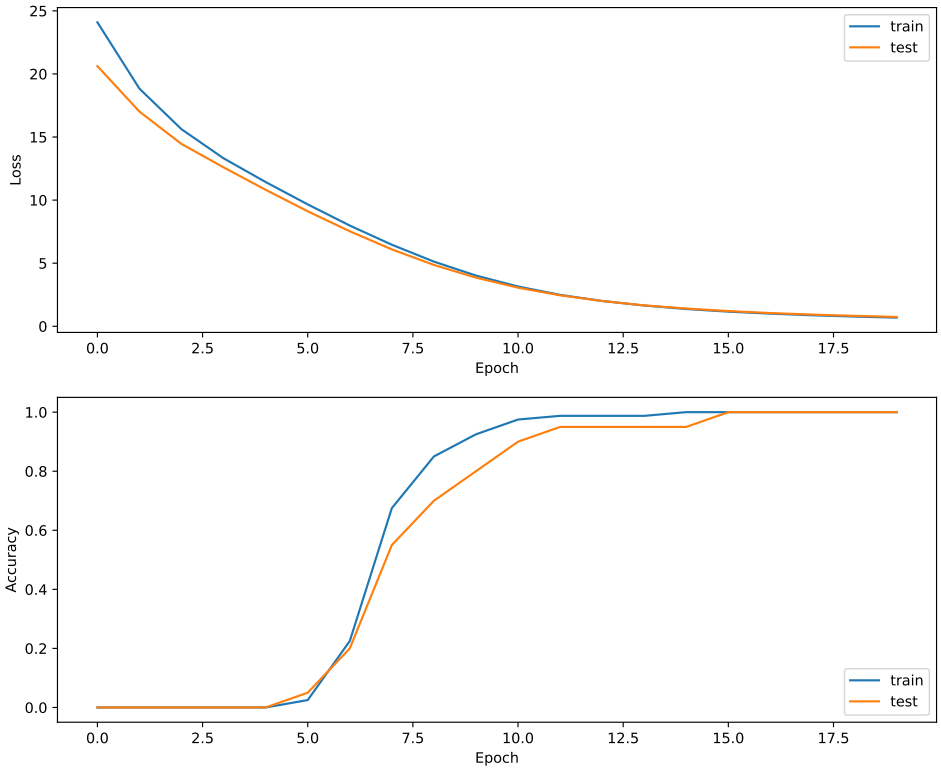
- `num_epochs`: A tanulási iterációk száma, azt adja meg, hogy a modell összesen hányszor "látja" a teljes tanítóhalmazt. Az alapértelmezett érték 10.
- `optimizer`: A tanítási folyamat során milyen optimalizálót használjon a modell, az `optax`[8] könyvtár alapértelmezett optimalizálói közül lehet választani. Az alapértelmezett érték `"adam"`[9].
- `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 beágyazás után hány dimenziós vektort kapunk. Alapvetően a beágyazást egy előre betanított `Palmtree`[7] 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.



3.3. ábra. A hiba és a pontosság változása a tanító- és tesztpéldakon a szegmentálás tanítása során.

Fordítás konfigurálása

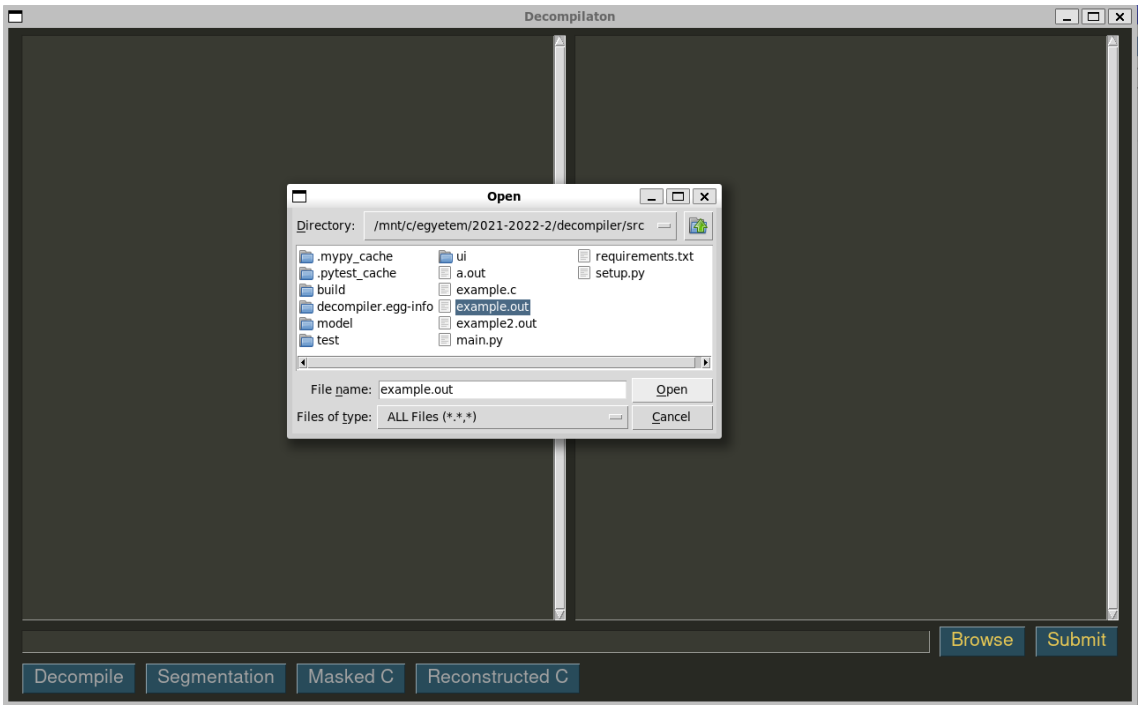
A fordítás tanítása során használt paramétereket a `model/train/translation_train.yaml` fájlban lehet konfigurálni. A felhasználó az alábbi beállításokat tudja módosítani:

- `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.

3.2. Betanított modell használata

3.2.1. 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 bal oldali szövegdobozban megjelenik a kiválasztott futtatható állományból kinyert Assembly kód.

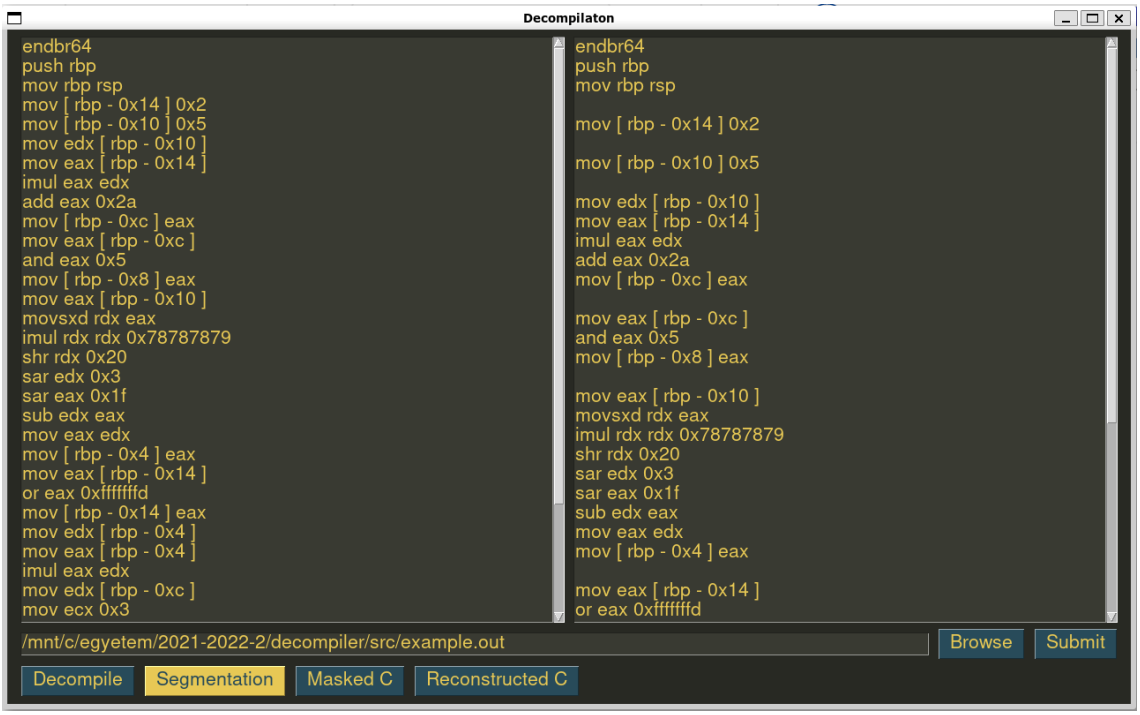


3.4. ábra. A visszafejtendő fájl betöltése a grafikus felületen.

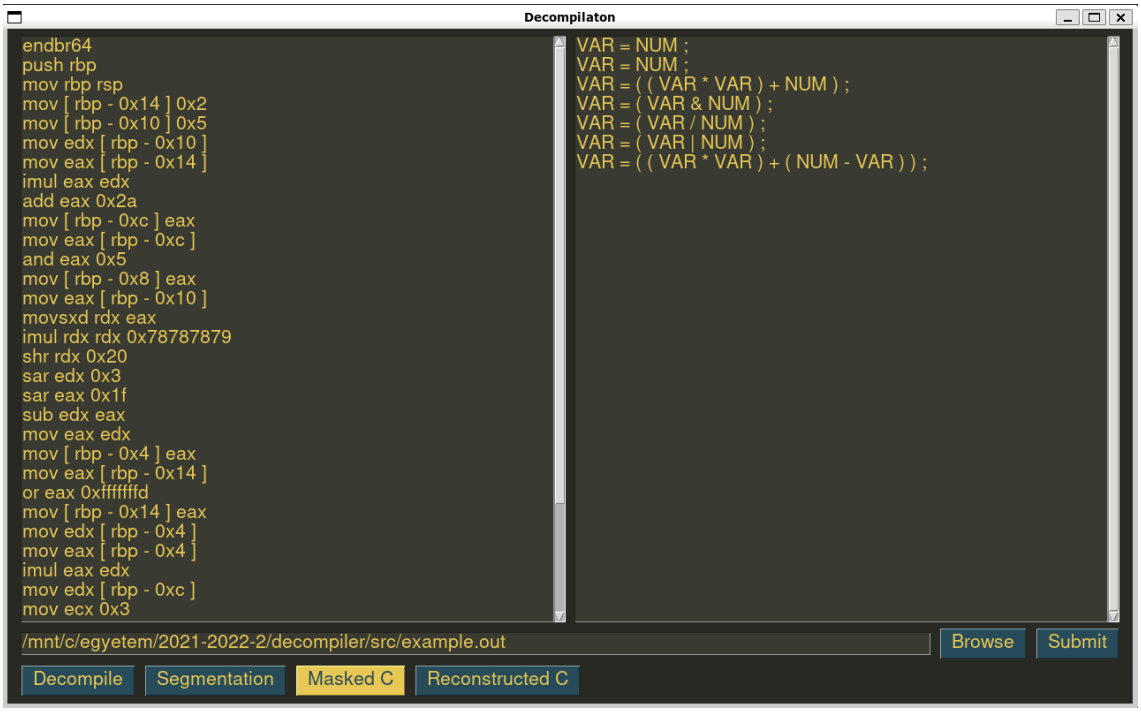
Ezután a `Decompile` gombra kattintva a jobb oldali szövegdobozban megjelenik a `Decompiling code, it may take a few seconds` felirat, majd megkezdődik a kód visszafordítása, ami az eredeti C kód bonyolultságától függően néhány másodperctől fél percig tart. A kódvisszafejtés végeztével a szövegdobozban a következő szöveg jelenik meg: `Decompilaton done, click on the buttons below to see the result.`

Ezután megnézhetjük, hogy a bevezetésben leírt lépesek hogy zajlottak. a `Segmentation` gombra kattintva a jobb oldali 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.

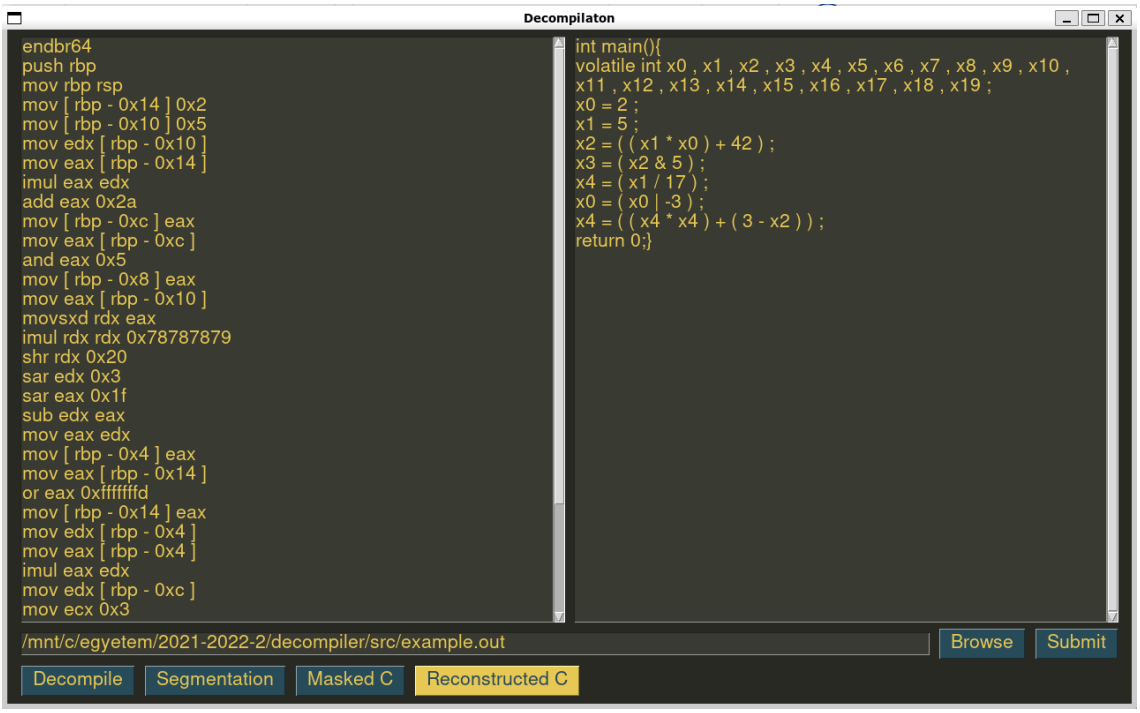
[1] A betanított modellek nem 100% pontosságúak, így nem garantált, hogy mindig sikeres lesz a visszafordítás.



3.5. ábra. A szegmentálás megjelenítése a grafikus felületen.



3.6. ábra. A maszkolt C kód megjelenítése a grafikus felületen.



3.7. ábra. A rekonstruált C kód megjelenítése a grafikus felületen.

3.2.2. 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 a program a `Closing program...` szöveget írja ki a képernyőre, és leáll a program futása. Ha nem létező fájlnevet adunk meg vagy nem megfelelő formátumú a fájl, akkor a program egy `File` *`fájlnév`* `not found` hibaüzenetet ír ki, majd leáll a program. 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.

```
ORIGINAL:
endbr64
push rbp
mov rbp rsp
mov [ rbp - 0x10 ] 0x2a
mov edx [ rbp - 0x10 ]
mov eax [ rbp - 0x10 ]
imul eax edx
sub eax 0x5
mov [ rbp - 0xc ] eax
mov edx [ rbp - 0x10 ]
mov eax [ rbp - 0xc ]
and eax edx
mov [ rbp - 0x8 ] eax
mov eax [ rbp - 0x8 ]
add eax 0x1
mov [ rbp - 0x4 ] eax
mov eax 0x0
pop rbp
ret
```

(a) A modell bementét képező Assembly kód.

```
SEGMENTAION:
endbr64
push rbp
mov rbp rsp

mov [ rbp - 0x10 ] 0x2a

mov edx [ rbp - 0x10 ]
mov eax [ rbp - 0x10 ]
imul eax edx
sub eax 0x5
mov [ rbp - 0xc ] eax

mov edx [ rbp - 0x10 ]
mov eax [ rbp - 0xc ]
and eax edx
mov [ rbp - 0x8 ] eax

mov eax [ rbp - 0x8 ]
add eax 0x1
mov [ rbp - 0x4 ] eax

mov eax 0x0
pop rbp
ret
```

(b) A szegmentált Assembly kód.

```
MASKED C:
VAR = NUM ;
VAR = ( ( VAR * VAR ) - NUM ) ;
VAR = ( VAR & VAR ) ;
VAR = ( VAR + NUM ) ;
```

(c) A maszkolt C kód.

```
RECONSTRUCTED C:
int main(){
volatile int x0 , x1 , x2 , x3 , x4 , x5 , x6
 , x7 , x8 , x9 , x10 , x11 , x12 , x13 , x14
 , x15 , x16 , x17 , x18 , x19 ;
x0 = 42 ;
x1 = ( ( x0 * x0 ) - 5 ) ;
x2 = ( x0 & x1 ) ;
x3 = ( x2 + 1 ) ;
return 0;}
```

(d) A rekonstruált C kód.

3.8. ábra. A kódvisszafejtés lépései a konzolos interfészen.

3.2.3. A modell konfigurálása

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

- `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"`.
- `translation/hidden_size`: Ld. szegmentálás konfigurálása.
- `translation/max_input_len`: Ld. fordítás konfigurálása.
- `translation/max_output_len`: Ld. fordítás konfigurálása.

4. fejezet

Fejlesztői dokumentáció

A főprogram (betanított modell használata) alapvetően nézet-modell architektúrát követ. A programot a `main.py` fájl futtatásával tudjuk elindítani. Ebben a fájlban az argumentumok parszolása után a `-gui` kapcsoló értékétől függően a grafikus vagy a konzolos felületért felelős osztályt példányosítjuk, paraméterül átadva neki a mögöttes működésért felelős `Decompiler` osztály egy példányát.

4.1. Felhasználói felület

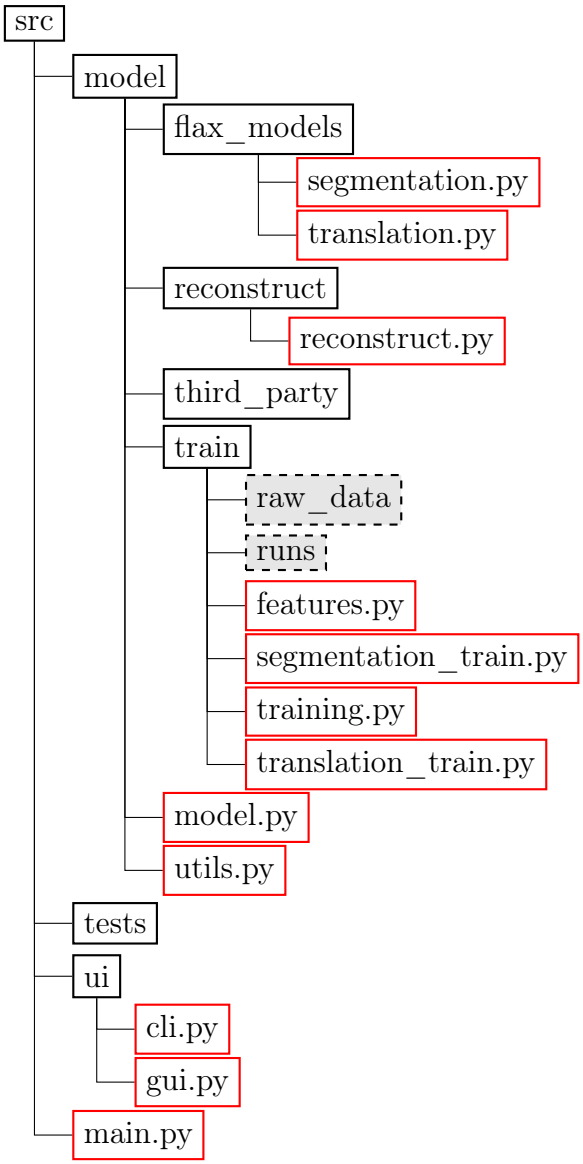
A grafikus és a konzolos felhasználói felületek a `ui` könyvtárban érhetőek el. A konzolos interfész egy nagyon egyszerű kommunikációt biztosít: bekéri a felhasználótól a fájl nevét, majd az eredményt struktúrálva kiírja a konzolra.

A grafikus felület ezzel szemben komplexebb, ennek implementálása a `PySimpleGUI`[10] könyvtár segítségével történt. A `GUI` osztály konstruktorában történik meg a felhasználói felület elemeinek létrehozása, melyek a következők:

- 2 szövegdoboz (`Multiline`): a bal oldali jeleníti meg az eredeti Assembly kódot, a jobb oldali a visszafejtés 3 lépését: a szegmentált Assembly-t, a maszkolt C és a rekonstruált C-t.
- Szövegbeviteli mező (`TextInput`): Itt lehet megadni a betölteni kívánt fájl nevét. Ez alapvetően inaktív, a szövegbevitelt egy fájlkereső biztosítja.
- Fájlkereső (`FileBrowswer`): A `PySimpleGUI` könyvtár több beépített gombfajtát is tartalmaz, ezek közül a `FileBrowser` az egyik. Erre kattintva egy új ablakban megjelenik egy fájlkereső, ahol kiválaszthatjuk a kívánt futtatható fájlt.

Alapértelmezetten a `layout`-ban előtte lévő felhasználói felületi objektumhoz köti a kiválasztott fájl elérési útvonalát, esetünkben ez most a `TextInput` mező.

- Gomb (`Button`): A fájlkeresőt leszámítva összesen 5 gomb található a felhasználói felületen:
  - `Submit`: Erre kattintva a `TextInput`-ban lévő elérési útvonalat átadjuk a modell `open_binary_file` metódusának, majd a bal oldali szövegdobozban megjelenítjük a betöltött fájlból kinyert Assembly-t.
  - `Decompile`: Erre kattintva csak meghívjuk a modell `decompile` metódusát, illetve jobb oldali szövegdobozban jelezzük a felhasználó számára, hogy elkezdődött, illetve befejeződött a kódvisszafejtés folyamata.
  - `Segmentation`, `Masked C`, `Reconstructed C`: Ezen gombokra kattintva a modell megfelelő getter-ét hívva megjelenítjük a jobb oldali szövegdobozban a kódvisszafejtés folyamatának megfelelő lépését.



4.1. ábra. A kódbázis felépítése.

4.2. Gépi tanulási modellek

4.2.1. Jax & Flax

Ma már rengeteg különböző magas szintű könyvtár érhető el a `Python` nyelvhez, melyek megkönnyítik a mélytanulási algoritmusok implementálást, ezek közül most a Google által fejlesztett `Jax`-et[11] és az arra épülő `Flax`-et[12] használjuk.

A Jax könyvtár két legfőbb komponense a gyorsított lineáris algebra (`XLA`) és az automatikus differenciálás. Mindkettő nagyban hozzájárul ahhoz, hogy könnyebben 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 elengedhetetlen, hogy ezeket minél gyorsabban elvégezzük. Ebben segít az `XLA`, ami egy doménspecifikus fordító, célja, hogy a lineáris algebrai műveletek elvégzését minél gyorsabban meg tudjuk tenni, ezzel gyorsítva a kutatási munkákat.

A másik fontos része a `Jax`-nek az automatikus differenciálás. Segítségével tetszőleges `Python` és `NumPy`[13] függvények kompozícióinak is tudjuk venni a deriváltját, ezzel nagyban megkönnyítve a gradiens módszer használatát.

4.2.2. Szegmentáló modell

A szegmentálást a `flax_models/segmentation.py` modulban található `SegmentationModel` osztály végzi. Ez az osztály a `Flax` könyvtár `linen` moduljának `Module` osztályából származik le, ez egy általános, neurális hálókat leíró osztály. A bemeneti adat először egy rekurrens neurális hálón (RNN) kerül átfuttatásra, ezt az `RNN` osztály végzi, ami egy egyszerű, `LSTM` cellákból álló neurális háló. Ezután következik egy teljesen kapcsolt réteg, majd a kimenetre még alkalmazzuk a szigmoid-függvényt, amivel a kimenetet a $(0, 1)$ intervallumba képezzük le.

4.2.3. Fordító modell

A maszkolt C kód előállítását a `flax_models/translation.py` modulban található `Seq2seq` osztály végzi. Ez egy egyszerű enkóder-dekóder modell, az `Encoder` osztály működése szinte megegyezik a `SegmentationModel` működésével, a teljesen kapcsolt réteget és a szigmoid-függvényt leszámítva. Ezután az enkóder által előállított kontextusvektorból állítja elő a `Decoder` a kimeneti tokeneket. A dekóder is egy `LSTM` cellákból álló rekurrens neurális háló. Ezt követi aztán egy teljesen kapcsolt réteg, ahol a kimeneti rétegben lévő neuronok száma megegyezik a lehetséges kimeneti tokenek számával. Erre aztán még alkalmazzuk a `softmax` függvényt, ami lényegében egy valószínűségi eloszlást rendel a kimeneti tokenekhez.

4.3. Tanítási folyamat

4.3.1. Nyers adatok feldolgozása

A tanító példákat egyszerű C fájlokból nyerjük ki. Ezek előállítása a felhasználó feladata, lehet valamilyen külső könyvtárat használni, vagy akár egy saját szkriptet írni a megfelelő példák generálására.

A tanításhoz szükséges információkat a `model/train/features.py` szkript futtatásával tudjuk kinyerni a nyers adatokból. Ez röviden a következőképpen működik:

1. `gcc`-vel[14] lefordítjuk az adott C programot debug módban, az utóbbi a szegmentálás előállításához kell.
2. Az `objdump`[4] bináris elemző eszközzel visszafejtjük az eredeti Assembly-t.
3. Reguláris kifejezésekkel kinyerjük a `main` függvényhez tartozó kódrészt.
4. A kinyert kódrészből előállítjuk az Assembly sorokat és a hozzátartozó címkéket (0/1).
5. Az Assembly sorokat beágyazzuk a `Palmtree`-vel.
6. Az eredeti C kódból előállítjuk a maszkolt C-t a változók és a számliterálok lecserélésével.
7. A beágyazott Assembly-t, a címkéket, valamint a maszkolt C sorokat elmentjük a `model/train/data.json` fájlba.

4.3.2. Modellfüggetlen tanítási lépések

Az általános, a szegmentáló és a fordító modell által is használt tanítási függvényeket a `model/train/training.py` fájl tartalmazza. Az ebben a modulban lévő metódusok felelősek egy tanítási iteráció végrehajtásáért, közben a modell belső állapotát leíró paraméterek frissítésért, valamint ezen állapot exportálásáért, továbbá a tanulási eredmények vizualizációjáért.

train_epoch

A tanítási folyamat egy iterációjáért a `train_epoch` függvény felelős. Ez először véletlenszerűen permutálja a tanítóhalmaz elemeit, majd ezeket részmintákra bontja.

Ezután minden részmintára végrehajtja a tanítás egy lépését (közben frissítve a modell paramétereit), majd visszatér a tanítási iterációt jellemző metrikákkal (hiba és pontosság). A függvény a következő paramétereket várja:

- `state`: A modell állapotát leíró változó. Tartalmazza a különböző paramétereket, illetve egy `apply` függvényt, amivel az adatok egy előre propagációját tudjuk végrehajtani.
- `train_x`: A tanítóhalmaz bemenetei.
- `train_y`: A tanítóhalmaz kimenetei.
- `train_lengths`: A tanítóhalmaz elemeinek hossza.
- `batch_size`: A részminta mérete, ekkora darabokra fogjuk feldarabolni a tanítóhalmazt.
- `epoch`: Éppen hányadik tanítási iterációban vagyunk, ennek a hiba és pontosság megjelenítésekor van csak szerepe.
- `rng`: Véletlenszám generátor a `Jax` könyvtár `random` moduljából, ez szabályozza a tanítóhalmaz permutációjának véletlenségét.
- `error_fn`: Hiba függvény, a modell által adott kimenet és az elvárt kimenet (valamint a példák hosszai) alapján számol egy hibát, amit majd a gradiens módszer egy lépésénél használunk.
- `metric_fn`: Metrika függvény, az `error_fn` működésénél annyival több, hogy a hibán túl egy pontosság értéket is számol.

A függvény kimenete a `state` változó frissített értéke, valamint a tanítási iterációra vonatkozó metrikák.

train_step

A `train_step` függvény egy részmintára hajtja végre a tanítási folyamat egy lépését. A függvényen belül definiálva van egy `loss_fn` nevű függvény, ami a modell belső állapotát jellemző paraméterek függvényében adja meg a modell hibáját. A `Jax` könyvtárba épített automatikus differenciálásnak köszönhetően könnyedén meg tudjuk határozni a paraméterek szerinti gradienseit a hibafüggvénynek. Ezután a

gradiens módszernek megfelelően léptetünk egyet a modellen, ezt a `state` változón meghívott `apply_gradients` metódus fogja végezni, amit a `state` létrehozásakor hozzákötöttünk az `optax` könyvtár optimalizálójához.

A függvény a `state` változó frissített értékével, valamint a részmintára vonatkozó metrikákkal tér vissza.

A függvény felett szerepel a `jax.jit` dekorátor, ez azt jelenti, hogy ez a függvény futási időben fordított (just-in-time compiled) lesz. Ennek az az előnye, hogy csak akkor fogja a fordító újrafordítani a metódust, ha más szerkezetű argumentumokkal hívjuk meg, egyébként mindig a lefordított változatot hívja. Mivel a tanítás során minden esetben azonos dimenziójú paramétereket adunk át a függvénynek, ezért elég csak a tanítás során egyszer lefordítani a függvényt, ezzel lényegesen meggyorsítva a tanítás folyamatát.

Sajnos a `Jax` függvényparaméterekkel nem tud dolgozni, ezért hogy működjön a fenti megoldás, meg kell adnunk, hogy az `error_fn` és `metric_fn` paraméterek statikusak. Ekkor mindig, amikor egy másik hibafüggvényt vagy metrikafüggvényt adunk át, újra kell fordítani a `Jax`-nek a függvényt, de mivel egy tanítási folyamat során végig ugyanazokat használjuk, ezért ez nem okoz problémát.

eval_model & eval_step

Az `eval_model` és `eval_step` függvények végzik a modell kiértékelését. Bemenete a teszthalmaz, kimenete pedig a modell által prediktált és a valódi kimenetekből számolt hiba és pontosság. A `train_step` függvényhez hasonlóan a `eval_step` függvény is futási időben fordított, ezzel gyorsítva a tanítási folyamatot.

visualize & plot

A `visualize` és `plot` függvények a `matplotlib`[15] könyvtár segítségével grafikonon jelenítik meg a felhasználó számára, hogy hogyan változott a tanítási folyamat során a modell hibája és pontossága a tanító,- illetve teszthalmazon. A vízszintes tengelyen a tanítási iteráció, a függőleges tengelyen pedig a hiba, illetve a pontosság szerepel. Ezt szemlélteti a 3.3. ábra.

export

Az `export` függvény a modell belső állapotát leíró paramétereket menti el a `Flax` könyvtár `serialization` csomagjának segítségével. Később ezt majd be lehet tölteni és újra felhasználni.

4.3.3. Szegmentáló modell tanítása

A szegmentálást végző modell tanítását a `model/train/segmentation_train.py` script futtatásával tudjuk megtenni. A program először beolvasssa a nyers adatok feldolgozása során előállított `model/train/data.json` fájlt, valamint a tanítás során használt paramétereket tartalmazó `model/train/segmentation_train.yaml` fájlt. Ezután a `Scikit-learn`[16] könyvtár `train_test_split` metódusával véletlenszerűen felbontjuk az adathalmazt tanító- és teszthalmazra. Alapértelmezetten az adatok 80%-a teszi ki a tanítóhalmazt, a maradék 20% pedig a teszthalmazt, de ez a konfigurációs fájl `test_ratio` mezőjében tetszőlegesen módosítható.

Ezután a bemeneti adatokat, valamint a hozzájuk tartozó kimeneteket azonos hosszúságúra egészítjük ki (`padding`). Ennek az az oka, hogy a `Jax` azonos hosszúságú (dimenziójú) adatokkal tud hatékonyan dolgozni. A `padding` okozta hibaszámítási problémákat a `mask_sequences` függvény segítségével fogjuk majd orvosolni.

Ezután a konfigurációs fájban megadott tanítási iterációk száma (`num_epochs`)-szor lefuttatunk egy tanítási iterációt a `training.py` modulban megadott függvények felhasználásával. Közben minden lépésben kiíratjuk a modell hibáját és pontosságát a tanító- és tesztadatokon, valamint a tanítási iterációk végeztével ezeket az értékeket grafikonon is megjelenítjük. Végül elmentjük a betanított modellt.

create_train_state

A modellt a `create_train_state` függvény hozza létre és állítja be a kezdeti paramétereit. A modell leírását a `model/flax_models/segmentation.py` modul tartalmazza.

hiba és pontosság

A modell hibáját a `binary_cross_entropy_loss` függvény határozza meg. Argumentumként megkapja a modell által prediktált értékeket (0.0 és 1.0 közötti érték, `logits`), a tényleges kimeneteket (0 vagy 1, `labels`) és a kimenetek eredeti

hosszait (`lengths`). Ez utóbbi ahhoz kell, hogy az eredeti hossz utáni tévedéseket ne számoljuk hibának. Ezt úgy oldjuk meg, hogy a `mask_sequence` függvény segítségével a prediktált és az elvárt értékeket is az adott hossz után kinullázzuk, így az már nem befolyásolja a hibát, ennek következtében a gradiens módszert sem, vagyis a modell tanulását sem.

A hiba konkrét értékét a prediktált és elvárt értékekből számolt bináris kereszt-entrópia határozza meg. Ez az érték 0, ha a modell minden kimenetet pontosan eltalált, és $+\infty$, ha pont az ellentétét prediktálta valós kimenetnek (0 helyett 1-et, vagy 1 helyett 0-t).

A `compute_metrics` függvény egyrészt a `binary_cross_entropy_loss` metódus hívásával kiszámolja a modell hibáját, másrészt számol egy pontosságot, vagyis, hogy az esetek hányad részében prediktálta helyesen a kimeneti tokent. Itt, ha a modell által prediktált érték 0.5 alatti, akkor azt 0-nak tekintjük, különben 1-nek.

4.3.4. Fordító modell tanítása

A maszkolt C kód előállítását végző modell tanítását a `model/train/translation_train.py` script futtatásával tudjuk megtenni. A tanítási folyamat alapvetően megegyezik a szegmentáló modell tanításánál látottakkal, a hibát és a pontosságot számító modellek működésében van eltérés. A fordító modell hibáját a `cross_entropy_loss` függvény határozza meg, ez egy kereszt-entrópiát számol a modell által prediktált tokenek valószínűségeiből és a tényleges kimenetből. Valamint a `mask_sequences` függvény segítsével figyelmen kívül hagyja a `padding`-ből származó, a kimenet eredeti hosszán túl lévő hibákat.

A `compute_metrics` metódus egyrészt kiszámolja a modell hibáját a `cross_entropy_loss` metódus segítségével, másrészt megadja, hogy az esetek hány százalékában találta el a modell teljes egészében az elvárt kimenetet. Ennek a metrikának a hátránya, hogy csak a tökéletes működést fogadja el, ugyanúgy kezeli, ha csak egyetlen tokent tévesztett el a modell és azt, ha teljesen elrontotta a kimenetet.

4.4. A kódvisszafejtés folyamata

A bevezetésben leírtaknak megfelelően a kódvisszafejtés három lépésben történik. Ezek közül az első kettő használ neurális hálókat, az ezért felelős modulok a

`model/flax_models` csomagban vannak.

A harmadik lépésben az előállított maszkolt C és az eredeti Assembly segítségével rekonstruáljuk, hogy az eredeti C kódban milyen számliterálok voltak, illetve, hogy mikor hivatkoztunk ugyanarra a változóra. Az ezért felelős kódrészlet a `model/reconstruct` csomag `reconstruct.py` moduljában található.

A program mögöttes logikájáért a `model/model.py` modulban található `Decompiler` osztály felelős. A felhasználói felület ezen osztály metódusait hívja és jeleníti meg az eredményt. A következő metódusokat tartalmazza az osztály:

- konstruktor (`__init__`): A konstruktor paraméterül egy szótárat kap, ami a a `model_config.yaml` fájl által megadott konfigurációs paramétereket tartalmazza. A következő adattagok kerülnek inicializálásra:
  - `self.palmtree`: a `utils.py` modulban található `load_palmtree` függvény segítségével betöltjük az Assembly beágyazását végző `Palmtree` modellt.
  - `self.segmentation`: A `flax_models/segmentation.py` modulban található `Segmentation` osztályt példányosítjuk.
  - `self.translation`: A `flax_models/translation.py` modulban található `Translation` osztályt példányosítjuk.
- `open_binary_file`: Paraméterül egy bináris állomány elérési útvonalát kapja, majd ebből visszafejti az Assembly kódot, amit be is ágyaz a `palmtree` segítségével, ezeket az osztály `self.asm` és `self.asm_embeddings` adattagjaiba menti.
- `Decompile`: Ez az osztály fő metódusa, ez felelős a kódvisszafejtésért. Először a `segmentation` modell segítségével előállítja az Assembly blokkokat, majd a `translation` modell segítségével a maszkolt C kódot. Ezután az Assembly blokkokból és a maszkolt C kódból a `reconstruct.py` modul `retrieve` metódusának segítségével előállítja az eredeti C kódot. A részlépések eredményeit elmentjük az osztály adattagjaiba.
- Getterek: A `get_segmentad_asm`, `get_masked_c` és `get_reconstructed_c` metódusok a kódvisszafejtés részlépéseinek eredményeit adják vissza a felhasználói felületen megjeleníthető formátumban.

4.4.1. Assembly szegmentálás

Az Assembly blokkok előállításáért a `flax_models/segmentation.py` modul `Segmentation` osztálya felel. Az osztály az alábbi paramétereket kapja meg a konstruktorában:

- `params_file`: A betanított szegmentáló modell paramétereit tartalmazó fájl elérési útvonala.
- `model`: A `SegmentationModel` osztály egy példánya, ez fogja végezni a tényleges szegmentálást.
- `max_len`: a kimenet (és a bemenet) lehetséges maximális hossza.
- `embedding_size`: A beágyazás során előállított vektor mérete, alapértelmezetten 128.

A `max_len` és az `embedding_size` a modell paramétereinek inicializálásához szükéges, ami a betanított modell paramétereit tartalmazó fájl betöltéséhez kell.

A `get_segmentation` metódus állítja elő a szegmentálást. A `model`-en átfuttatva minden Assembly sorra kapunk egy 0 és 1 közötti értéket. Ezután úgy tekintjük, hogy ha ez kisebb, mint 0.5, akkor 0-nak vesszük, egyébként 1-nek. Ezután a későbbi egyszerűbb számolás érdekében azon sorok indexeit adjuk vissza, amihez 1-es címkét rendeltünk.

4.4.2. Maszkolt C előállítása

A maszkolt C kód előállításáért a `flax_models/translation.py` modul `Translation` osztálya felel. Az osztály az alábbi paramétereket kapja meg a konstruktorában:

- `params_file`: A betanított fordító modell paramétereit tartalmazó fájl elérési útvonala.
- `model`: A `Seq2seq` osztály egy példánya, ez fogja végezni a kimeneti tokenek előállítását.
- `vocab`: A lehetséges kimeneti tokeneket és a hozzájuk rendel indexeket tartalmazó szótár.

- `max_input_len`: A bemenet lehetséges maximális hossza.
- `embedding_size`: A beágyazás során előállított vektor mérete, alapértelmezetten 128.

Ezután a konstruktorban betöltjük a betanított modell paramétereit tartalmazó fájlt a `Segmentation` osztály konstruktorához hasonlóan.

A tényleges kimeneti tokenek előállítását a `translate` metódus végzi, ez paraméterként az Assembly blokkok beágyazott változatát kapja meg. A hatékony működés érdekében ezen blokkokat kiegészítjük 0-kal, hogy azonos hosszúságúak legyenek. Ezután a `model` által prediktált legnagyobb valószínűségú kimeneti tokent választjuk. Mivel a háló által adott kimeneti tokenek száma fix hosszúságú, ezért az első `;` után lévő tokeneket eldobjuk, majd a maradékot space-ekkel összefűzve visszaadjuk.

4.4.3. A változók és a számliterálok rekonstruálása

A fordítható C kód előállításáért a `model/reconstruct` csomag felelős. Ebben a csomagban található a tényleges számításokat végző `reconstruct.py` modul, valamint három fájl (`div_magic.json`, `mod_magic.json`, `mul_magic.json`), amik a bevezetőben leírt osztás, modulozás és szorzás során felmerülő komplikációk megoldásában segítenek. -100-tól 100-ig minden számra tartalmazzák, hogy ha az adott számmal osztunk (szorzunk, modulozunk), akkor az Assembly kódban milyen számliterálok jelennek meg. Ezeket a program futása során is elő lehetne állítani, ugyanakkor gyorsabb előre legenerálni és csak egy fájlból betölteni.

A visszafordítás során a `model/model.py` modulban a `Decompiler` osztály a `retrieve` függvényt hívja, ami a neurális háló által előállított Assembly blokkokból és maszkolt C kódból rekonstruálja, hogy az eredeti C kódban hol melyik számliterál és változó szerepelt. A függvény működése a következő:

1. Vesszük a következő sor maszkolt C kódot és a hozzá tartozó Assembly blokkot.
2. Az Assembly blokkból kinyerjük a változókat és a számliterálokat.
3. A kinyert változóknak vesszük a következő permutációját és behelyettesítjük a `VAR` tokenek helyére.

4. A kinyert számoknak vesszük a következő permutációját és behelyettesítjük a `NUM` tokenek helyére.

5. Az így kapott programot lefordítjuk, majd összehasonlítjuk, hogy az abból kinyert Assembly egyezik-e a paraméterül kapott Assembly blokkok megfelelő prefixével.

6. Ha igen, akkor vesszük a következő sort és blokkot, különben a következő permutációt.

A változók kinyerését az `extract_vars` függvény végzi. Ez az Assembly blokkon túl kap egy szótárat is, amiben azt tartjuk számon, hogy egy adott regiszter melyik változóra hivatkozik. Ha új regiszterrel találkozik a függvény, akkor a legkisebb olyan xi nevet adja a változónak, amilyen i még nem volt ($i \geq 0$).

A számliterálok kinyerését az `extract_nums` függvény végzi. Itt először a regiszterek offszeteit kell maszkolni, mert azokat is számnak vennénk különben. Ezután minden hexadecimális ($0x[0-9a-f]+$ alakú) számot kikeresünk, majd ezeket tízes számrendszerbe váltjuk, ez utóbbiért az `s16` függvény felelős.

Az `extra_nums` függvény a kinyert számokhoz előállítja a lehetséges extra számokat a `*_magic.json` fájlok alapján. Például, ha osztunk és az Assembly-ből kinyert számok a $(30841, 32, 3, 31)$, akkor a 17-et fogja visszaadni a függvény, mint "extra szám".

A `compile_and_get_blocks` függvény az iteratívan előállított programrészletet fordítja le, majd állítja elő belőle az Assembly blokkokat. Ezt aztán a `compare` metódus összehasonlítja az eredeti Assembly blokkokkal. Mivel teljes egyezés nem várható el (nem determinisztikus, hogy pontosan melyik regiszterekre fogunk hivatkozni), ezért csak azt vizsgáljuk, hogy egyrészt ugyanolyan hosszú-e a két kód, másrészt hogy a változók és a számok ugyanabban a sorrendben vannak-e.

4.5. Segédprogramok

A `model/utils.py` modul tartalmazza azon általános segédfüggvényeket, amelyek a program több komponensében is felhasználásra kerülnek, de egyiknek sem képezik szerves részét.

4.5.1. Palmtree

A `load_palmtree` függvény végzi az Assembly beágyazását végző `Palmtree` modell betöltését. A függvény egy enkóderrel tér vissza, ennek az `encode` metódusát hívva tudjuk beágyazni a bemeneti Assembly kódot, ezzel könnyítve a neurális háló(k) tanulását. A metódus egy sztringeket tartalmazó listát vár, ahol a lista minden eleme egy megfelelően tokenizált Assembly sornak felel meg. Az adatok előfeldolgozása során fontos, hogy az Assembly sorokat megfelelően tokenizáljuk, ennek hiányában a beágyazás nem fog kellően jól működni. Az egyes tokeneket space-ek választják el, egy megfelelően tokenizált példát tartalmaz az alábbi kódrészlet:

```
mov rbp rdi
mov ebx 0x1
mov rdx rbx
call memcpy
mov [ rcx + rbx ] 0x0
mov rcx rax
mov [ rax ] 0x2e
```

4.5.2. Fájlok betöltése / elmentése

A `save_as_json` függvény egy fájl útvonalat és egy `JSON`[17] formátumúvá alakítható python objektumot vár, majd ezt elmenti a megadott fájlba. A `load_json` függvény pedig egy elmentett `JSON` fájl betöltését végzi, például a megtisztított adatok betöltését.

A konfigurációs fájlok `YAML`[18] formátumban vannak, ezek betöltéséért a `load_yaml` metódus felelős. Ez egy szótárat ad vissza, amiben a kulcsok a konfigurálható paraméterek (pl. `learning_rate`), az értékek pedig a megfelelő paraméter értéke (pl. 0.003).

4.5.3. Fordítás és az Assembly kód kinyerése

Az alábbi függvények elsősorban két helyen vannak használva, egyrészt a nyers adatok feldolgozásánál, valamint a kódvisszafejtés harmadik lépésében, amikor a neurális háló által előállított maszkolt C kódba próbáljuk visszahelyettesíteni a tényleges változókat és számliterálokat.

A `compile` függvény segítségével egy C fájlt tudunk lefordítani. Paraméterként vagy a konkrét fájl elérési útvonalát kell megadni, vagy magát a fájl tartalmát. Továbbá meg kell adni a fordítási kapcsolókat (`O0` optimalizációval és debug módban fordítjuk a kódot) és hogy fájlba mentse a futtatható állományt vagy sztringként adja vissza. Alapértelmezetten szövegként adjuk át a lefordítandó C fájl tartalmát és a futtatható állomány tartalmát is szövegként kapjuk vissza, ennek előnye, hogy nincsenek fölöslegesen ideiglenes fájlok létrehozva.

Ezután az `objdump` bináris elemző eszköz segítségével visszafejtjük a futtatható állományból a megfelelő Assembly kódot és egyéb információkat is, többek között, hogy az egyes blokkok melyik C sornak felelnek meg, ezt azért tudjuk kinyerni, mert a `-g` kapcsolóval (debug mód) fordítottunk.

Az `extract_fun` metódus segítségével nyerhetjük ki az `objdump` metódus által visszaadott sztringből azt a részletet, ami az eredeti C-ben `main` függvénynek feleltethető meg. Ebből aztán az `extract_asm` függvény kinyeri a tényleges Assembly kódot, majd a fentieknek megfelelően a tokenizálást is elvégzi.

Az `extract_labels` függvény a debug módban való fordításból kapott információkat kihasználva elvégzi az Assembly sorok szegmentálását. A függvény nem a blokkokat adja vissza, hanem a tanítás során használt címkéket, vagyis hogy a megfelelő sorra kezdődik-e ott új blokk (1) vagy nem (0).

4.6. Tesztelés

A kód tesztelése alapvetően két módon történt, egyrészt egységtesztekkel, a `pytest`[19] könyvtár segítségével, másrészt típusellenőrzésre is sor került a `Mypy`[20] könyvtár segítségével.

4.6.1. Egységtesztek

Az egységtesztek a moduloknak megfelelően külön fájlokban vannak a `tests` könyvtárban, a `test_<modulnév>.py` konvenciót használva. A neurális hálók tanításának nemdeterminisztikus volta miatt a tesztesetek elsősorban az egyes modulok különböző segédfüggvényeit fedik le.

- `test_features.py`: A nyers adatok feldolgozás során használt függvények tesztelése.

- `test_utils.py`: A futtatható állományból a szükséges adatokat (Assembly kód, címkék) előállító függvények, valamint a `JSON` fájlok elmentését és betöltését végző metódusok tesztelése.
- `test_segmentation.py`, `test_translation.py`: Annak ellenőrzése, hogy a neurális hálók által visszaadott tenzorok megfelelő dimenziójúak-e.
- `test_segmentation_train.py`, `test_translation_train.py`: A tanítás során használt metódusok (`padding`, metrikafüggvény, kimenet maszkolása) tesztelése. Továbbá annak ellenőrzése, hogy egy tanítási lépés során változnak-e a modell belső állapotát leíró paraméterek.
- `test_train.py`: A folyamatjelző sáv működésének tesztelése.
- `test_reconstruct.py`: A program harmadik lépésének (az Assembly blokkokból és a maszkolt C kódból az eredeti C kód rekonsruálása) tesztelése.

A teszteket a `pytest` parancs kiadásával tudjuk lefuttatni a gyökér könyvtárból. A tesztelő beállításait a `pytest.ini` konfigurációs fájl tartalmazza.

4.6.2. Típusellenőrzés

A `Python` egy dinamikus típusozású nyelv, ugyanakkor lehetőség van típus annotációk megadására. Ezek a programot nem befolyásolják sem működésében, sem hatékonyságában, ugyanakkor a programozó számára nagy segítséget nyújthatnak a kód értelmezésében. Az ellenőrzést a `mypy` parancs kiadásával tudjuk elvégezni a gyökér könyvtárból. A beállításokat a `mypy.ini` fájl tartalmazza, itt meg lehet adni többek között, hogy mennyire legyen szigorú az ellenőrzés, illetve hogy egyes modulokat ugorjunk át az ellenőrzés során.

5. fejezet

Összegzés

Dolgozatomban egy olyan programot mutattam be, melynek segítségével lehetőség van a bináris állományból az eredeti C kód visszaállítására neurális hálók segítségével. A kódvisszafejtés három fő lépésben történik, ezek közül az első kettő használ neurális hálókat.

Előkészítésként a futtaható állományból egy bináris elemzővel kinyerjük az Assembly kódot. Ezután az első lépés, hogy ezt blokkokra bontjuk, ahol egy blokk egy sor C kódnak felel meg, itt a blokkok meghatározását végzi egy neurális háló. Második lépésként ezen blokkokból előállítunk egy sablon kódot, ahol a változók és a számliterálok konkrét értékei még nem ismertek. Ezen lépés hasonlít leginkább a neurális gépi fordítás során használt modellekre. Végül harmadik lépésként az Assembly-ből kinyert változókat és számokat behelyettesítjük a második lépésben előállított maszkolt C kódba, így visszakapjuk az eredeti C kódot.

A program futtatása során lehetőség van a fentiek kipróbálására grafikus és konzolos felületen is, valamint a neurális hálók tanítására saját adatokkal.

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] Jacob Devlin és tsai. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2018. DOI: 10.48550/ARXIV.1810.04805. URL: https://arxiv.org/abs/1810.04805.

[4] GNU Project. *GNU Binary Utilities*. https://www.gnu.org/software/binutils/. 2022.

[5] Sepp Hochreiter és Jürgen Schmidhuber. „Long Short-term Memory". *Neural computation* 9 (1997. dec.), 1735–80. old. DOI: 10.1162/neco.1997.9.8.1735.

[6] Tomas Mikolov és tsai. *Efficient Estimation of Word Representations in Vector Space*. 2013. DOI: 10.48550/ARXIV.1301.3781. URL: https://arxiv.org/abs/1301.3781.

[7] 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*. ACM, 2021. DOI: 10.1145/3460120.3484587. URL: https://doi.org/10.1145%2F3460120.3484587.

[8] Matteo Hessel és tsai. *Optax: composable gradient transformation and optimisation, in JAX!* 0.0.1. verzió. 2020. URL: http://github.com/deepmind/optax.

[9] Diederik P. Kingma és Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

[10] PySimpleGUI. *PySimpleGUI*. https://pysimplegui.readthedocs.io/en/latest/. 2018.

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

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

[13] Charles R. Harris és tsai. „Array programming with NumPy". *Nature* 585.7825 (2020. szept.), 357–362. old. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[14] GNU Project. *GNU Compiler Collection*. https://gcc.gnu.org/. 1987.

[15] J. D. Hunter. „Matplotlib: A 2D graphics environment". *Computing in Science & Engineering* 9.3 (2007), 90–95. old. DOI: 10.1109/MCSE.2007.55.

[16] F. Pedregosa és tsai. „Scikit-learn: Machine Learning in Python". *Journal of Machine Learning Research* 12 (2011), 2825–2830. old.

[17] Douglas Crockford. *JavaScript Object Notation*. https://www.json.org/. 2001.

[18] Ingy döt Net, Clark Evans és Oren Ben-Kiki. *YAML Ain't Markup Language*. https://yaml.org/. 2021.

[19] Holger Krekel és tsai. *pytest 7.1*. https://pytest.org/. 2004.

[20] Jukka Lehtosalo és tsai. *Mypy*. http://mypy-lang.org/. 2012.

Ábrák jegyzéke