

TDK-dolgozat

Kovács Gergely
Zsolt

Teleki Sándor

Szalay Gergő

Neurális kódvisszafejtés rekurzív szegmentálással

EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

PROGRAMOZÁSELMÉLET ÉS SZOFTVERTECHNOLÓGIAI TANSZÉK



Szerzők:

Kovács Gergely
Zsolt
programtervező
informatikus MSc.
2. évfolyam

Teleki Sándor
programtervező
informatikus MSc.
1. évfolyam

Szalay Gergő
programtervező
informatikus MSc.
2. évfolyam

Témavezetők:

Pintér Balázs
egyetemi adjunktus

Gregorics Tibor
tanszékvezető egyetemi docens

Budapest, 2023. december 3.

Tartalomjegyzék

1. Bevezetés	1
1.1. Kontribúcióink	4
1.2. Gépi tanulás, mélytanulás	4
1.3. Kapcsolódó munkák	9
2. Módszerek	14
2.1. Adatgenerálás, adathalmazok	14
2.2. Alacsony szintű kód darabolás és párosítás	15
2.3. Előfeldolgozás	16
2.3.1. Az eredeti C kód optimalizációs transzformációja	19
2.3.2. Az assembly kód előállítás	20
2.3.3. Változó-memóriacím megfeleltetés	21
2.3.4. A szegmentációs példák előállítás	22
2.3.5. A kódvisszafejtési példák előállítás	22
2.3.6. Utólagos műveletek az adathalmazon	23
2.4. Modellek	25
2.4.1. Szegmentáló modell	25
2.4.2. Kódvisszafejtő modell	26
2.5. C kód előállítás	28
3. Kiértékelés	29
3.1. Mérőszámok	29
3.2. Tanító adathalmaz és tanítási paraméterek	30
3.3. Eredmények	30
3.3.1. Szegmentálás	31
3.3.2. Kódvisszafejtés	31
3.3.3. Kódvisszafejtés több modellel	32
3.3.4. Valós példák	33

4. Diskusszió	35
5. Összefoglalás	37
Irodalomjegyzék	39

Kivonat

A kódvisszafejtés (decompilation) feladata alacsony szintű kódot visszaalakítani vele ekvivalens magas szintű kóddá. Mivel a fordító által létrehozott kódban a nyelv magas szintű információinak nagy része elveszik vagy átalakul, nagy kihívást jelent kódvisszafejtés során az, hogy a visszafejtett kód idiomatikus, ember által is olvasható legyen. Hagyományos kódvisszafejtő eszközökkel máig sem sikerült megoldani ezt a feladatot (ezen eszközök kimenetei gyakran az alacsony szintű kód működését utánozzák magas szintű nyelvi elemekkel, ezért sokszor nehezen értelmezhetők), így többen a gépi tanuláshoz fordultak.

Ebben a dolgozatban a korábbi munkánk továbbgondolását mutatjuk be. Előző munkánkban a másoló mechanizmus használatával megoldást nyújtottunk a literálok és azonosítók kezelésére. Jelenlegi munkánk során kidolgoztuk a *rekurzív szegmentálás* technikáját, mely a bemeneti alacsony szintű kódot kisebb, könnyebben visszafejthető részekre darabolja. A technikánk képes ezeket a kóddarabokat szükség szerint rekurzívan tovább-bontani. Ennek alapján a kódvisszafejtő rendszerünkben alkalmazunk egy újabb neurális hálót, mely elvégzi a szegmentálást. Így a kódvisszafejtést továbbra is egy *single-pass* módszerrel oldjuk meg.

Az alacsony szintű szerkezeti információk hasznosítása érdekében egy új enkódolót hoztunk létre, ami csökkenti a két nyelv eltérő információsűrűségéből adódó aszimmetriát is. Újabb nyelvi elemek visszafejtését is támogatjuk a korábbiakhoz képest: elől tesztelő (*do-while*) ciklusok, *augmented assignment*ek (pl. $+=$, $-=$) visszafejtése. Emellett elsőként kezeltük a hosszú switch utasítások esetén keletkező *jump table* szerkezetek visszafejtését.

1. fejezet

Bevezetés

Kódvisszafejtésről (vagy más néven *decompilation*ról) akkor beszélhetünk, amikor egy gépi kódhoz közeli nyelven írt programkódot alakítunk át magasabb absztrakciós szintű programozási nyelven írt kódra. Ennek különböző felhasználási területei például a rosszindulatú vagy hibás szoftverek felismerése és kiszűrése, a zárt forráskódú programok működésének megismerése, illetve az elveszett vagy sérült forráskód visszaalakítása.

A kódvisszafejtés feladata a fordítóprogramok megjelenésével egyidejűleg vetődött fel, mégis közel tíz évet kellett várni, mire az első kódvisszafejtő alkalmazások megjelentek. Az alapfeladat komplexitását a fordítás során bekövetkező absztrakciósszint-csökkenés okozza: elvesznek a magas szintű vezérlési szerkezetek és az azonosítók. Habár a gépi kód magas absztrakciós szintben való imitálása megoldható, a kódvisszafejtés teljes körű megoldása, idiomatikus kód visszaalakítása napjainkig is egy nyitott probléma.

Korábban több különböző programot is készítettek, melyekkel a felvetett feladatot szerették volna megoldani. Ezek közül a korai próbálkozások mind statikus (gépi tanulást nem alkalmazó) megközelítések voltak. Ide sorolhatóak az IDA Pro¹, a RetDec², a Radare2³, a Ghidra⁴. Tekintve, hogy az említett programok fejlesztéséhez komoly erőforrásokat kell mozgósítani (fordítóprogramok terén jártas tárgyszakértők, többéves fejlesztőmunka), így kijelenthető, hogy a kódvisszafejtő programok fejlesztése jelentős költségekkel jár. Emellett a legtöbb ilyen program csak bizonyos fordítóprogramokat és azoknak is csak egyes verzióit támogatja, így egy újabb fordí-

¹<https://hex-rays.com/ida-pro/> 2022

²<https://retdec.com/> 2022

³<https://rada.re/n/> 2022

⁴<https://ghidra-sre.org/>

tóprogram vagy verzió támogatása jelentős kiadásokkal járhat. Ezen felül az említett munkák közül a legtöbb nem generál idiomatikus kódot, azokban jellemző az ugrások intenzív használata, valamint a regiszterimitáció.

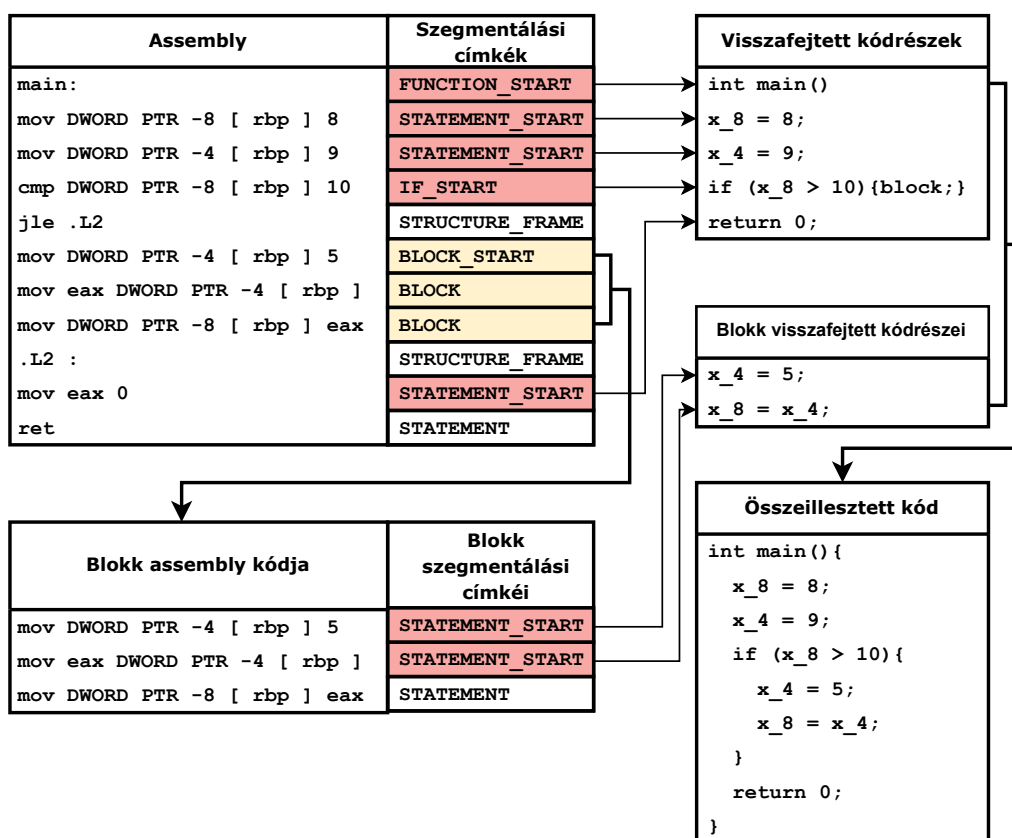
A kódvisszafejtés tekinthető két nyelv közötti szövegátalakításként, így érthető, hogy a természetes nyelvek esetén használt eszközök előtérbe kerültek. A természetes nyelv feldolgozás (Natural Language Processing, NLP) során emberi nyelven megfogalmazott szövegekkel dolgozunk. Ebbe a kategóriába tartozik a gépi fordítás (Neural Machine Translation, NMT), a szövegösszegzés, -generálás és -feldolgozás további területei. Az elmúlt évtizedekben a gépi tanulás (Machine Learning, ML) kimagasló sikereket ért el ezen a téren, típusfeladatává vált az emberi nyelvek kezelése. Különösképpen kiemelkedő a gépi fordítás feladata, azon belül is a francia-angol fordítás, melyben a mélytanulás (Deep Learning, DL) a legsikeresebb megközelítés. Hasonlóan a természetes nyelvekhez, a programnyelveknek is van egyfajta alapszókincse és nyelvtana, emellett az egyes nyelvek által kifejezhető gondolatok átvihetőek a másik nyelvre. Érthető, hogy a természetes nyelvek fordításában és feldolgozásában sikeres eszközök kerültek előtérbe. Ennek nyomán több kutatás is született a témában.

Ezen területen az első munkában [1], melyet a továbbiakban alapmunkának tekintünk, egy egyszerű *seq2seq* architektúrával próbálták megoldani a feladatot, viszont több nehézség is megmutatkozott. Az első javítások [2, 3] különböző hibajavító módszereket és köztes reprezentációkat vezettek be. Előbbieket a literálok és az azonosítók gyakran helytelen kezelése, utóbbiakat pedig a szigorú nyelvi szabályok indokolták. Az eredmények javulásával az általánosabb *seq2seq* problémák kezelése is előtérbe került. A hosszlimitáció problémájára az assembly kód szegmentálása [4] nyújtott megoldást, ha egyből nem is teljes sikerrel. Látván a *transformer* architektúra sikerességét az NMT terén, többen is ebbe az irányba fordultak [5, 6].

Korábbi munkánk során [7] megmutattuk, hogyan lehet hatékonyan kódvisszafejtést végezni egyetlen neurális hálóval, utólagos javítások nélkül. A literálok és változók hatékony kezelése érdekében a másoló mechanizmussal (**copying mechanism**) [8] egészítettük ki a modellünket.

Ezen munkánk viszont még nem alkalmazta a darabolás nyújtotta lehetőséget, így erősen függött a bemenet hosszától a modell sikeressége. Az egyszerű szegmentálás, mely során csupán egyszer megyünk végig a kódon, majd daraboljuk fel azt, viszont a kód komplexitásának növelésével csökkentheti a hatékonyságát: a vezér-

lési szerkezetek által közrefogott kódot nem lehet egyetlen bejárással elkülöníteni a vezérlési szerkezettől, így egyetlen nagy egységet fog továbbra is képezni. Ennek érdekében jelenlegi munkánk során egy új megközelítést alkalmazunk a szegmentálásra. A szegmentálás során a vezérlési szerkezetek belső blokkjait egy egységként kezeljük, melyet a bejárást követően külön feldarabolunk, ezáltal megoldva a komplexitás növelésével megjelenő nagyobb kódrészleteket. Ennek működését a 1.1 ábra szemlélteti.



1.1. ábra. Egy példa a rekurzív szegmentációval történő kódvisszafejtésre. Az assembly kódot kisebb részekre bontjuk, és a vezérlési szerkezetek belső blokkjait rekurzívan tovább daraboljuk. A kisebb részeket külön-külön visszafejtjük C kóddá, majd a visszaalakított kódrészleteket összeillesztve megkapjuk a teljes C kódot.

A szegmentálásnak köszönhetően a kódvisszafejtést végző komponensnek egy rövidebb alacsony szintű kódrészt kell egy magas szintű utasításra visszaalakítani. Itt megmutatkozik az absztrakciós szintbeli különbség, miszerint egy hosszabb assembly kódblokkot kell mindössze egy sor C utasítássá alakítani. Ezt ellensúlyozandó, az alacsony szintű kód feldolgozását a korábbi munkákkal ellentétben nem csak tömbösítve végeztük. A rendszer kiegészítéseként soronként is feldolgoztuk a bemenetet, így annak szerkezete (utasítás, paraméterek) egyértelműbbé vált a modell számára.

Korábbi munkánk fejlesztéseként több új nyelvi elem támogatására is felkészítettük a rendszerünket.

1.1. Kontribúcióink

Az alábbiakban részletesen bemutatjuk munkánk kontribúcióit:

1. A rekurzív szegmentálás módszerének bevezetése, melynek segítségével jelentősen csökkenthető az egyben feldolgozandó bemenet hossza a függvények, a vezérlési szerkezetek és a vezérlésfolyam megtartása mellett (az 1.1 ábrán egy példa szemlélteti ennek működését).
2. Új nyelvi elemek támogatása: elől tesztelő (*do-while*) ciklus, *augmented assignment*ek (pl. $+=$, $-=$), nagy méretű switch utasítás, ami az assembly kódban *jump table*-t eredményez.
3. Az absztrakciós szintekből adódó különbségek modell szintű kompenzálása: a bemeneti alacsony absztrakciós szintű nyelv tokenenkénti feldolgozása mellett annak soronkénti feldolgozásával bővített encoder alkalmazása.
4. Sikerült felülmúlnunk a korábbi eredményeket mind szintetikus, mind valós példákon. Utóbbinál 9,2 százalékponttal magasabb tökéletes visszaalakítási arányt értünk el.

1.2. Gépi tanulás, mélytanulás

A mélytanulás egy olyan gépi tanulási paradigma, amely a mesterséges neurális hálózatok többrétegű architektúráját alkalmazza különböző feladatokban, mint például beszédfelismerés, fordítás, képosztályozás vagy tartalomgenerálás, használata azonban mostanra már elterjedt programkódok manipulálására is.

A mélytanulás gyökerei az 1940-es években kezdődő neurális hálózatok kutatásához nyúlnak vissza, amelyek az emberi agy struktúráját és funkcióját próbálták leutánozni. [9] A komplex rendszer egyszerű alapegységekből (neuronokból) épül fel, amelyek rétegekbe rendezve és egymással összekapcsolva bonyolult feladatok megoldására képesek. Ezen kapcsolatok súlyozottak, amiket tanulási folyamat során az elért teljesítmény alapján optimalizálunk. A neurális hálózat rétegei három csoportba sorolhatók: a bemeneti réteg fogadja az adatokat, amelyeket továbbít a rejtett rétegeknek, amik feldolgozzák azokat, ami alapján a kimeneti réteg elkészíti a modell eredményét.

A rétegek belső felépítése és összekapcsolásuk módja a következő matematikai formulákkal írható le:

- Legyen x_i az i -edik bemeneti jel, w_{ij} az i -edik bemenet és a j -edik neuron közötti kapcsolat súlya, b_j a j -edik neuron küszöbértéke (*bias*), a_j a j -edik neuron aktivációs értéke, f az aktivációs függvény, és y_j a j -edik neuron kimeneti jele. Ekkor az l . rétegben a neuronok aktivációs értéke és kimeneti jele a következőképpen számítható:

$$a_j^{(l)} = \sum_i w_{ij}^{(l)} x_i^{(l)} + b_j^{(l)}$$

$$y_j^{(l)} = f(a_j^{(l)})$$

- A rétegeket olyan módon kapcsoljuk össze, hogy azok kimeneti jelei a következő bemenetei lesznek:

$$x_i^{(l+1)} = y_i^{(l)}$$

- A neurális hálózat kimenete megegyezik a legutóbbi rétegével:

$$\mathbf{y} = \mathbf{y}^{(L)}$$

A számolásokat azonban valójában nem ezen a módon implementáljuk, mivel az értékeket vektorizált formában számolva nagyon jelentős teljesítménynövekedés érhető el.

Legyen $\mathbf{x}^{(l)}$ az l . réteg bemeneti vektora, $\mathbf{W}^{(l)}$ az l . réteg súlymátrixa, $\mathbf{b}^{(l)}$ az l . réteg küszöbértékvektora, $\mathbf{a}^{(l)}$ az l . réteg aktivációs vektora, $\mathbf{y}^{(l)}$ az l . réteg kimeneti

vektora, és f az aktivációs függvény. Ekkor az l . rétegben a neuronok fentiekkel megegyező aktivációs és kimeneti értéke vektorizáltan a következőképpen számítható:

$$\mathbf{a}^{(l)} = \mathbf{W}^{(l)} \mathbf{x}^{(l)} + \mathbf{b}^{(l)}$$

$$\mathbf{y}^{(l)} = f(\mathbf{a}^{(l)})$$

Itt a jelölés az aktivációs függvény elemenkénti alkalmazását jelenti.

Több réteg egymás utáni alkalmazása esetén a rétegek közötti kapcsolatokat a következőképpen írhatjuk fel:

$$\mathbf{y}^{(l+1)} = f(\mathbf{W}^{(l+1)} \mathbf{x}^{(l+1)} + \mathbf{b}^{(l+1)}) = f(\mathbf{W}^{(l+1)} f(\mathbf{W}^{(l)} \mathbf{x}^{(l)} + \mathbf{b}^{(l)}) + \mathbf{b}^{(l+1)})$$

Itt is természetesen egy réteg kimenete a következő bemenete: $\mathbf{y}^{(l)} = \mathbf{x}^{(l+1)}$

A neurális hálózat tanulása során a kapcsolatok súlyait és a neuronok küszöbértékeit úgy módosítjuk, hogy minimalizáljunk egy veszteségfüggvényt, amely a hálózat kimenete és az elvárt kimenet közötti eltérést méri. A leggyakoribb tanulási algoritmus a hiba-visszaterjesztés (backpropagation), amelynek lényege, hogy először kiszámoljuk a veszteségfüggvény gradiensét minden súlyra és küszöbértékre vonatkozóan (lényegében megkeresve a hiba irányát), majd ezekkel arányosan frissítjük őket.

Hogy mindebből hogyan válik lehetségessé komplex összefüggések megtanulása, arra a képfelismerés intuitív illusztrációként szolgálhat (még ha tökéletlen és leegyszerűsített is az analógia). Itt gondolhatunk arra, hogy számos réteg mindegyike egyre magasabb szintű jellemzőket és reprezentációkat tanulhat meg az adatokról. Például egy képfelismerő hálózat első rétege megtanulhatja észlelni az alapvető formákat, mint a vonalak és a görbék, a második az ezekből felépülő egyszerűbb síkidomokat, mint a körök és a négyszögek, majd a bonyolultabb alkotóelemeken keresztül végül olyan komplex formákat, mint például arcok, kezek, autók.

Szekvencia-szekvencia transzformáció

Jelen munka azonban szöveges adatokkal dolgozik, amelyek kezelésére jól bevált mód a **szekvencia-szekvencia transzformáció** (*seq2seq*), egy olyan gépi tanulási architektúra, amely egy adott sorozatot egy másikká alakít. Ezek jellege, hossza és

elemi egységei is számos formában elképzelhetők, például egy lefordítandó szöveg szavakból, szótagokból vagy karakterekből állhat, egy diktálás hangfelvétele amplitúdóértékekből. A modell célja, hogy megtanulja a bemeneti és a kimeneti sorozat közötti kapcsolatot, és új adatokon a későbbiekben a transzformáció önálló elvégzésére képes legyen.

Ezen feladat leggyakoribb megközelítése az **encoder-decoder architektúra**, amely két neurális hálózatot használ: egy enkódolót, amely a bemeneti sorozatot egyetlen vektorra alakítja, és egy dekódolót, amely ebből a vektorból előállítja a kimenetet. Ezt a kettő közötti vektort kontextusvektornak nevezzük, ami tömörített formában tartalmazza az átalakítás elvégzéséhez szükséges információkat.

Mindkét részfeladat elvégzésére alkalmasak az **RNN**-ek (*recurrent neural networks*, [10]), amik olyan neurális hálózatok, amelyek a szekvenciális adatokat egységként, ugyanazon lépéseket ismételve dolgozzák fel. A lépések közben a cellák az új bemeneti adatok mellett a korábbi időpillanatokból származó információkat (pontosabban az azokból számolt rejtett állapotot) is figyelembe veszik, így képesek megőrizni az adatok kontextusát és időbeli függőségét. Amennyiben fontos, hogy adott időpillanatban később következő elemek is ismertek legyenek, használható kétirányú **RNN**, ami a sorozatot párhuzamosan fordított irányban is feldolgozza. A hosszú távú kapcsolatok azonban a rejtett állapot korlátozott mérete és az újabb bemenetek korábbiakat szinte felülíró hatása miatt gondot okoznak. Ezt nevezik eltűnő gradiens jelenségnek, ami azt jelenti, hogy a gradiensek, amelyek a hibafüggvény szerinti változásokat mutatják, nagyon kicsivé, szinte elhanyagolhatóvá válnak, ahogy visszaterjesztés során haladunk visszafelé. Ennek következtében a hálózat nem tud tanulni a régebbi bemenetektől, és elveszíti a hosszú távú memóriáját.

LSTM és GRU

Az **LSTM**-ek (*long short-term memory*, [11]) olyan speciális **RNN**-ek, amelyek ezen probléma kezelésére lettek kifejlesztve. A rejtett állapotot az információk hosszú távú tárolására teszi képessé azzal, hogy szabályozza azok áramlását. Ez három kapu segítségével történik: egy bemeneti kapu, amely eldönti, hogy milyen új információt vegyen fel a sejt; egy felejtőkapu, amely eldönti, hogy milyen régi információt dobjon el a sejt; és egy kimeneti kapu, amely eldönti, hogy milyen információt továbbítson a sejt a következő időlépéshez.

A teljes modell megépítésekor ezeket az egységeket kapcsoljuk össze. A kódoló futása során minden lépésben egy új bemeneti tokent adunk a cellának, ami ennek hatására frissíti a rejtett állapotát, amit tovább ad majd a következő lépéshez. A bemenet végével ez a vektor fogja tartalmazni az összes információt, ami a dekódoló számára rendelkezésre áll. A dekódoló hasonlóan működik, itt a rejtett állapot kezdőértéke a kontextusvektor, ami alapján tokenenként generálja a kimenetet. (Megjegyzendő még, hogy itt bemenetként szokás a megelőző generált értéket megadni.)

A **GRU**-k (*gated recurrent unit*, [12]) az **LSTM**-hez hasonló, a hosszútávú memóriát elősegítő **RNN**-ek, amelyek kevesebb paraméterrel képesek legtöbbször azonos szintű eredmény elérésére. Felépítésük egyszerűbb, mivel csak két kaput használnak: egy frissítőkaput, amely meghatározza, hogy az előző állapotot vagy inkább az új bemenetet szerepeljen nagyobb súllyal; és egy visszaállítókaput, amely eldönti, hogy az előző állapot mennyire kerüljön feledésbe.

Attention

Az LSTM hálók azonban nem oldottak meg minden nehézséget a hosszú távú memória használatával. Az **attention mechanizmus** [13] tovább javítja a modell teljesítményét azzal, hogy lehetővé teszi, hogy szelektíven figyeljen a bemeneti adatok releváns részeire. Az attention mechanizmus az encoder rejtett állapotait adott kimeneti elem előállításához tartozó relevanciával súlyozva összesíti, majd a decodernek átadja. Ezzel jelentősen megnöveli a bemeneti és kimeneti tokenek feldolgozása, létrehozása közötti kapcsolatot, hiszen már nem csak a kontextusvektor, de az encoder rejtett állapotai is rendelkezésre állnak a decoder számára.

Másolás

Az encoder-decoder architektúra, mivel előre meghatározott nyelvi elemekre (tokenekre) épít, nem képes hatékonyan kezelni a ritka vagy új szavakat, amelyek nem szerepelnek a modell szókincsében, de pontosan meg kell őrizni a kimeneten (jelen esetben például számliterálok). Ennek megoldására javasolták a **CopyNet** [8] architektúrában a másoló mechanizmust, amely lehetővé teszi különböző tokenek közvetlen másolását a bemeneti sorozatból a kimeneti sorozatba.

A **CopyNet** ezt az encoder-decoder struktúrába integráltan valósítja meg. Egységesen kezeli a decoderben történő megszokott tokengenerálást és az új másolási mechanizmust, amely kiválaszthatja a bemeneti sorozatban lévő tokeneket és elhelyezheti őket megfelelő helyeken a kimeneti sorozatban. A rendszer működése a következőképpen írható le:

A kódoló egy kétirányú RNN vagy LSTM hálózat, amely feldolgozza a bemeneti sorozatot és minden tokenhez eltárolja a rejtett állapotvektort egy M mátrixba (az attention mechanizmushoz hasonlóan). A dekódoló szintén egy RNN vagy LSTM hálózat, amely minden időlépésben két valószínűségi eloszlást számít ki: egy generálási eloszlást és egy másolási eloszlást. A előbbi a modell szókincsében szereplő tokenekhez, az utóbbi, a bemenetben szereplőkhöz rendel egy valószínűséget. A kettő együttesen határozza meg a kimeneti tokenek valószínűségét a generálási és másolási valószínűség összegeként. További különbség az egyszerű dekódolóval szemben, hogy minden időpillanatban az előző generált kimenet értéke mellett a memóriában hozzá tartozó rejtett állapot is figyelembe van véve, mivel a tartalom mellett a pozíció (és az abból adódó, memóriában eltárolt kontextus) is releváns információ lehet.

1.3. Kapcsolódó munkák

A gépi tanulás alapú kódvisszafejtés feladatát D. S. Katz és tsi. [1] munkássága alapozta meg. Munkájuk során egy *seq2seq* hálót tanítottak be bináris állományból C forráskódot létrehozni. A bemenetet nyers bináris állomány jelentette, melyet többféle tokenizálási módszerrel is próbáltak feldolgozni, végül a leghatékonyabbnak a bájttonkénti darabolás bizonyult. Emellett a bemenetet fordított sorrendben adták a modellnek, hogy a függőségek közelebb kerüljenek egymáshoz a bemenetben és a kimenetben. Tanítási példák hossza nem haladta meg a 112 tokent bináris kódok esetén, míg ez az érték a kimeneti C kódoknál 88 volt. Fordítóként a **clang**-ot használták **-O0** kapcsolóval (optimalizációk kikapcsolása), mely x86-os assembly kódot adott vissza eredményül. Ezen példákon elért Levenshtein-távolság 0,7 volt, valamint a programok tökéletes visszafejtésének aránya nem érte el a 12%-ot.

O. Katz és tsi. [2] egy saját keretrendszert (*TraFix*) hoztak létre a probléma kezelésére. Ez két részre bontható:

- Az első egy NMT (*neural machine translation*, lényegében attention-t használó *seq2seq*) modell, mely egy kódvázat ad vissza fordított lengyel jelölésben (*RPN*) a kapott alacsony szintű bemenet alapján.
- A második rész ebbe illeszti bele a megfelelő azonosítókat és literálokat. Ezek meghatározásához gráfizomorfizmussal potenciális kimeneteket hoznak létre, melyeket újra lefordítva összevetnek a bemenettel.

A számliterálok kezelése végett azokat számjegyekre bontották a bemenet létrehozásakor. A tanítás során mindössze 10 000 adatot használtak egyszerre, azonban ha a modell elért egy bizonyos eredményt, és már nem tudott tovább tanulni, a példák egy részét lecserélték. A keretrendszer segítségével betanították a NMT modellt először **LLVM IR**, majd **AT&T x86**-os assembly C kódra alakítására. Ennek eredményeként megállapították, hogy az utóbbi nehezebb feladat kódvisszafejtés szempontjából, különösen, ha az akár csak az **-O0**-nál is jelen lévő optimalizációk végrehajtnak. Eredményeik kiértékeléséhez nem a kimeneti C kódokat, hanem azok visszafordítását vetették össze a bemenettel, melyeken a teljes egyezés arányát vizsgálták (tökéletes kódvisszafejtési arány). Az első tanítási fázist követően ebben kevesebb, mint 74%-ot tudtak elérni x86-os tesztelt példákkal. A kimenet előállításához 5 méretű nyalábkeresést (*beam search*) használtak. Hossz tekintetében a keretrendszerük által sikeresen visszaalakított leghosszabb AT&T x86-os kód 668 tokenből állt, míg ez LLVM IR esetén 845 volt.

Szintén az alapmunka eredményeinek javítását tűzték ki célul C. Fu és tsi. [3], akik a *CODA* nevű keretrendszert hozták létre. Ez két fő komponensre bontható:

- A kódvisszafejtő rész egy speciális, *N-áris fa-LSTM*-et és attentiont használó encoder-decoder modell, mely a kimenetét absztrakt szintaxisfa (*AST*) formában adja meg.
- A hibajavításokért felelős komponens iteratíván próbálta a generált hibákat detektálni majd javítani két külön modell segítségével. Ennek szükségességét a kétoperandusú műveletek kezelése, és azokon belül is a literálok és azonosítók helyes prediktálásának nehézsége eredményezte, ahogy O. Katz és tsi. [1] esetében is.

A hibajavításnak köszönhetően az átlagos tökéletes program visszaalakítási hatásfokuk 30% helyett 82% volt. Példáik átlagos hossza 309, a legbonyolultabbaké pedig 637 token volt. Példáikat clang fordítóval fordították minden optimalizációt kikapcsolva. Az ő munkájuk az első end-to-end (csak mélytanulási eszközöket használó) megközelítés.

R. Liang és tsi. [4] a fenti munkákon próbáltak javítani, létrehozva ezzel a *Neutron* keretrendszert. Tekintettel az előzőek hosszban limitált mivoltára, két lépésre bontották a kódvisszafejtést:

- Először egy neurális hálóval a bemenetet feldarabolják, amelynek alapesetben 93,85%-os pontosságát egy másik, hibajavító modellel 100%-ra sikerült javítani.
- Ezt követően a darabolt bemenetet egy másik neurális hálóval visszaalakítják C kódra, amelyben hibajavítással részletenként 78,03% helyett 99,98%-os pontosságot értek el.

Az így elkészült modell assembly szegmenseket volt képes visszaalakítani C kódrészekre, viszont a darabolás miatt a változók, a konstansok és a szövegek csak utólagos behelyettesítéssel és megfeleltetéssel lettek helyreállítva kézzel írt szabályok alapján. A megfelelő bemenet előállításához, illetve az adat- és vezérlésfolyam (*data flow*, *control flow*) helyreállításához gépi tanulást nem alkalmazó szoftvert alkalmaztak (*angr*⁵). Mivel minden assembly utasítást C utasításra próbáltak alakítani, így regiszterimitációt kellett alkalmazniuk az átmeneti értékek kezelésére. Amennyiben egy vezérlési szerkezetet nem sikerült visszaalakítaniuk, úgy alkalmaztak *goto* utasítást a megfelelő működés biztosítására. Fordításhoz gcc-t használtak minden optimalizációt kikapcsolva. Összességében ez a megközelítés 74,71%-kal múlt felül az egyszerű LSTM modell, míg 36,11%-kal az attention-t használó LSTM modell eredményét.

R. Liang és tsi. [5] a fenti munkájuk mellett transformer alapú megközelítést is alkalmaztak a feladat megoldására. Munkájuk eredménye a *SEAM* keretrendszer, mely egy saját köztes nyelvre alakítja vissza az assembly kódot, melyet utána könnyű C kóddá alakítani. A saját köztes reprezentáció előnye, hogy állításuk szerint megtartja a szemantikai információt, viszont formájában közelebb áll az assembly-hez.

⁵<https://github.com/angr/angr>

Korábbi munkájukhoz hasonlóan itt is szegmentálják az alacsony szintű kódot, megoldást nyújtva ezzel a hosszproblémára. Továbbiakban is gondot jelentett azonban a modellnek a literálok és az azonosítók kezelése, így ezt ezek kimaszkolásával, és utólagos behelyettesítésével oldották meg. Ennek részeként egy harmadik modellt is alkalmaztak, mely „beszédes” változóneveket próbált adni az egyes változóknak. Az azonosítókat a modell kimenetében kicserélték, így a modellnek ezeket nem kellett megtanulnia. Az ugrásoknál az assemblyben azok címkéjét iránytól függően az **up**, **down** tokenekkel helyettesítették. Mindezen módosításoknak köszönhetően a visszafordítás hatásfoka 36,10%-kal nőtt. A szegmentálás terén 98,64%-os hatásfokot értek el. A példákban az AST legnagyobb mélysége 5 volt. Tekintve, hogy egy kétope-
randusú művelet értékadása legalább 3 mélységet jelent, így a példák komplexitása alacsonynak mondható. Az általuk elért tokenenkénti pontosság 98,65% volt, míg a tökéletes visszaalakítás a valós kód példákon 94,41% volt.

I. Hosseini és tsi. [6] egy sokkal általánosabb megközelítéssel próbálta megoldani a feladatot. Munkájuk során próbálták a legáltalánosabb modellt megadni, amely visszafordításra alkalmas, így minimalizálva a doménspecifikus többlettudást, melyet a korábbi cikkek jelentősen kiaknáztak a hatásfok növelése érdekében. Így ők egyszerű szöveggént kezelték a kódokat és nagyságrendileg 6 karakterenként dolgozták fel őket, törekedve a szótár méretének csökkentésére. Kimérték azonban, hogy Fortran nyelven ez a darabolás a karakterenkénti feldolgozáshoz képest 0,03-mal rosszabb átlagos szerkesztési távolságot eredményez, amely nem szignifikáns különbség, de így is megkérdőjelezi a módszerük hasznosságát. Az általuk létrehozott keretrendszert 4 különböző magas szintű nyelvre való visszafordításra is betanították (Go, Fortran, OCaml, C), melyek közül a legnehezebbnek a OCaml bizonyult. Állításuk szerint a neurális kódvisszafejtők számára (sajátjukat is ideértve) a legnagyobb nehézséget a konstansok és az operátorok kezelése jelenti. Ember által írt kódokat használnak, mert szerintük azok változatosabbak. Előfeldolgozásként a tabulálást egységesítették, a kommenteket eltávolították és a szövegliterálokat kimaszolták. A számokat a kezelés érdekében számjegyekre darabolták fel. Modelljük az eredeti transformer [14] modell volt. Átlagos szerkesztési távolság, melyet elértek C nyelven 0,46, mely szerint közel minden második token érintett lenne valamilyen módosításon. Állításuk szerint a tesztelt példák nagy része két csoportba sorolható: szinte tökéletes vagy teljesen rossz.

Korábbi munkánkban [7] megmutattuk, hogy lehetséges egy end-to-end, single-pass neurális hálóval sikeresen végrehajtani kódvisszafejtést utólagos hibajavító technikák nélkül. Munkánkban elsőként alakítottunk vissza switch utasításokat és ternáris kifejezéseket. Az alapvető utasításkészletet tekintve a modellünk 98,0%-os tokenenkénti pontosságot ért el a gcc fordító alapértelmezett (**-O0**) optimalizációi mellett. Kijelenthető, hogy a munkánk felülmúlta az összes olyan cikk eredményét, amelyek nem alkalmaztak feldarabolást a bemeneti assembly kódon, és a többinek eredményeit is megközelítette vagy felülmúlta. Megmutattuk, hogy a modellünk számára az egymásba ágyazott kétoperandusú műveletek és a switch utasítások jelentik a legnagyobb nehézséget, míg az elágazások és ciklusok esetén több mint 99%-os tokenenkénti és 86% feletti tökéletes visszafordítási arányt értünk el. Korábbi munkánkat a Módszerek (2) fejezetben részletezzük az azóta történt módosításokkal egyetemben.

2. fejezet

Módszerek

Az alábbiakban részletesen bemutatjuk az alkalmazott technikákat. Először ismertetjük a használt adathalmazokat, részletesen bemutatjuk az adatok előfeldolgozását, mellyel előállnak a kódvisszafejtő- és a szegmentáló modell által használt adathalmazok. Ezt követően felvázoljuk a rekurzív szegmentálás módszerét, végül bemutatjuk ezen modellek felépítését, és azt, hogy ezek felhasználásával hogyan áll elő adott bemenetből egy kimeneti C kód.

2.1. Adatgenerálás, adathalmazok

A mélytanulás alapvető feltétele a megfelelő minőségű és mennyiségű adat, melyből a használt architektúra általánosításokat tud levonni, ezáltal tanul.

Munkánk során a tanítási példákat generáljuk, ez számos előnnyel kecsegtet:

- Generált adattal tanítva a modell nem csak a konvencionális kódokat fogja tudni visszaalakítani, hanem a nem szokványos műveleteket is.
- A generált kódban sokkal egyenletesebb az eloszlása az egyes nyelvi elemeknek, mint a kézzel írtakban. Meglehetősen ritka olyan C kódokat látni, amelyben több switch vagy több ternáris operátor is szerepel egy kisebb szakaszon, míg a generált kódokban ilyen példák is előfordulhatnak.
- Adatgenerálással tetszőleges mennyiségű kontrollált minőségű tanítópélda állítható elő, míg a kézzel írt kódok esetén ez korlátozott. Adataink előállításánál esetén magunk tudjuk testre szabni, hogy az egyes nyelvi elemek milyen gya-

korisággal forduljanak elő, illetve a létrejövő példák hossztartománya milyen intervallumba essen.

Adatgenerálásra a korábbi munkánkban [7] létrehozott algoritmus új nyelvi elemekkel kibővített változatát használtuk.

Emellett a korábban összegyűjtött valós kódpéldákat is felhasználtuk a modelünk kiértékelésére.

2.2. Alacsony szintű kód darabolás és párosítás

Fordítás során az utasítások száma nem csökken, sőt, általánosságban jelentősen megnő, miközben magas szintű kódból alacsony szintű kódokká alakulnak át. Az így összefüggőnek tekinthető utasításokat kiemelve a terjedős alacsony absztrakciós szintű kód feldarabolható oly módon, hogy egy utasítás sorozat megfeleljen egy magas szintű utasításnak. Annak érdekében, hogy megfelelő tanító példákat hozzunk létre mind a szegmentáló, mind a visszafejtő modell számára, pontosan és konzisztensen meg kellett határoznunk a sorozatok kezdési helyeit és jellegét, majd ezek alapján összepárosítani a sorozatokat a magas szintű utasításokkal.

A generált C példákat lefordítottuk a gcc¹ **-g** kapcsolójával, így a tárgykódba bekerülnek a hibajavítási információk (lényegében az assembly sorok egy részéhez azok „forrása”). A létrejövő tárgykódot a **binutils**² csomag **objdump** programjának segítségével visszaalakítjuk assembly kódra, melyben megtalálható, hogy az egyes utasítások mely C sorhoz tartoztak. Ezek alapján már el lehet kezdeni összepárosítani a két kód megfelelő részeit, és egy címkézést létrehozni az assembly kód minden sorához.

Bár naiv megközelítéssel elegendő lenne csak a sorozatok első elemét megkülönböztetni egyetlen címkével, ez nehézkessé tenné a vezérlési szerkezetek kezelését, melyek általánosságban közrefognak néhány önmagukban is elkülönülő utasítássorozatot. Ennek eredményeként a címkézést 12 különböző címke használatával végeztük:

- függvények
 - kezdete
 - egyéb részei

¹<https://github.com/gcc-mirror/gcc/tree/releases/gcc-9.4.0>

²<https://www.gnu.org/software/binutils/>

- nem vezérlési szerkezetekhez tartozó utasítások
 - kezdete
 - egyéb részei
- elágazások kezdete
- elől tesztelő ciklusok kezdete
- hátul tesztelő ciklusok kezdete
- switch utasítások kezdete
- nem szétválasztható vezérlési szerkezetek kezdete
- vezérlési szerkezetek egyéb részei
- vezérlési szerkezetek által tartalmazott utasításblokkok
 - kezdete
 - egyéb részei

Egyes esetekben két egymást közvetlenül követő vagy tartalmazó vezérlési szerkezet összemósódhat (ugyanazon címkét használják assemblyben), így nem, vagy csak nehezen eldönthető, hogy az egyes utasítások határai hol húzódnak. Ennek eredményeként ezeket az eseteket külön címkével (*Inseparable*) jelöltük (*nem szétválasztható vezérlési szerkezetek kezdete*).

A vezérlési szerkezetek által tartalmazott blokkok esetén az egész blokkot egyetlen speciális sorozatnak tekintjük, melyet a visszafejtő modell esetén BLOCK tokennel helyettesítünk mind az assembly, mind a C kódban, tartalmát kiemelve azt rekurzívan újra elkezdjük szegmentálni. Ennek eredményeképpen egyre kisebb részekre tudjuk a bonyolult vezérlési szerkezeteket is bontani oly módon, hogy elkülönítjük őket a belső blokkoktól.

2.3. Előfeldolgozás

Korábbi munkánk [7] során megmutattuk, hogy utólagos hibajavítás nélkül is lehet hatékonyan assembly kódot C programozási nyelven írt kódra alakítani. Ennek

egy feltétele volt, hogy a bemeneti és az elvárt kimeneti kódokban ugyanazok a literálok és azonosítók (függvénynevek) szerepeljenek. Tekintve, hogy a *gcc* fordítás során jelentősen átalakítja a kód struktúráját, így több mint 100 különböző optimalizációs szabályra volt szükség, illetve a *volatile* kulcsszó használatára, amivel garantálni tudja az egyértelmű megfeleltetést a változók és a memóriacímek (*offset*) között.

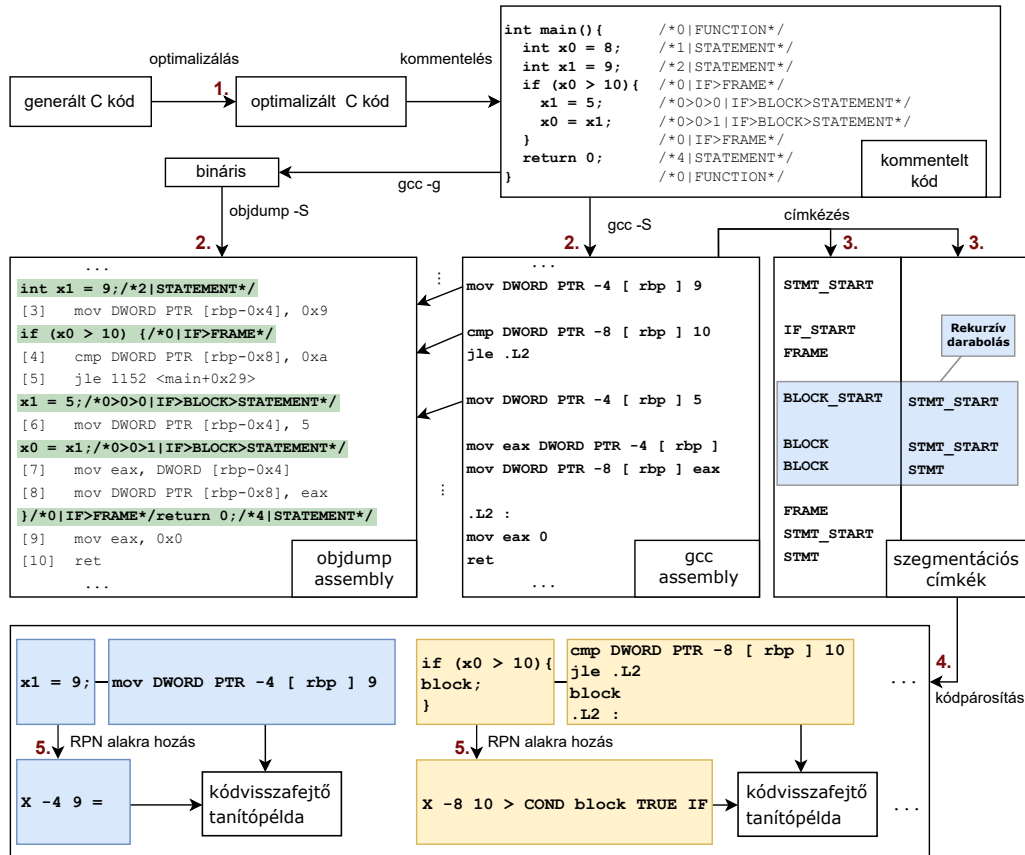
Jelenlegi munkánk során ezt a *volatile* kulcsszó használata nélkül is sikerült elérni. A változók és memóriacímek közötti megfeleltetésre alkalmazott módszerünket a 2.3.3 pontban részletezzük. Azonban ennek, illetve az új nyelvi elemek bevezetésének köszönhetően jelentős mennyiségű új szabályt kell megvalósítanunk. Ezek biztosítják, hogy a generált tanítási adatokban ne szerepeljenek azonos assembly-hez különböző C kódok.

A szabályok számát tovább növelte, hogy a másolást lehetővé tevő feltételeknek kisebb kódrészleteken kell teljesülnie a szegmentálásnak köszönhetően. Ezek értelmében meg kell egyezniük (vagy egyszerűen részhalmaz viszonyban lenniük) a literálok és azonosítók számának és értékeinek a tanítási be- és kimenetekben. Ezen szigorítás is több szabály megvalósítását eredményezte.

Az új nyelvi elemek bevezetésének eredményeképp összességében több mint 250 új szabályt implementáltunk le, melyekkel együtt a szabályaink száma meghaladja a 400-at.

A továbbiakban kitérünk az előfeldolgozás főbb lépéseire és azok feladatára, működésére. A szegmentációs- és visszafordítási adathalmaz generálása egyidejűleg történik, mivel az előállítási folyamataik egymásra hagyatkoznak. Ezen adathalmazok egy-egy példája a kódgeneráló komponens által generált egy-egy C kódpélda feldolgozásával áll elő. Az előfeldolgozás főbb lépéseire áttekintést nyújt az alábbi felsorolás, illetve a 2.1 ábra.

1. A C kódra optimalizációs transzformációk végrehajtása.
2. A kód kommentelésével többletinformáció hozzáadása; assembly kódok előállítás kétféle módon (a *gcc* és az *objdump* használatával).
3. A szegmentáló modell adathalmazának előállítása.
4. A szegmentáció által meghatározott assembly kódrészletek összepárosítása a nekik megfelelő C kódrészletekkel.



2.1. ábra. Az előfeldolgozás főbb lépései, egy példán keresztül. Első lépésként a generált kódra (bal felső sarok) alkalmazzuk a megfelelő optimalizációs szabályokat (a visszafejtés egyértelműségének segítése érdekében). A kapott optimalizált C kódhoz hozzáfűzünk a struktúrájáról információt tartalmazó kommenteket (kommentelt kód). Ezt követően két módon is lefordítjuk a kódot. Egyrészt a `gcc -S` használatával előállítjuk a visszafejtendő assembly kódot (gcc assembly). Másrészt a `gcc -g` segítségével először egy átmeneti bináris fájlt állítunk elő, majd ebből az `objdump -S` használatával előállítjuk az `objdump assembly` kódot. A gcc assembly sorait behelyettesítjük az `objdump assembly` sorainak helyére (esetenként lényegi különbség is előfordul a kettő között). Az `objdump assembly` hivatkozásokat tartalmaz a kommentelt C kódra; ezen kommentek segítségével visszafejthető, hogy melyik assembly sor milyen struktúrán belül helyezkedik el. A strukturális információ segítségével előállíthatók a szegmentációs címkék (a szegmentáló modell elvárt kimenetei). A szegmentációs címkék (illetve a kommentek) alapján összepárosíthatók az egyes assembly darabok a nekik megfelelő C kóddal. A vezérlési szerkezetek blokkjait egy helyettesítő tokenre cseréljük, mert a belső blokkokat az őket tartalmazó struktúráktól függetlenül, rekurzívan dolgozzuk fel. Végül a C kódrészeket RPN (Reverse Polish Notation) alakra hozzuk (ez a kódviisszafejtő modell elvárt kimenete).

5. A C kódrészletek RPN alakra hozása (ezzel előáll a kódvisszafejtő modell adathalmaza).
6. Utólagos műveletek az adathalmazon (speciális esetek kezelése, illetve a példák egy részének kiszűrése az adathalmazból).

2.3.1. Az eredeti C kód optimalizációs transzformációja

A fordítási folyamat során a gcc fordító rengeteg optimalizációs átalakítást végez el a kódon. Ezen átalakítások javítják a program hatékonyságát, azonban nagyban megnehezítik a program forráskódjának visszafejtését. Egyrészt az egymással ekvivalens kódok jelentős része (az optimalizációk következtében) ugyanarra a kimeneti assembly kódra fordul (például a $(x + x)$ és $(x * 2)$ kódok ugyanarra az assembly kódra fordulnak). Ez megnehezíti az egyértelműség kérdését, hiszen adott assembly utasításblokkhoz így több megegyező C utasítás is tartozhat. Másrészt a literálok kezelését is megnehezítik az optimalizációk, ugyanis az átalakítások során a számliterálok is átalakulhatnak. Emiatt a bemeneti assembly- és kimeneti C kód párban eltérhetnek a számliterálok, ami ellehetetlenítené a kódvisszafejtő modell másolási mechanizmusának a használatát. Hasonló módon a szegmentációs adathalmaz előállításánál is nehézséget okozhatnak az optimalizációk. Ha egy vezérlési szerkezet feltétele egy kielégíthetetlen formula vagy tautológia, akkor a keletkező assembly kód struktúrája nagyban eltérhet a C kódtól (például tautológia feltételű elágazás esetén az elágazás teljesen eltűnik, és csak a belső kódblokkja marad meg).

Ezen problémák motiválják azt, hogy az előfeldolgozás első lépéseként a generált C kódon optimalizációs transzformációkat hajtsunk végre, melyek a gcc optimalizációs átalakításait kívánják utánozni. Ezáltal egy olyan kódot kapunk, melyben a literálok megegyeznek a belőle forduló assembly kód literáljaival (pontosabban rész multihalmaz viszonyban vannak egymással), és az assembly kód struktúrája követi a C kód felépítését. Az egyértelműség érdekében ezt, a továbbiakban optimalizált C kódnak nevezett kódot definiáljuk a kódvisszafejtő modell elvárt kimenetének.

Korábbi munkánk [7] során már alkalmaztunk optimalizációkat, melyeket az alábbi munkánk során kibővítettünk. Az optimalizációk elvégzéséhez a C kódot a könnyebb módosítások érdekében a pycparser [15] segítségével absztrakt szintaxisfává (AST - Abstract Syntax Tree) alakítjuk. Az így létrejövő AST-t rekurzívan bejárjuk és részfát megpróbáljuk optimalizálni az aktuális állapot szerint. Amennyiben

egy részfa optimalizáláson esett át, úgy az őt tartalmazó részfat is megpróbáljuk újra optimalizálni, tekintve, hogy nem teljes általánosságban lettek az optimalizációs szabályok megírva, hivatkozhatnak a többi optimalizáció helyes működésére. Elkerülve az esetleges körkörös optimalizációkat a részfa állapotait vizsgáljuk és amennyiben visszatér egy korábbi állapotba, úgy detektáljuk a körkörösséget és nem folytatjuk a példa feldolgozását. Hasonlóan limitáltuk, hogy egy csúcsba hányszor lehet visszatérni (az esetleges számliterálokkal való nem ciklikus végtelen módosítások korlátozása végett). Az utóbbi limitáció főleg a fejlesztési fázisban töltött be szerepet, a tényleges adatgenerálás során nem jelent korlátot.

2.3.2. Az assembly kód előállítása

A C kódot kétféle módon is assembly kóddá alakítjuk: egyrészt a **gcc** fordítóval előállítunk egy átmeneti bináris fájlt, melyet az **objdump** program segítségével egy olyan assembly kóddá alakítunk, melyből kinyerhető a C kód strukturális információja (ez a szegmentáció meghatározásában elengedhetetlen szerepet játszik). Másrészt a **gcc** fordítóval előállítjuk azt az assembly kódot, mely a kódvisszafejtő modell bemenetétül szolgál

Az objdump által adott assembly

Először a gcc **-g** kapcsolójának használatával egy olyan átmeneti bináris fájlt hozunk létre, melyben szerepelnek debug információk. Ezt a bináris fájlt az **objdump** program **-S** kapcsolójával visszaalakítjuk assembly kóddá. A debug információknak köszönhetően az objdump meg tudja határozni, hogy az assembly kód egyes részeihez mely C kódrészletek tartoznak (az assembly kódrészletek elé bemásolja azt a C kódot, amiből az assembly kódrészlet keletkezett). Ez alapján ugyan össze lehet párosítani az assembly és C kódrészleteket egymással, de a vezérlési szerkezetek egymásba ágyazottságára csak nehézkesen lehet belőle következtetni (a rekurzív szegmentálás meghatározásához ez kifejezetten fontos).

Annak érdekében, hogy a szegmentációs adathalmaz előállítása során az egymásba ágyazottságra is tudjunk következtetni, a C kódot többletinformációval látjuk el. Minden C kódbeli sorról meghatározzuk, hogy mely vezérlési szerkezetek belsejében helyezkedik el (többszörösen egymásba ágyazott vezérlési szerkezetek esetén az is fontos, hogy milyen sorrendben tartalmazzák egymást). Ahhoz, hogy ez a struktu-

rális információ az assembly kódban is elérhető legyen, a fordítás előtt az egyes C sorokhoz komment formájában hozzáfűzzük a róluk számon tartott információt. Az **objdump** kimenetében ezek a kommentek a megfelelő helyeken megjelennek, így az egyes assembly kódblokkokhoz hozzá tudjuk rendelni a strukturális információt.

A gcc által adott assembly

A gcc **-S** kapcsolóját használva előállítjuk a kódvisszafejtő modell bemenetéül szolgáló assembly kódot. A kapott kódból eltávolítjuk a kódvisszafejtés szempontjából lényegtelen utasításokat (kanonizálás). Ezt követően az objdump által adott kimenet assembly sorait lecseréljük a gcc által adott assembly soraira (a szintaktikai eltérések mellett a kettő között előfordulhatnak lényegi különbségek is). Ezzel biztosítjuk, hogy az adatfeldolgozás során mindig megfelelő formátumú assembly kóddal dolgozunk. A továbbiakban ezt a keletkezett kódot (melyben a C kód strukturális információi és az assembly sorok szerepelnek) összevegyített assembly kódnak nevezzük.

2.3.3. Változó-memóriacím megfeleltetés

A kimenetül kapott assembly kódban az eredeti C kódbeli változók nevei elvesznek, a változók helyett csak memóriacímek szerepelnek az assembly kódban. Mivel a kódvisszafejtő modell csak ezeket a memóriacímeket látja, ezért nem várhatjuk azt tőle, hogy a változók neveit visszafejtse. Helyette célszerűbb a memóriacímek segítségével hivatkozni a változókra a kimenetben is. Ilyen kimenet előállításához azonban szükséges ezen memóriacímek és C kódbeli változók összepárosítása.

Ennek a párosításnak az elvégzéséhez a C kód generálásánál speciális értékadási műveleteket szúrunk be a kód elejére úgy, hogy minden, a kódban használt változónak értékül adunk egy olyan egyedi számot, ami biztos nem fordul elő a generált kódban máshol. Mivel az értékadások az assembly kódba is bekerülnek, az értékül adott egyedi szám alapján meg tudjuk határozni, hogy melyik változó melyik memóriacímhez tartozik.

Miután végeztünk az összepárosítással, mind az assembly-, mind a C kódból eltávolítjuk ezeket az értékadásokat, hogy ne torzítsák az adathalmazt. Ez biztonságosan megtehető, mivel ezek az értékadások nincsenek kihatással a kód többi részére, azaz eltávolításuk nem okoz inkonzisztenciát.

2.3.4. A szegmentációs példák előállítása

A szegmentációs példák előállítása az összevegyített assembly kódban jelen lévő C kódrészletek (illetve az azokban elkódolt strukturális információk) segítségével történik. A szegmentációt az assembly kódban szereplő függvényekre külön-külön határozzuk meg. A szegmentációs adathalmazt rekurzívan állítjuk elő, azaz miután egy adott assembly kódszakaszra meghatároztuk a szegmentációt, az esetleges vezérlési szerkezetek belső blokkjait rekurzívan tovább daraboljuk.

A szegmentációtól elvárjuk, hogy:

- Elkülönítse az assembly kódban szereplő függvények inicializáló utasításait.
- A C kódban lévő egyszerű utasításoknak és vezérlési szerkezeteknek feleljen meg egy-egy kóddarab a szegmentációban.
- A vezérlési szerkezeteknél legyenek elkülönítve egymástól a belső blokkok, illetve a feltételkiértékelő/vezérlési folyamat kezelő kódrészek.
- A vezérlési szerkezetek belső blokkjaira is legyen meghatározva a szegmentálás. Ez azt jelenti, hogy egy C kódból több szegmentációs tanítópélda is keletkezik (mert a blokkok szegmentációi újabb tanítópéldákként kezelhetők).

A szegmentációt az assembly kódsorokhoz rendelt címkékkel adjuk meg. A használt címkék a 2.2 részben fel vannak sorolva. A szegmentáló modellnek a feladata az egyes assembly sorokra ezeket a címkéket meghatározni.

2.3.5. A kódvisszafejtési példák előállítása

A kódvisszafejtési példák előállítása a szegmentációs példák segítségével történik. Mivel a kódvisszafejtő modell bemenetei már szegmentált assembly kódrészletek, ezért a tanítási adathalmaz előállításához az assembly darabokat összepárosítjuk a nekik megfelelő C kódrészletekkel. A C kódrészleteket fordított lengyel jelöléssel (RPN) adjuk meg, és a kódvisszafejtő modelltől is ilyen alakban várjuk a kimenetet.

Az assembly és C kóddarabok párosítása

Adott C forráskód esetén először a szegmentációs adatok alapján feldaraboljuk az assembly kódot (a belső blokkokat is, rekurzívan). Az összevegyített assembly kódban szereplő információk alapján az assembly sorokhoz hozzárendelhetők a C

kód sorai (az összevegyített assembly kódban szereplő C kódrészletek közvetlenül nem használhatók fel, mert túl rendezetlenek). Vezérlési szerkezeteknél és függvényeknél a belső blokkot helyettesítjük egy blokk jelzővel mind az assembly- mind a C kódban, ezáltal a tanítópéldák rövidülnek (ezen blokkok tartalma külön példaként kerül feldolgozásra).

RPN alakra hozás

Szerkezetét tekintve az assembly kód távol áll a C kódtól, így egy vele teljesen megegyező RPN alakra hozzuk, mely jobban igazodik az assembly kód struktúrájához. Ezt a reprezentációt elsőként O.Katz és tsi. [2] alkalmazta, illetve mi is a megelőző munkánk [7] során.

Az átírás alatt a szám literállal való szorzás, osztás és maradékszámítás átalakul egy gyorsabb számítási módra bitműveletek, kivonások, összeadások és szorzások sorozatává. Ilyen átalakulás lehet osztás és maradékszámítás esetén a multiplikatív inverzzel való szorzás (véges számábrázolás miatt létezhet ilyen), illetve szorzás esetén bitshiftelés alkalmazása.

Másik transzformációt is alkalmazunk az RPN alakra hozás közben: a ciklusok feltételében lévő logikai állításokat negáljuk, így az elágazásokkal megegyező feltétel esetén történik meg az ugrás.

A fenti két transzformáció egyértelműen visszaalakítható, így adat és funkcióvesztés nem történik.

2.3.6. Utólagos műveletek az adathalmazon

Példák kiszűrése az adathalmazból

A másoló mechanizmus megfelelő működése érdekében elvárjuk, hogy a C kódban megjelenő literálok rész multihalmaza legyen assemblyben lévő literáloknak: minden literálnak a C kódból elő kell fordulnia az assembly kódban és legfeljebb annyiszor fordulhat elő a C kódban mint az assembly kódban. Annak érdekében, hogy ezek a feltételek teljesüljenek a példák megfeleljenek a fenti kritériumnak több optimalizációt is leimplementáltunk a 2.3.1 részben leírtaknak megfelelően.

A literálokon kívül hasonló módon ellenőrizzük még a relációs operátorokat is.

Mivel a szegmentálásnak köszönhetően a kódok mérete csökkent, így a kódokban található literálok és relációs operátorok száma is csökkent az egyes részpéldák ese-

tén. Ennek köszönhetően szigorodtak a fenti feltételek a korábbi munkánkhoz képest: amennyiben egy teljes C fájlban lévő literálok korábban rész multihalmazai voltak a keletkező assembly kódnak, úgy most a darabolás miatt a rész multihalmazok nem biztos, hogy megfelelnek ugyanezen követelménynek a megfelelő assembly literálok rész multihalmazaira nézve. Ez a szigorítás további optimalizációs szabályok implementálását vonta maga után.

Összességében a generált példák kevesebb, mint 5%-át dobtuk ki.

Switch ugró táblák kezelése

Az assembly kódban a switch vezérlési szerkezetek egyes esetekben ugró táblákat tartalmaznak. Ezek azért jönnek létre, hogy csökkentsék az összehasonlítások számát a kódban. Az ugró táblák azonban jellemzően hosszúak és a kódvisszafejtő modell számára nehezen értelmezhetőek, ezért az assembly kódban átalakítjuk őket egy rövidebb formára, amit a modell könnyebben vissza tud fejtetni.

Annak érdekében, hogy a modellünk kezelni tudja az ugró tábla bejegyzéseihez tartozó literálokat, minden bejegyzés után beszúrjuk azok sorszámát kommentként a sor végére. Ezzel az assembly jelentését nem módosítjuk, más többletinformációt nem adunk hozzá az előfeldolgozás során.

Szegmentációs példák utólagos módosítása

Mellékhatásos kifejezések esetén előfordul, hogy az assembly kód darabolása (az eredeti C kód ismerete nélkül) nem egyértelmű. Mivel a modell tanításánál ez problémát okozhat, ezeket a többértelműségeket az adatgenerálás során megszüntetjük.

Vegyük példának az alábbi ekvivalens kódokat:

1. `y = ++x;`
2. `x += 1;`
`y = x;`

Ezek ugyanazt az assembly kódot eredményezik, ezért értelemszerűen a szegmentáció se egyértelmű: az assembly kódot lehet úgy is darabolni, hogy az első kódnak feleljen meg, és úgy is, hogy a másodiknak. Ahhoz, hogy feloldjuk a többértelműséget, mi mindig azt a darabolást választjuk, ami több részre tagolja a kódot (azaz jelen példában a második kódnak megfelelő szegmentálást választanánk). Ezért az

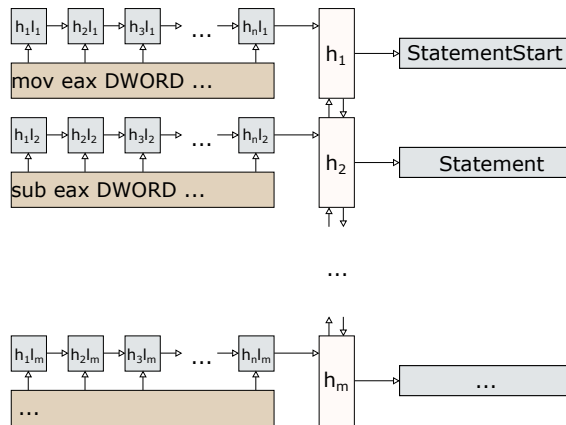
ilyen esetekben módosítjuk az elvárt szegmentációt úgy, hogy minél több kóddarab keletkezzen. Ez a módosítás azon mellékhatásos kifejezéseket érinti, melyeknek a mellékhatása a kifejezés legelején hajtódik végre (így ezek a mellékhatások kezelhetők egy külön C kódbeli utasításként).

Emiatt a szegmentálási- és kódvisszafejtési adathalmaz között kisebb inkonzisztencia lép fel (hiszen lehet, hogy a szegmentáció nem tükrözi az eredeti C kód felépítését), azonban ez nem probléma, mivel a továbbiakban ezeket az adathalmazokat már külön kezeljük. Ezen többértelműség kezelését optimalizációs szabályokkal is meg lehetne oldani, azonban ez olyan szabályok implementálását vonná maga után, melyeket a gcc fordító nem végez el. Emellett ilyen módon a kódvisszafejtő modell többféle kódból is tanulhat, hiszen a kódvisszafejtési adathalmazban továbbra is előfordulnak a különböző mellékhatásos kifejezések (itt már nincs gond az egyértelműséggel, hiszen ezen példánál már megtörtént a szegmentálás).

2.4. Modellek

Munkánk során két mélytanulási architektúrát alkalmaztunk: egyet a szegmentálás elkészítésére, egyet pedig a kódvisszafejtés megoldására.

2.4.1. Szegmentáló modell



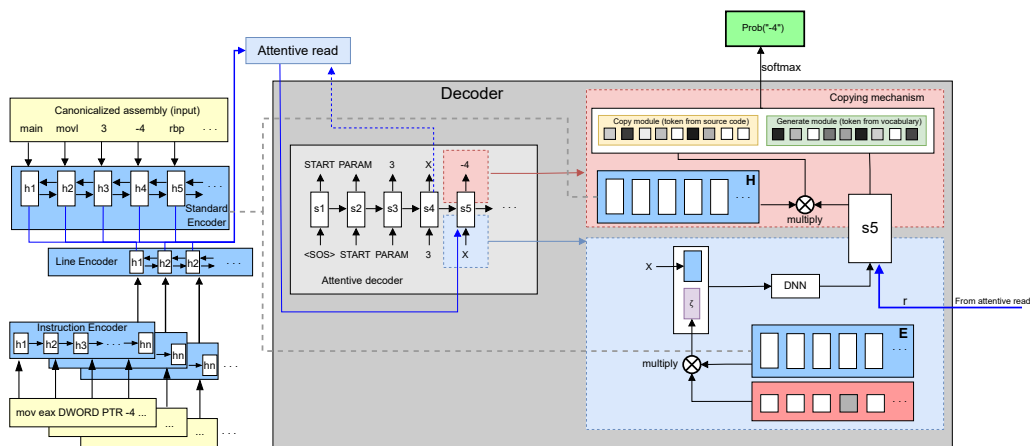
2.2. ábra. A szegmentáló modell felépítése. A kapott bemenetet (az ábrán barnával jelölve) beágyazás után szavanként feldolgozzuk egy GRU segítségével lineárisan, ezeket az ábrán h_{ilj} állapotokkal vannak jelölve. A szavankénti feldolgozást követően a sor egy rejtett vektorba lesz enkódolva, mely bemenetét képezi a kétirányú GRU-nak (az ábrán függőlegesen van elhelyezve). Ennek kimeneteire alkalmazott lineáris rétegek fogják a kimeneti címkéket megadni.

A szegmentáló modellünk egy beágyazó (**embedding**) réteget használ a bemeneten, amely egy súlyokat tartalmazó vektort rendel minden egyes bemeneti tokenhez. A túltanulás elkerülése érdekében az embedding réteg kimenetére dropout-ot alkalmazunk 0,1-es eldobási paraméterrel. A bemenetet ezután soronként egy egyirányú **GRU** [12] segítségével egy-egy rejtett állapotvektorba tömörítjük. Ezek sorozatát dolgozza fel egy második, immár kétirányú GRU réteg, mely kimenetét két egymás után fűzött (*ReLU* aktivációs függvénnyel elválasztott) lineáris réteg képezi le a megfelelő mennyiségű ($12 + 1$ padding) kimeneti címkére.

Tekintve, hogy a számliterálok nem adnak többlet információt a szegmentálás esetén, így azokat a szegmentálás idejére **NUM** tokenel maszkoltuk ki, így csökkentve a létrejövő szótár méretét. A címkék rugalmas kezelése érdekében a **.L<NUM>** alakot feldaraboltuk a **.L** részre és számjegyekre. Így a korábban ismeretlen címkék használata nem okoz gondot a modellnek: a címke jelzőjéből a címke tényét, míg a számliterálok alapján annak azonosítóját képes észlelni, és más előfordulási helyekkel összekötni.

A modell tanítására az **Adam** [16] algoritmust, hibájának mérésére a **Cross Entropy Loss** függvényt használjuk.

2.4.2. Kódvisszafejtő modell



2.3. ábra. A kódvisszafejtő modell felépítése. Az ábra a korábbi munkánkban [7] található ábra alapján készült.

Kódvisszafejtő modellünk korábbi munkánkhoz hasonlóan [7] most is egy RNN alapú encoder-decoder modell [10] attention-nel [13] és másoló mechanizmussal [8] kiegészítve. Tekintettel arra, hogy a C absztrakciós szintje jelentősen magasabb az

assembly nyelvéhez képest (az előbbi egy utasítása az utóbbiban számosnak felel meg), ezért egy sorozatnyi assembly kódot alakítunk át egy rövid C utasítássá (RPN alakban). Ennek kezelésére kiegészítéseket tettünk az **encoder** működésében megváltoztatva ezzel az egész modellünk viselkedését.

Korábban a modell bemenete egyetlen tokensorozat volt, melyben az assembly tokenek a sorok elválasztása nélkül szerepeltek, így a modellnek azt is külön meg kellett tudnia tanulnia, hogy az egyes utasítások hol kezdődnek és meddig tartanak. Ennek javításaként a bemenet feldolgozását a szegmentáló modell (2.4.1) bemenetét is feldolgozó architektúrával egészítettük ki. Az assembly így soronként is feldolgozásra kerül megadva a modell számára a lehetőséget, hogy az utasításokat egységként is kezelni tudja.

Tömbösített feldolgozás esetén a bemenet egy beágyazó (**embedding**) rétegen halad keresztül, mely által megadott vektorokat egy kétirányú GRU dolgozza fel. Soronkénti feldolgozásnál ugyanazon embedding réteg által adott súlyokat először soronként egy egyirányú GRU dolgozza fel, majd ennek kimeneteit dolgozza fel egy kétirányú GRU réteg. Az így kapott két kimenetet (összesen 4 vektor reprezentáció) összefűzzük, majd egy lineáris réteg segítségével negyedére tömörítjük. Ezt a vektor reprezentációt kapja meg a decoder, hogy a kimenetet létrehozhassa.

A **decoder** a kimeneti tokenek létrehozásához figyelembe veszi az előző lépés kimenetét képző rejtett állapotot, a legutolsó kimeneti tokenet, az előző lépés másolási súlyait és az encoder rejtett állapotait, majd ezek alapján létrehozza a következő kimeneti tokenet meghatározó eloszlást.

A másoló mechanizmus [8] által bevezetett attention mechanizmust alkalmazzuk: az előző másolási valószínűségek és kimeneti token alapján kiszámoljuk a *selective read* értékeit, majd ezekkel súlyozzuk az encoder által létrehozott, memóriában eltárolt vektorokat.

Ezen felül használjuk az attention mechanizmust [13] is, melyhez összeszorozza az aktuális rejtett állapotot az encoder kimeneteivel, majd ezen szorzat normalizált alakjával súlyozza az encoder kimeneteit.

A két attention eredményét és a bemenetre alkalmazott embedding réteg kimenetét összefűzi, majd bementként odaadja a saját egyirányú GRU rétegének, ezzel léptetve az aktuális rejtett állapotot is. A kapott új rejtett állapotot *ReLU* aktivációs függvényekkel elválasztott lineáris rétegek sorozata a nem-másolható tokenek súlyaivá alakítja. Emellett a másoló mechanizmus működése értelmében az új rejtett

állapotot összeszorozzuk az encoder rejtett állapotaival, ezzel egy súlyozást hozva létre a bemenet egyes tokenjeihez. Az így kapott két súlyozásból egyetlen valószínűségi eloszlást hozunk létre a **softmax** függvény segítségével, majd a bemenetben többször szereplő tokenek súlyait összevonjuk, ezáltal megkapva a másolási valószínűségeket.

Tanítás során hibafüggvényként az **NLLLoss**-t alkalmazzuk a kimeneti valószínűségek (elemenkénti) logaritmusának ellentettjére, míg tanító algoritmusként az **Adam**-et [16] használtuk. A mélyebb tanítás érdekében a decoderben az embedding réteg kimenetére alkalmaztunk egy **dropout** réteget is.

2.5. C kód előállítás

A részek visszafordítása után újra össze kell illeszteni a létrejövő RPN darabokat. Mivel a kódrészek rendezettsége a darabolástól kezdődően végig megmarad, helyüket pedig a **BLOCK** kulcsszó egyértelműen jelöli a visszafordítást követően is, így elegendő végigmenni a **BLOCK** kulcsszavakon és kicserélni őket a megfelelő RPN kódrészletre. Ezt az algoritmust a feldarabolási rekurzió legalsó szintjétől kezdjük és haladunk egyre fel a rekurzió mentén. Az eljárás eredményeként megkapjuk a teljes RPN kódot.

A C kódot korábbi munkánkhoz[7] hasonlóan alakítjuk vissza RPN alakból. Ennek első lépése a szorzás, osztás, maradékszámítás szabály alapú felismerése és visszaalakítása. Ezt követően az RPN-t AST formává alakítjuk egy verem segítségével, operátor vagy vezérlési szerkezet észlelése esetén kivesszük a megfelelő mennyiségű elemet a veremből és aktuális nyelvi elem alá rendelve új csúcsot hozunk létre az AST-ben. Az AST formából a **pyparser**[15] segítségével egyértelműen visszaalakítjuk C programkóddá.

3. fejezet

Kiértékelés

Az alábbiakban ismertetjük kiértékeléshez használt mérőszámokat és a módszereink eredményeit.

3.1. Mérőszámok

Korábbi munkánkhoz hasonlóan [7] most is két mérőszámot alkalmazunk.

Tokenenkénti pontosság Az első mérőszám a tokenenkénti pontosság, mely tokenenként vizsgálja az elvárt és a kapott kimenetet. Amennyiben az adott pozíción megegyeznek a tokenek, úgy 1, ellenkező esetben 0 értékkel veszi az adott tokenet. Az így kapott értékeket összeadja és elosztja az elvárt kimenet hosszával, így minden bemenetre megkapjuk a helyesen visszaadott tokenek arányát. A későbbiekben az egyes bemenetekre számolt érték átlagát fogjuk érteni a tokenenkénti pontosságon.

Tökéletes kódvisszafejtés/címkézés aránya A második mérőszám a tökéletes visszafordítási/címkézési arány, mely a kapott és az elvárt kimenet teljes egyezését vizsgálja. Ezen érték 1, ha az adott bemenetre a teljes kód megegyezik, tehát a kapott kimenet minden tokenje megegyezik a megfelelő tokennel az elvárt kimenetben, ellenkező esetben az érték 0. A továbbiakban ezen értéknek a vizsgált bemenetekre vett átlagát fogjuk tökéletes visszafordítási aránynak nevezni, hasonlóan a tokenenkénti pontossághoz.

3.2. Tanító adathalmaz és tanítási paraméterek

Tanító adathalmazként a fent (2. fejezet) leírt módon generáltunk 200 000 példakódot, melyeket ezután előfeldolgoztunk. Az így létrejövő 1 200 000 szegmenst tartalmazó kódvisszafejtési és szegmentálási adathalmazzal tanítottuk a modellünket.

A szegmentáló modellünk tanítási adatai a 3.1 táblázatban olvashatóak.

Paraméter	Érték
tanítási ráta	10^{-4}
batch méret	1024
rejtett állapot mérete	128
soronkénti rejtett állapot mérete	16
beágyazási méret	16
epochok száma	100
dropout kidobási arány	0,1

3.1. táblázat. Szegmentáló modellünk tanítási paraméterei

Kódvisszafejtő modellünk tanítási paraméterei a 3.2 táblázatban olvashatók.

Paraméter	Érték
tanítási ráta	3×10^{-5}
batch méret	512
rejtett állapot mérete	512
soronkénti rejtett állapot mérete	32
beágyazási méret	128
epochok száma	100
dropout kidobási arány	0,4

3.2. táblázat. Szegmentáló modellünk tanítási paraméterei

A szegmentálásnak köszönhetően az egyszerre feldolgozott kódszegmensek mérete lecsökkent. Az adathalmazokban lévő példák címkénkénti maximális token hossza a 3.3 táblázatban olvashatóak.

3.3. Eredmények

Az alábbiakban ismertetnénk a munkánk során elért eredményeket. Mivel két külön modellt alkalmazunk a szegmentálásra és a kódvisszafejtésre, így ezek eredményeit is külön mérjük ki.

Nyelvi elem	Maximális assembly hossz
Statement	170
If	103
While	122
Do-while	100
Switch	251
Inseparable	258
Other	34

3.3. táblázat. A leghosszabb assembly példák hossza egyes nyelvi elemenként.

3.3.1. Szegmentálás

A szegmentáló modellünk esetén a tokenenkénti pontosságot a helyesen címkézett sorok arányával értelmezzük, tekintve, hogy minden sor saját címkével van ellátva a modell kimenetében. Hasonlóan a tökéletes címkézés az egész címkézett blokkra van számítva. A címkék, ellentétben a korábbi munkákkal, nem csak a darabolás helyeit határozták meg, hanem azt is, hogy az egyes sorok milyen jellegű C kódrészlethez kapcsolódnak. Modellünk a 99,9%-os tokenenkénti pontosságot és 99,1%-os tökéletes címkézési arányt ért el.

3.3.2. Kódvisszafejtés

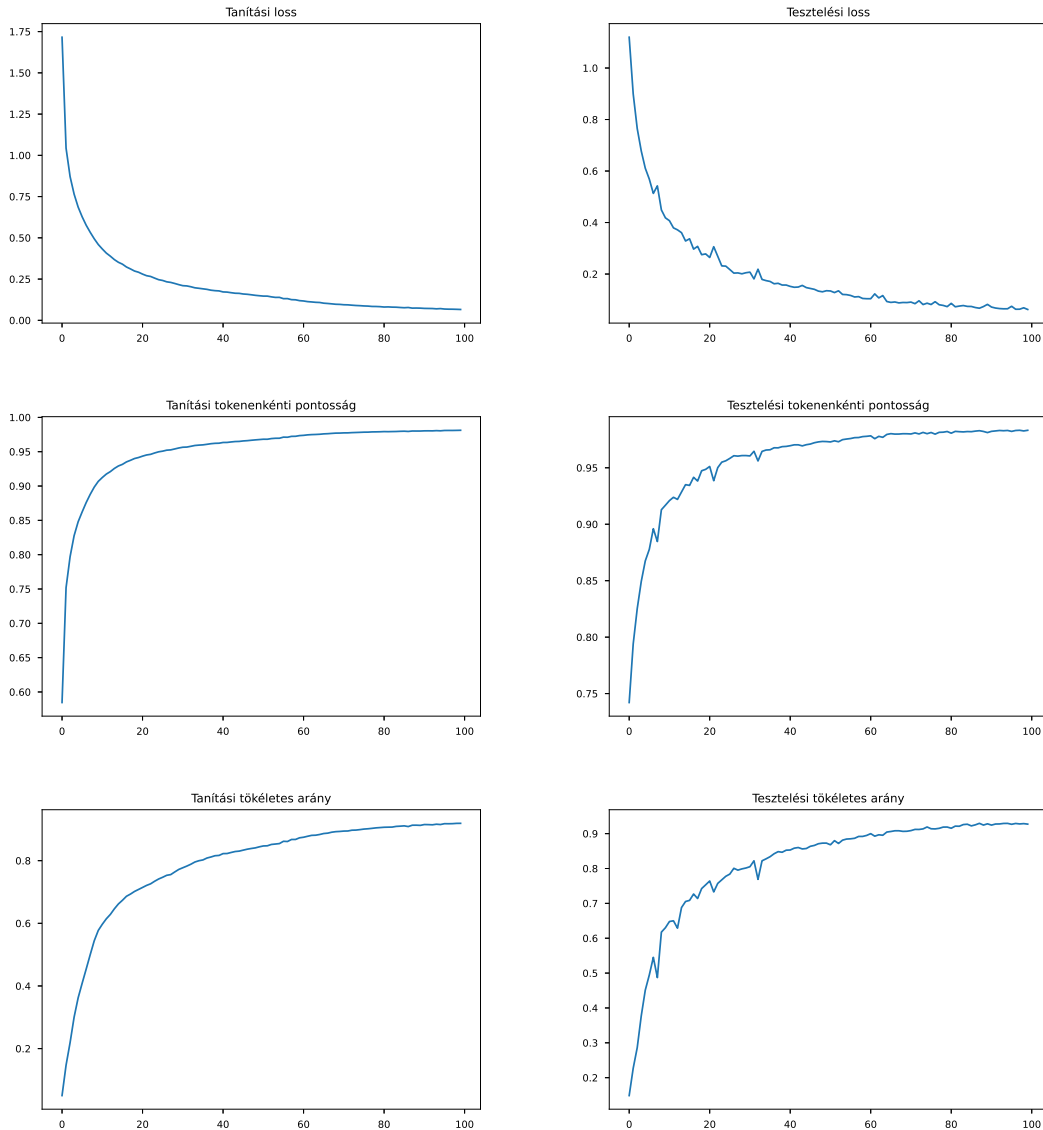
Nyelvi elem	Tokenenkénti pontosság	Tökéletes kódvisszafejtési arány
Statement	98,0%	92,4%
If	99,8%	99,4%
While	99,9%	99,4%
Do-while	99,8%	99,2%
Switch	96,9%	76,3%
Inseparable	93,3%	74,0%
Other	100,0%	100,0%

3.4. táblázat. A kódvisszafejtő modell eredményei az egyes címkéken.

A kódvisszafejtés feladatára sikeresen betanítottunk egy modellt, mely az összes típusú (címkéjű) assembly szegmensen tanult. Ezt a modellt 100 epoch-on keresztül tanítottuk, ennek végén 98,3%-os tokenenkénti pontosságot és 92,9%-os tökéletes

visszafordítási arányt ért el. Modellünk eredményei az egyes nyelvi elemeken a 3.4 táblázatban olvashatók.

A tanítás során a modellünk tanítási és tesztelési mérőszámainak alakulása az alábbi ábrákon (3.1) láthatók.



3.1. ábra. A tanítási és tesztelési eredmények

3.3.3. Kódvisszafejtés több modellel

A kódvisszafejtéshez használt egyetlen modell nem használja fel a szegmentáló által adott címkéket így azok csak a darabolás folyamatában játszanak szerepet. Ebből kifolyólag nem veszi figyelembe, hogy az egyes részek a szegmentáló szerint C nyelv szinten milyen kategóriába lettek sorolva. Ezen információ felhasználása érde-

Nyelvi elem	Tokenenkénti pontosság	Tökéletes kódvisszafejtési arány
Statement	98,0%	94,3%
If	99,4%	98,4%
While	99,5%	96,8%
Do-while	99,4%	97,3%
Switch	98,9%	94,2%
Inseparable	98,9%	95,9%
Other	100,0%	100,0%

3.5. táblázat. Az egyes címkékhez tartozó modellek eredményei.

kében betanítottunk 7 különböző, kisebb kódvisszafejtő modellt, melyek csak adott címkéhez tartozó assembly szegmenseken tanultak. A modellek hiperparaméterei eltérhetnek a többihez képest, viszont a modell architektúra megegyezik mindegyik esetén az egy modelles esettel. Hasonlóan, ahol szükséges volt, speciálisan generált adatokkal kiegészítettük a tanítópéldák halmazát. Ezen modellek eredményeit a 3.5 táblázatban részletezzük.

A modellek teljesítményét az utolsó 10 epoch-ban az alábbi ábrák (3.2) szemléltetik.

3.3.4. Valós példák

Modelljeinket valós kód példákon is kiértékeljük. Tesztpéldáinkat a Codeforces¹ oldalról gyűjtöttük. Ezen adathalmaz megegyezik a korábbi munkánk [7] során használt valós kódokkal.

A példakódok közül kiszűrtük azokat, melyek csak általunk támogatott nyelvi elemeket tartalmaztak. A szűrt példákból eltávolítottuk a beolvasó és kiíró függvényhívásokat, a `for` ciklusokat velük megegyező `while` ciklusokra cseréltük le.

A létrejövő 1004 darab példát ezután lefordítottuk assemblyre a `gcc` segítségével, majd a szegmentáló modellünkkel feldaraboltuk. A létrejövő darabokat a kódvisszafejtő modellünk részenként RPN alakra hozta, ezeket ezután a beágyazott blokkok behelyettesítésével összefűztük. Az így létrejövő RPN kódot végül összehasonlítottuk az eredeti C programból készített RPN kóddal. Rendszerünk eredménye 60,2%-os tökéletes visszaalakítási arány.

¹<http://codeforces.com/> 2022



3.2. ábra. Tanítás és tesztelés a címkénkénti modelleken – az utolsó 10 epoch eredményei

4. fejezet

Diszkusszió

Korábbi munkánk során a leghosszabb sikeresen visszaalakított assembly kód hossza 908 token volt, viszont a hasonlóan hosszú, illetve ennél hosszabb példák esetén jelentősen lecsökkent a rendszer hatásfoka. Ezzel szemben jelenlegi munkánk során hosszabb kódok is kezelhetővé váltak, mivel a teljes kódhossz helyett az egyes szegmensek méretére terelődött át a hossz limitáció. A leghosszabb kezelt szegmens kevesebb, mint 260 tokent tartalmazott, így a modellünk jobb eredményeket tudott elérni. Csak a *Switch* és az *Inseparable* utasítások esetén fordultak elő 170 tokennél hosszabb példák, így a leggyakrabban használt nyelvi elemek esetében a hosszlimitáció kezelése kijelenthető.

Bevezettük a hátul tesztelő ciklusok támogatását, melyeket a szegmentáló modell sikeresen megtanult kezelni. Ezen ciklusok csupán egyetlen utasítással térnek el az elől tesztelős ciklusoktól, a kódvisszafejtő modellünk mégis képes volt ezt megtanulni és megkülönböztetni a két vezérlési szerkezetet assembly szinten.

Szegmentálás után az egyes nyelvi elemekre külön kimértük és betanítottuk a modellünket, az így létrejövő rendszerek a nehezebb vagy ritkább nyelvi elemeken a vártak megfelelően jobban tudtak teljesíteni, közel hibátlan eredményt tudtak nyújtani az adott nyelvi elemeken. Az egyszerű vezérlési szerkezetek esetén a kisebb modellek teljesítményében enyhe visszaesés látható, mert a címkéktől független, általános szabályok tanulását a modell mérete és a példák alacsonyabb diverzitása megnehezítette.

Korábbi munkánkhoz képest 9,2 százalékponttal jobb tökéletes visszaalakítási arányt értünk el valós kód példákon annak ellenére, hogy a támogatott nyelvi elemeket jelentősen bővítettük munkánk során. A szegmentálásnak köszönhetően sikerült

az eddigi legnehezebbnek vélt nyelvi elemek (egymásba ágyazott kétoperandusú művelet, switch) visszaalakítása esetén jelentős javulást elérni.

Méréseink során néhány gyakoribb hibát észleltünk a modellek kapcsán, a továbbiakban ezeket ismertetjük.

Vezérlési szerkezet feltételében lévő mellékhatásos prefix unáris operátorok esetén ($++$, $--$) gyakran külön utasításként értelmezi ezeket a szegmentáló modell, így az elvárt kimenettől eltérő, ámbár azonos működésű C kódot fordít vissza a rendszer.

A kétoperandusú kommutatív operátorok operandusait gyakran megcseréli a kódvisszafejtő modell visszaalakítás során. A kapott eredmény jelentésben azonos az elvárttal, viszont a mérések során hibának számítjuk ezen eltéréseket.

Emellett, ahogy a 3.4 táblázatban is látható, az *Inseparable* nyelvi elem jelentősen nehezebbnek bizonyult a többi nyelvi elemhez képest kódvisszafejtés terén mind az általános, mind a csak erre tanított modell esetén.

Amennyiben a szegmentáló modell hibásan darabolja fel a bemenetet, úgy az esetek döntő többségében a kódvisszafejtő modell már nem képes a szegmensből az elvárt kimenethez hasonlóat visszaadni. Ez a limitáció a szegmentáló modell magas tökéletes címkézési aránya miatt azonban csak korlátozottan befolyásolja a rendszert.

5. fejezet

Összefoglalás

Készítettünk egy *single-pass end-to-end* kódvisszafejtő rendszert másoló mechanizmussal [8] és rekurzív szegmentálással. Előbbinek köszönhetően a kódvisszafejtő modellünk képes helyesen prediktálni az azonosítókat és a literálokat, míg a rekurzív szegmentálás megoldást jelent a korábbiakban jellemző hossz limitáció problémájára.

Eredményeink megmutatták, hogy sikeresen lehet egyszerű mélytanulási eszközökkel alacsony szintű kódot feldarabolni, majd a darabokat visszaalakítani magas szintű kódra. Módszereinkben nem használtunk utólagos hibajavítási technikákat, ezeket a másoló mechanizmus [8], a rekurzív szegmentáció és az assembly nyelvhez igazodó feldolgozás váltotta ki. Kijelenthető, hogy utólagos hibajavítás nélkül modellünk felülmúlta a korábbi munkák utólagos hibajavítási technikák alkalmazása nélkül vett eredményeit, hibajavítással vett eredményeit pedig elérte.

Modelljeink megkapták a feldolgozandó bemenetet soronkénti feldolgozásra is, lehetővé téve ezáltal, hogy a modellek a tényleges információra tudjanak fókuszálni.

A korábbi munkánkhöz képest [7] több új nyelvi elem támogatását is bevezettük: *do-while* ciklus, *switch* esetén keletkező *jump table*, *augmented* értékadások.

Ezek eredményeként sikerült 9,2 százalékponttal növelni a tökéletes visszaalakítási arányt valós kód példákon (51%-ról 60,2%-ra).

A továbbiakban tervezzük a támogatott nyelvi elemek halmazát tovább bővíteni, illetve egy alkalmazást készíteni, mely integrálható fejlesztői környezetekbe, ezzel további valós példákat gyűjtve és a felhasználók számára elérhetővé téve az eredményeinket.

Köszönetnyilvánítás

Köszönjük témavezetőink segítő útmutatását.

A Kulturális és Innovációs Minisztérium ÚNKP-23-2 kódszámú Új Nemzeti Kiválóság Programjának a Nemzeti Kutatási, Fejlesztési és Innovációs Alapból finanszírozott szakmai támogatásával készült.



NEMZETI KUTATÁSI, FEJLESZTÉSI
ÉS INNOVÁCIÓS HIVATAL

Irodalomjegyzék

- [1] Deborah S. Katz, Jason Ruchti és Eric Schulte. „Using recurrent neural networks for decompilation”. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2018, 346–356. old. DOI: 10.1109/SANER.2018.8330222.
- [2] Omer Katz és tsai. „Towards Neural Decompilation”. *CoRR* abs/1905.08325 (2019). arXiv: 1905.08325. URL: <http://arxiv.org/abs/1905.08325>.
- [3] Cheng Fu és tsai. „Coda: An End-to-End Neural Program Decompiler”. 32 (2019). Szerk. H. Wallach és tsai. URL: <https://proceedings.neurips.cc/paper/2019/file/093b60fd0557804c8ba0cbf1453da22f-Paper.pdf>.
- [4] Ruigang Liang és tsai. „Neutron: an attention-based neural decompiler”. *Cybersecurity* 4.1 (2021), 1–13. old. DOI: 10.1186/s42400-021-00070-0.
- [5] Ruigang Liang és tsai. „Semantics-Recovering Decompilation through Neural Machine Translation”. *CoRR* abs/2112.15491 (2021). arXiv: 2112.15491. URL: <https://arxiv.org/abs/2112.15491>.
- [6] Iman Hosseini és Brendan Dolan-Gavitt. „Beyond the C: Retargetable Decompilation using Neural Machine Translation”. *arXiv preprint arXiv:2212.08950* (2022).
- [7] Gergő Szalay és tsai. „Neurális kódvisszafejtés másoló mechanizmussal”. (2022).
- [8] Jiatao Gu és tsai. „Incorporating Copying Mechanism in Sequence-to-Sequence Learning”. *CoRR* abs/1603.06393 (2016). arXiv: 1603.06393. URL: <http://arxiv.org/abs/1603.06393>.
- [9] Warren S McCulloch és Walter Pitts. „A logical calculus of the ideas immanent in nervous activity”. *The bulletin of mathematical biophysics* 5 (1943), 115–133. old.

- [10] Ilya Sutskever, Oriol Vinyals és Quoc V Le. „Sequence to Sequence Learning with Neural Networks”. *Advances in Neural Information Processing Systems*. Szerk. Z. Ghahramani és tsai. 27. köt. Curran Associates, Inc., 2014. URL: <https://proceedings.neurips.cc/paper/2014/file/a14ac55a4f27472c5d894ec1c3c743d2-Paper.pdf>.
- [11] Sepp Hochreiter és Jürgen Schmidhuber. „Long short-term memory”. *Neural computation* 9.8 (1997), 1735–1780. old.
- [12] KyungHyun Cho és tsai. „On the Properties of Neural Machine Translation: Encoder-Decoder Approaches”. *CoRR* abs/1409.1259 (2014). arXiv: 1409.1259. URL: <http://arxiv.org/abs/1409.1259>.
- [13] Dzmitry Bahdanau, Kyunghyun Cho és Yoshua Bengio. „Neural Machine Translation by Jointly Learning to Align and Translate”. (2015). Szerk. Yoshua Bengio és Yann LeCun. URL: <http://arxiv.org/abs/1409.0473>.
- [14] Ashish Vaswani és tsai. „Attention is All you Need”. *Advances in Neural Information Processing Systems*. Szerk. I. Guyon és tsai. 30. köt. Curran Associates, Inc., 2017. URL: <https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf>.
- [15] Eli Bendersky. *pyparser*. <https://github.com/eliben/pyparser>. 2021.
- [16] Diederik P. Kingma és Jimmy Ba. „Adam: A Method for Stochastic Optimization”. (2015). Szerk. Yoshua Bengio és Yann LeCun. URL: <http://arxiv.org/abs/1412.6980>.