

TDK-dolgozat

Póór Máté Bálint

Szalay Gergő

Neurális kódvisszafejtés másoló mechanizmussal

EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

PROGRAMOZÁSELMÉLET ÉS SZOFTVERTECHNOLÓGIAI TANSZÉK



Szerzők:

Poór Máté Bálint

programtervező informatikus BSc.

3. évfolyam

Szalay Gergő

programtervező informatikus MSc.

1. évfolyam

Témavezetők:

Pintér Balázs

egyetemi adjunktus

Gregorics Tibor

tanszékvezető egyetemi docens

Budapest, 2022. december 4.

Tartalomjegyzék

1. Bevezetés	1
1.1. Mélytanulás	2
1.2. Neurális kódvisszafejtés	5
1.3. Saját ötletünk	6
2. Kapcsolódó munkák	9
3. Módszerek	12
3.1. Adathalmazok	12
3.2. Előfeldolgozás	13
3.2.1. Első típusú optimalizációk	14
3.2.2. Második típusú optimalizációk	15
3.2.3. Harmadik típusú optimalizációk	15
3.2.4. A generált példák szűrése	16
3.3. A használt modell	18
3.3.1. Encoder	19
3.3.2. Decoder	19
3.3.3. A hibafüggvény és a hiperparaméterek	20
3.4. C kód előállítás	21
4. Kiértékelés	23
4.1. Módszerek a teljesítmény mérésére	23
4.2. Teljesítmény a bonyolultsági szinteken	24
4.3. Teljesítmény az egyes nyelvi elemeken	25
4.4. Teljesítmény egymásba ágyazott bináris operátorok nélkül	26
4.5. Eredmények valós adatpéldákon	26
5. Diskusszió	29

6. Összefoglalás	31
Irodalomjegyzék	33
A. Mellékletek	35

Kivonat

A kódvisszafejtés (decompilation) célja, hogy egy gépközeleti nyelvű kódrészletet magasabb absztrakciós szintű nyelvű kóddá alakítsunk, ami könnyebben értelmezhető emberek számára.

A hagyományos kódvisszafejtő eszközök tárgyszakértők által megállapított, kézzel írt szabályokon alapulnak, így fejlesztésük, karbantartásuk és bővítésük lassú, időigényes. Ezen problémák megoldására az utóbbi évtizedben több, mélytanuláson alapuló kezdeményezés is született, melyek assembly kódot próbáltak meg C programozási nyelven írt kódra visszaalakítani. Az általuk készített architektúrák nem voltak képesek maradéktalanul kezelni a literálokat és az azonosítókat, így különféle utólagos hibajavítási technikákat kellett alkalmazniuk.

Az utólagos hibajavítás elkerülésére előálltunk egy end-to-end, single-pass kódvisszafejtő eszközzel, mely a másoló mechanizmus (copying mechanism) felhasználásával alakítja vissza az alacsony absztrakciós szintű kódot. A másoló mechanizmus lehetővé teszi tokenek (szavak) közvetlen másolását a bemeneti adatokból. A módszer alkalmazásával az utólagos hibajavítások elkerülhetők, hiszen már elsőre a megfelelő tokeneket tudja generálni a rendszer.

Az architektúránk hatékonyan alakította át a bemenetet idiomatikus, C programozási nyelven írt kódra a regiszterek imitálása és goto utasítások használata nélkül. Korábbi munkákhoz képest három új nyelvi elemmel bővítettük a modellünk által támogatott nyelvi elemek listáját. A modell teljesítményét egy 10 szintből álló, fokozatosan nehezedő szintrendszeren mértük. Emellett külön is megvizsgáltuk, hogy milyen hatékonyan tudja az egyes nyelvi elemeket visszafejteni. Az eredményeink bizonyítják, hogy utólagos hibajavítás nélkül is lehet készíteni neurális kódvisszafejtő rendszert, mely hatékonyan képes helyes eredményt adni.

1. fejezet

Bevezetés

A kódvisszafejtés lényege, hogy alacsony absztrakciós szintű kódot alakítunk vissza magas absztrakciós szintű, ember számára is könnyen értelmezhető nyelven írt kódra úgy, hogy a két kód jelentése, működése megegyezzen. Ez a folyamat a fordítóprogram munkájának (fordítás) inverz folyamatként is elképzelhető. A kódvisszafejtésnek több, a való életben is jelentős felhasználási módja van. Legfontosabbak ezek közül a rosszindulatú szoftverek beazonosítása, sebezhetőségek felismerése, sérült vagy elvesztett programkód visszaszerzése.

A hagyományos (mélytanulási technikákat nem alkalmazó) kódvisszafejtő eszközöket tapasztalt tárgyszakértők fejlesztik. Ezen programok mintákat keresve törekednek a legnagyobb pontosság elérésére egy adott fordító esetén, melynek következménye rengeteg kézzel írt szabály alkalmazása. A sok szabály nagy mértékben lerontja a kódvisszafejtő alkalmazás implementációjának átláthatóságát, karbantarthatóságát és bővíthetőségét, emiatt fejlesztésük rendkívül költséges és lassú. Néhány a legfontosabb hagyományos kódvisszafejtő rendszerek közül: IDA Pro¹, RetDec², Radare2³. Ezek a kódvisszafejtő szoftverek bizonyítottan jól teljesítenek [1], viszont általános hiányosságai is vannak:

- Csak azok a kódsémák alakíthatóak vissza, melyekhez visszaalakító szabályok készültek.
- A generált kód gyakran nem idiomatikus, az alacsony szintű kódban előforduló részletek jelennek meg benne, mint például regiszterek és ugrások használata.

¹<https://hex-rays.com/ida-pro/> 2022

²<https://retdec.com/> 2022

³<https://rada.re/n/> 2022

- A hagyományos kódvisszafejtők általában csak egy-egy fordítóprogramhoz (azoknak is gyakran csak bizonyos verzióihoz) lettek elkészítve, így nem képesek más fordítóprogramokkal vagy verziókkal dolgozni, mert ezekben az esetekben gyakran hibás kimenetet adnak vissza.

Ezen limitációk kiküszöbölésére a kutatók egy csoportja a mélytanuláshoz és az általa nyújtott lehetőségekhez fordult.

1.1. Mélytanulás

A mélytanulás alkalmazása egyre elterjedtebb a fordítási feladatok terén. Az elmúlt évtizedben több mélytanuláson alapuló, a fenti problémák megoldására törekvő cikk is megjelent, melyek C programozási nyelven írt kódra próbálták meg alacsony szintű kódot visszaalakítani neurális gépi fordítás (neural machine translation, NMT) segítségével.

A **mélytanulás** alapegysége a **mesterséges neuron**, mely egy bemenő számértékekből kimenő értéket számoló egység.

A neuronok összekapcsolásával neurális hálókat készíthetünk, melyek sokkal bonyolultabb számításokra is alkalmasak. A hagyományos módszer a neuronok összekötésére az, hogy rétegekbe rendezzük őket, ahol minden réteg felfogható egy oszlopnyi neuronként. Ezután a szomszédos rétegeket kötjük össze egymással úgy, hogy minden neuront az egyik rétegből összekapcsolunk mindegyikkel a másikkól, így egy úgynevezett sűrű neurális hálót (dense neural network, DNN) kapunk. Az első és az utolsó réteg a bemeneti, valamint a kimeneti réteg, a köztes rétegeket pedig rejtett rétegeknek (hidden layers) nevezzük.

A neuronok közti kapcsolatokhoz tartoznak súlyok is, melyek valós számok, és szorzótényezőként funkcionálnak: az első rétegben levő neuron kimenetét (x_i , egy valós szám) megszorozzuk a kapcsolathoz tartozó súllyal (w_i), és ez lesz a második rétegben levő neuron ebből a konkrét kapcsolatból származó bemenete. Természetesen több bemenete is van egy neuronnak, ha az előtte levő rétegben több neuron is található: pontosan annyi, mint amennyi azok száma. A tényleges kimenet (z) ezek szorzatösszege lesz a kapcsolatokhoz tartozó súllyal:

$$z = \sum_{i=1}^n w_i \cdot x_i + b = \mathbf{w}^T \mathbf{x} + b$$

Speciális paraméterként megjelenik a b eltolási konstans, mely a neuron egyedi súlya. Jól látható, hogy ezek a számítások vektorokkal, azok közötti lineáris algebrai műveletekkel is definiálhatók, ekkor a $\mathbf{w}^T \mathbf{x}$ felfogható úgy, mint $\mathbf{w}$ és $\mathbf{x}$ oszlopvektorok skaláris szorzata.

Azonban ha csak lineáris műveleteket alkalmaznánk, akkor az matematikailag átalakítható lenne úgy, mintha csak egyetlen réteget használtunk volna, azaz így a modellünk nem lenne elég komplex ahhoz, hogy jól teljesítsen bonyolultabb problémákon is. Ha szeretnénk a legtöbbet kihozni az architektúrából, akkor valamilyen nemlineáris aktivációs függvényt is használnunk kell mielőtt továbbadjuk egy neuron kimenetét a következő réteg számára.

Egy g nemlineáris függvény alkalmazása mellett a képlet a következőképpen módosul:

$$a = g(\mathbf{w}^T \mathbf{x} + b)$$

A neurális háló eredményeinek kiszámítása azt jelenti, hogy egy olyan algoritmust alkalmazunk az architektúrán, ami a bemenetként kapott adatokra kiszámítja az első réteg eredményeit, ezt továbbadja a következő rétegnek, és ily módon halad tovább a számítás egészen a háló utolsó rétegéig, amiből megkapjuk a végleges kimenetet. Ezt a módszert szokás forward passnek is hívni. Általában az adott kimenet nem egyezik meg az elvárt kimenettel, és a kettő közti eltérést (hibát) fel tudjuk használni arra, hogy utána kis mértékben módosítsuk a hálóban levő súlyokat annak érdekében, hogy ha később hasonló bemenetet lát a hálózat, akkor jobb (kisebb hibájú) megoldást adjon. Ez úgy történik, hogy a háló utolsó rétegétől elkezdünk hátrafelé haladni egészen az elsőig, és minden rétegben elvégezzük a súlyváltoztatásokat. A hátrafelé haladó módszer neve backpropagation. Ebből a leírásból már érzékelhető, hogy ha azt szeretnénk, hogy egy problémára kis hibával rendelkező megoldást adjon a program általánosságban is, akkor ahhoz rengeteg tanító példára (bemenet és elvárt kimenet párja) van szükségünk. A tanítás tehát abból áll, hogy a tanító példákból álló tanító halmaz elemeit egymás után odaadjuk a hálónak, akár többször is, egészen addig, amíg a háló látszólag már nem tudja csökkenteni a hibát. Annak érdekében, hogy a tanítás hatásfokát, sebességét növeljük, egyszerre több tanítási példát is oda lehet adni a modellnek, melyeket az párhuzamosan dolgoz fel. Az így megadott példatömböket batch-eknek hívjuk.

A **seq2seq** (sequence-to-sequence) [2] architektúra egy bemenetként kapott szekvenciát egy másik szekvenciává alakít át. Két fő komponense az encoder és a decoder, melyek a bemenetet dolgozzák fel, valamint a kimenetet generálják le. Az encoder a bemenetet egy magasabb dimenziójú reprezentációvá alakítja, a decoder pedig képes ennek a vissza-

fejtésére egy másik szókészletet alkalmazó szekvenciává. Mindkettőnek van egy saját szókészlete, amiben az általuk értelmezett szavakat tárolják.

Az egyszerű RNN (rekurrens neurális háló) egy olyan hálózati architektúra, mely szekvenciális bemeneteket (pl. mondatok szavai, idősorok) képes kezelni. Fontos kiemelni, hogy az egyszerű RNN mellett más RNN-ek is léteznek, például a GRU-k [3] (gated recurrent unit) vagy az LSTM-ek [4] (long short-term memory). Ezeket később részletesebben is tárgyaljuk. Az egyszerű RNN a bemenetet tokenenként (szavanként) kezeli, és minden token feldolgozása után (a t. időpillanatban) generál egy kimeneti tokent. A t. időpillanatban a háló a t. bemeneti tokent, valamint a rejtett reprezentációt (egy vektor) felhasználva generálja a t. kimeneti tokent. A rejtett reprezentáció minden lépés után változik a korábbi bemenetek alapján. A háló akkor áll le, amikor egy mondat végét jelző szót generált, és ekkor az összes eddig generált szó lesz a teljes kimenet. Fontos megjegyezni, hogy az architektúra mindig ugyanannyi kimeneti tokent fog generálni, mint amennyi bemeneti tokent kapott, de nem feltétlenül nézi végig az egész bemenetet, ha korábban is mondat vége jelet generál. A sűrű neurális hálókkal ellentétben képes kezelni változó hosszúságú bemenetet is, és több-kevesebb sikerrel képes a korábbi tokenekre is emlékezni, azokat is számításba venni a jelenlegi token értelmezésénél.

RNN-eket lehet seq2seq architektúra encoderén és decoderén belül is alkalmazni. Ez azért lehet előnyös, mert önmagukban az RNN-ek nem képesek a bemenettől eltérő hosszúságú kimenetet generálni, a seq2seq azonban igen.

Egyszerű RNN-ek nem alkalmazhatók hatékonyan seq2seq részeként, mert az emlékező képességük nagyon limitált. Csupán néhány tokenrel korábbi dolgokra emlékeznek, így például egy hosszabb mondat végén nem tudnak olyan dolgokra hivatkozni, amik a mondat elején voltak megemlítve.

Ennek az egyik oka, hogy az egyszerű RNN-ek esetén minden időpillanatban beszorozzuk az előző időpillanat rejtett reprezentációját a súlymátrixszal. Egy sok lépéssel korábbi bemeneti token információjának végig kell mennie az összes azóta elvégzett lépésen, mire eljut a jelenlegi időpillanathoz. Ha a súlymátrix sok kis (nullához közeli) számot tartalmaz, akkor ezek ismételt összeszorozása még kisebb számokat eredményez, így ez az információ könnyen elveszik.

Egy másik oka a gradiens eltűnés problémája. Az egyszerű RNN-ek esetében a backpropagation, azaz a súlyok frissítése úgy történik, hogy az RNN által generált kimenetet visszaterjesztjük az időben. Egy adott időpillanatban a súlyok frissítését a deriváltmátrix segítségével tesszük meg. Mivel a backpropagation az utolsó időpillanattól

halad az elsőig, így a deriváltmátrix meghatározásához felhasználjuk a későbbi időpillanatokat is. A gyakorlatban ez is sok mátrix összeszorzásával történik, azaz itt is fennáll a veszélye annak, hogy a kis számok összeszorzásával még kisebb számokat kapunk. Ha ez megtörténik, akkor egy pár lépés után már annyira közel lesznek a nullához a deriváltmátrix elemei, hogy alig változnak a súlyok, és a hálózat nem fog tudni tanulni az adatokból.

A fenti emlékezési és gradiens problémákra nyújtanak megoldást az LSTM-ek [4] és a GRU-k [3]. Ezek az RNN architektúrák sokkal jobb emlékező képességgel rendelkeznek egy egyszerű RNN-hez képest és a felépítésüknek köszönhetően az eltűnő gradiens problémáját is csökkentik, emiatt inkább ezeket szokták használni a seq2seq rendszerekben. Minden problémát azonban az így javított seq2seq sem tud megoldani, ugyanis a seq2seq csak limitáltan képes ritkán előforduló szavakat (pl. egy programban használt egyedi számliterálokat) pontosan idézni a bemeneti szövegből. Ennek a megoldása fontos feladat a program visszaalakításának szempontjából, hiszen egy rosszul idézett számliterál vagy azonosító megváltoztathatja a program teljes működését.

1.2. Neurális kódvisszafejtés

A mély neurális hálók (seq2seq) nagyon hatékonyak bizonyultak a gépi fordítás [5] terén. A kódvisszafejtés két nyelv között történő fordításnak tekinthető, így jogosan vetődik fel az a gondolat, hogy az eddig megalkotott gépi fordító eszközöket alkalmazzuk a problémán. A leglényegesebb különbség a kódvisszafejtés és a gépi fordítás hagyományos alkalmazásai között, hogy a kódvisszafejtés esetében a nyelvek formálisak, szemben a gépi fordításban általában alkalmazott természetes nyelvekkel. A formális nyelvek kötöttsége, szabályrendszere bizonyos szempontból könnyebbé is teheti a fordítást, hiszen a rendszernek elegendő a szabályokat megtanulnia. Cserébe viszont egy kis hiba is teljesen elronthatja az utána következő részeket a kimenetben. A modell által vétett hibákat a kutatók eddig különféle utólagos hibajavítási technikák alkalmazásával próbálták orvosolni.

A legkomolyabb kihívást az ezen a területen végzett kutatások esetében az jelentette, hogy megfelelően prediktálják az azonosítókat és a literálokat. Mivel a gépi kódban lévő számok tetszőlegesen változatosak lehetnek, viszont a legtöbbnek nagyon kicsi az előfordulási gyakorisága, így egy egyszerű seq2seq architektúrának esélytelen minden bemenetben megjelenő számliterálhoz a vele megegyező kimeneti számliterált megtanulnia.

A hagyományos gépi fordítási feladatot ellátó modellek, mint az RNN-eket alkalmazó seq2seq mélytanulási architektúra, nem képesek pontos predikcióra a fenti esetekben. D. S. Katz et al. [6] voltak az elsők, akik megpróbálták megoldani a neurális kódvisszafejtés problémáját mélytanulás segítségével. Az ő munkájukat tekintjük alapozó cikknek. Ők egy egyszerű seq2seq hálót [2] használtak. Ezzel az architektúrával képesek voltak viszonylag alacsony Levenshtein távolságot (a hálózat hibájának egy mérőszáma) elérni, de a helyesen visszafordított programok aránya meglehetősen alacsony maradt. Ők csak strukturális hibákat javítottak utólagosan, így a hibásan prediktált változók és literálok nem lettek korrigálva. Az ő munkájuk tekinthető az első eredménynek a területen.

O. Katz et al. [7] javítottak az alapozó cikk eredményein a saját keretrendszerükkel, melynek neve TraFix. Nyolc különböző nehézségi szintet definiáltak a modell hatásfokának mérésére. A hibásan prediktált változók és literálok javítására egy utólagos kereső algoritmust készítettek, mely minden potenciális javítást behelyettesített a programba, az így keletkező forráskódot lefordította, és összehasonlította a bemenettel. Amennyiben ez a kettő nem egyezett meg, akkor tovább próbálgatta a javítási lehetőségek behelyettesítését. Elmondásuk szerint az alapozó cikk sikertelensége az utólagos hibajavítási technikák hiányából fakadt, véleményük szerint a probléma ezek nélkül nem megoldható.

Fu et al. [8] létrehoztak egy két részből álló kódvisszafejtő keretrendszert, mely a Coda névre hallgat. Az első rész egy kódvázatot hoz létre, melyet a második rész javít egy neurális, iteratív hibajavító rendszer segítségével, amely két almodellből áll. A hibadektálásokat és -javításokat egymás után, kis lépésekben tették meg az iteratív hibajavító rendszerrel.

1.3. Saját ötletünk

Ahogy azt fent láttuk, a korábbi munkák egy közös problémába ütköztek: a helyes változók és literálok prediktálása meglehetősen nehéz egyetlen end-to-end és single-pass modellel. Egy modell akkor single-pass, ha a bemenet csupán egyszer halad át a rendszeren, azaz nem használ iteratív módszereket. Az end-to-end pedig azt jelenti, hogy az egész problémát egy mélytanuló rendszer oldja meg az elejétől a végéig, hagyományos utófeldolgozás, javítás nélkül. A feladat jelentőségét fejezi ki, hogy a TraFix szerzői szerint ez lehetetlen. A problémát a kutatók különböző kiegészítő komponensek és modellek alkalmazásával próbálták megoldani, melyek utólagos hibajavítási módszereket biztosíta-

nak. Ez alól egyedül az alapozó cikk a kivétel, ami csak strukturális javítási technikákat alkalmazott, az azonosítók és literálok javításával nem foglalkozott.

Egy single-pass, end-to-end modell sokkal hatékonyabb lenne a fentiekhez képest: ha egy modell elsőre is képes lenne helyesen prediktálni a megfelelő változókat és literálokat, akkor nem lenne szükség a különböző javítási technikák alkalmazására. Ez azt is jelentené, hogy ezen megoldás sokkal skálázhatóbb lenne, ugyanis nem függne a visszafordítás sebessége az utólagos hibajavítás sebességétől, melynek jelentős időbe is beletelhet a megfelelő megoldás előállítása. Például a gráf-izomorfizmusra épülő javítási módszer [7] vagy az iteratív javítási módszer [8] sebessége nagyban függhet az adatok hosszától.

A feladatot mi egy end-to-end és single-pass modellel közelítjük meg, mely képes hatékonyan gépi kódot visszafejteni, ezzel cáfolva a TraFix [7] szerzőinek korábbi kijelentését. Ennek alapja egy seq2seq háló [2] kiegészítve másoló mechanizmussal (copying mechanism, copy) [9] és attention mechanizmussal [10].

A copy egy olyan technika, mely lehetővé teszi egy token másolását közvetlenül a bemenetről a kimenetre, így a ritkán előforduló azonosítókra vagy literálokra nem a seq2seq-nek kell megpróbálnia visszaemlékezni. Ez azt is jelenti, hogy képes helyes eredményt generálni még akkor is, ha abban egy korábban még nem tanított token szerepel, amennyiben a token másolható és a token kontextusából arra következtet, hogy a tokent másolni kell. Modellünk a copy-t a bemenetben és az elvárt kimenetben is megjelenő literálok, változók és függvénynevek másolására használja.

Emellett alkalmazzuk az attention mechanizmust is. Ez egy olyan módszer, ami képes a seq2seq memóriáján javítani, hogy időben még korábbi dolgokra is képes legyen emlékezni. Ezt úgy éri el, hogy képes kiválasztani azokat a részeket a bemenetből, amikre oda kell figyelni. Az így kiválasztott tokenek ezután nagyobb mértékben lesznek figyelembe véve a következő token generálásakor. Így azokat a tokeneket is pontosabban tudja értelmezni a rendszer, amelyeket nem másolunk a copy segítségével. Ezeknek köszönhetően a single-pass modellünk mellé semmilyen hibajavítási technika sem szükséges (azaz end-to-end is), ez egy hatalmas előrelépést jelent a korábbi architektúrákkal szemben.

A modell eredményességének mérésére a TraFix [7] által bevezetett szintrendszer általunk bővített változatát alkalmazzuk. A 8 nehézségi szintjükhöz képest mi nehezebb nyelvi elemeken is tanítjuk a modellünket, ezáltal még 2 újabb nehézségi szintet hozzáadva az eddigiekhez. Az új nyelvi elemek mind olyanok, amiket a programozók rendszeresen használnak a gyakorlatban, azonban a korábbi megoldások még nem próbálták betanítani. Adathalmazaink a 6. szinttől tartalmazznak ternáris operátorokat, a 9. szinttől

switch utasításokat (case és default ágakkal), a 10. szinttől függvény definíciókat és függvényhívásokat.

Munkánk fő kontribúciói az alábbiak:

- Egy end-to-end, single-pass neurális háló elkészítése kódvisszafejtéshez. Modellünk felhasználja a másoló mechanizmust [9] egy seq2seq architektúra keretén belül, attention mechanizmussal [10] kiegészítve.
- Két nehezebb szint definiálása, melyben lévő nyelvi elemek gyakran előfordulnak a programozók által írt kódokban. Az új nyelvi elemek a switch utasítások, a ternáris operátorok, a függvényhívások és a függvénydefiníciók.
- Modellünk szignifikánsan jobb eredményeket ért el a sikeres kódvisszafejtésben, mint a TraFix [7] a 7. (while ciklusok) és a 8. (egymásba ágyazott elágazások és while ciklusok) szinten anélkül, hogy bármilyen hibajavítási technikát is alkalmaztunk volna.

2. fejezet

Kapcsolódó munkák

Számos korábbi kísérlet született a kódvisszafejtés problémájának megoldására mélytanulás segítségével.

Az első kapcsolódó cikk D. S. Katz et al. [6] kutatásához fűződik. Ők bináris állományok kódját használták a modelljük bemeneteként, melyeket rövid C kódrészletekből fordítottak le, utóbbi jelentette az elvárt kimenetet. A bemenetek átlagos hossza 60 token, míg az elvárt kimenetek átlagos hossza 38 token volt. Az adathalmazuk egy nyílt forráskódú, kézzel implementált C könyvtáron alapult, melyet egy absztrakt szintaxisfá-vá (abstract syntax tree, AST) alakítottak. Az elkészített AST-ről vettek különböző ágakat, mely később az adathalmazuk bemenetét jelentette. Ennek eredményeként több mint 700 000 különböző tanítási példát hoztak létre, mely mind tartalmazott egy bináris kódot és a hozzá tartozó C nyelvű megfelelőjét. A bemenetet bájtonként tokenizálták, emellett az architektúrájuk közvetlenül próbálta prediktálni a C kódot elő- és utófeldolgozási algoritmusok használata nélkül. Annak ellenére, hogy ezzel a technikával egy alacsony átlagos Levenshtein távolságot voltak képesek elérni (0,7), a prediktált kimenetek csupán egy elenyésző hányadát lehetett újra lefordítani bináris kódra.

O. Katz et al. [7] iteratív hibajavítást adtak a keretrendszerükhöz (TraFix) a probléma megoldása érdekében. A tanítási adathalmazukban nyelvi elemek gyakoriságának fenntartása érdekében 10 000 tanítási példát generáltak, melyek egy részét a tanítás során időnként lecseréltek a túltanulás megelőzése végett. Az új adatokat a tanítás közben generálták a modell addigi eredményei alapján: a generált adatok a modell aktuális nehézségeire próbáltak fókuszálni. Az architektúrájukat két független tanítás során megtanították visszafejteni AT&T x86-os szabványú assembly kódot és LLVM IR kódot. Az elvárt kimenetet mindkét betanított modell esetén fordított lengyel jelölésben (reverse polish no-

tation, RPN) adták meg, mely sokkal közelebb áll az assembly kód szerkezetéhez, mint a C megfelelője. A modelljük sikerességének mérésére visszafordították a generált kódot és a bemeneti assembly kóddal vetették össze. Ennek oka, hogy különböző C programok is fordulhatnak ugyanarra az assembly kódra, így a kimenet különbözhetett az általuk elvárt eredménytől, noha ugyanazzal a működéssel bírt. Habár az architektúrájuk képes volt a bemenetek többségét visszafordítani, módosították az assembly kódjukat, így az változókat tartalmazott regiszterek helyett. Emellett csak a vezérlésfolyam-gráfot határozta meg a modelljük, a változók és a literálok, melyek a tokenek jelentős hányadát jelentik, iteratív hibajavítási módszerrel lettek behelyettesítve kézzel írt szabályok alapján.

Fu et al. [8] egy end-to-end megközelítéssel próbálták megoldani a feladatot. Ennek eredménye lett a CODA, mely tartalmazott egy neurális iteratív hibajavító modellt, ezzel korrigálva a rosszul prediktált tokeneket. Az említett háló képes volt megtanulni, hogy mely helyeken kell a hibajavításokat elvégezni, és azt is, hogy milyen jellegű hibajavítást kell ott végezni. Tanítási adathalmazként saját maguk által generált adatokkal dolgoztak. Ezzel ellentétben a tesztelést kézzel írt ciklusmentes példákon végezték. Eredményeik megmutatták, hogy a kétoperandusú műveletek jelentik a legnagyobb nehézséget az elhibázott tokenek után. Hogy elkerüljék a szintaktikus hibákat az eredményt AST formában prediktálták. Saját bevallásuk szerint a hibajavítás több mint 60%-kal emelte az architektúrájuk hatásfokát, ami egyben azt is jelenti, hogy az első modell az esetek jelentős részében hibás kódot generált.

Liang et al. [11] megalkotta a Neutront, mely fő újítása, hogy nem egyben próbálta meg a teljes assembly kódot visszafordítani. Ehelyett először feldarabolta a bemenetet egy osztályozó modell segítségével oly módon, hogy a kisebb részek mind egy-egy értelmes C kódrészletnek feleltethetők meg. Ezek a kódrészletek külön-külön lettek visszafordítva, majd újra összeillesztve adták vissza a teljes függvényt. Annak érdekében, hogy az egész vezérlésfolyamot és ezáltal a függvényt vissza tudják állítani, egy hagyományos kódvisszafejtő eszközt, az *anr*-t [12] használták. Annak eredményeként, hogy feldarabolták az assembly kódot, a Neutron jól kezelte a hosszabb bemeneteket is. A darabolás negatív hatásaként jelent meg, hogy a rendszer rosszul kezelte az optimalizált assembly kódokat, mivel ezekből gyakran hiányoznak a regiszterekbe történő adatmozgatások sebességnövelés céljából. Ez azt jelenti, hogy egyes utasítások újrahasználhatnak értékeket, amik már be lettek töltve a regiszterekbe, így megspórolva az adatmozgatás idejét. Ezek az optimalizációk okozzák a független részekre szegmentálás nehézségét.

Az elmúlt évben, Liang et al. [13] transformer modelleket is alkalmaztak [14] a kód-

visszafejtéshez. A konstansok kezelésének problémája náluk is megjelent, melyet ők a számliterálok kimaszkolásával és későbbi visszahelyettesítésével orvosoltak.

A TraFix, a CODA és a Neutron működésében több hasonlóság is felfedezhető:

- A generált kód szerkezetét mélytanulási módszerekkel alakították vissza. A szintaktikailag helyes programok garantálása érdekében különféle problémáspecifikus utólagos átalakításokat alkalmaztak.
- A hibásan prediktált tokenek problémájára különböző hibajavítási technikákat használtak. Ezekre a módszerekre azért volt szükség, mert a literálok és a változók gyakorisága nagyban megnehezítette a modell számára kezelésük megtanulását.

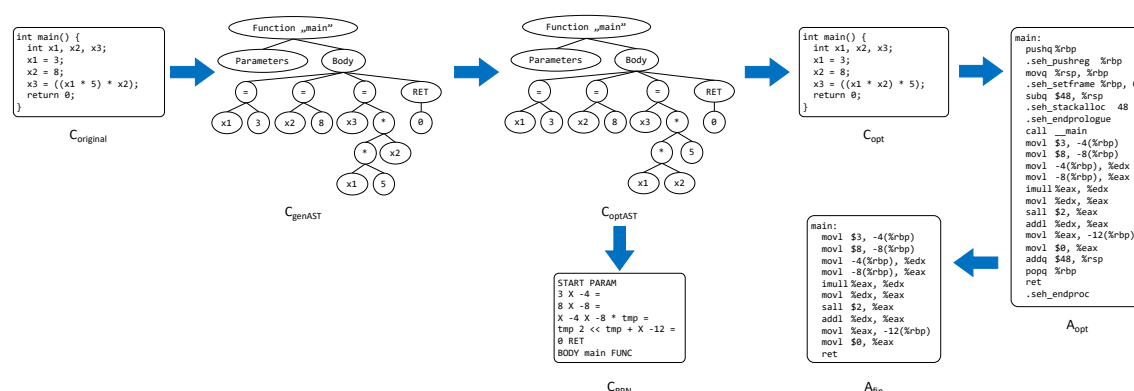
Ezzel szemben a mi módszereink nem használnak iteratív hibajavítási technikákat a hibás literálok és változók javítására, hiszen a mi modellünk már elsőre képes a helyes tokeneket prediktálni a másoló mechanizmusnak köszönhetően (azaz a modellünk single-pass is).

3. fejezet

Módszerek

A következőkben bemutatjuk az adatgenerálásunkat, az előfeldolgozási módszereinket, a modellünk belső működését és a C programozási kód előállítását a modellünk kimenetéből.

3.1. Adathalmazok



3.1. ábra. Egy tanítási példa előállítása. A kezdeti C kódot ($C_{original}$) AST-vé alakítjuk (C_{genAST}), melyet ezután optimalizálunk (C_{optAST}). Az optimalizált AST ezt követően vissza lesz alakítva C kóddá (C_{opt}). Az elvárt kimenet előállításához az optimalizált C_{optAST} -t átalakítjuk ekvivalens módon módosított fordított lengyel jelöléssé (C_{RPN}). Az assembly kód (A_{opt}) C_{opt} fordításaként jön létre, melynek a kódvisszafeltértek tekintetében lényegtelen információt tartalmazó részeit levágjuk, így jön létre a végleges bemenete a modellnek (A_{fin}). A tanítási példában a bemenetet és az elvárt kimenetet adjuk meg: (A_{fin} , C_{RPN}). A fenti párost kiegészítjük az eredeti C kóddal is, így kapjuk a végső adatnak minősülő hármast: ($C_{original}$, C_{RPN} , A_{fin})

Az interneten rengeteg nyilvános forráskódot lehet találni, melyeket fel lehet használni egy neurális kódvisszafeltértek tanítására. Például D. S. Katz et al. [6] egy nyílt forráskódú

programtároló kódbázisát használták fel (Fedora csomagok).

Az emberek által írt kódokban viszont bizonyos nyelvi elemek előfordulása sokkal ritkább, mint a generált kódok esetében, így az ember által írt kódok nem számítanak jó minőségű tanítási példának egy mélyháló számára. Például a nyílt forráskódú adathalmazai a Codának [8] és a Neutronnak [11] (Karel¹, Hacker's Delight ciklusmentes programok [15]) nem tartalmaztak olyan nyelvi elemeket, melyeket a modellünknek tanítottunk, mivel a legtöbb ilyen példa elágazás- és ciklusmentes vagy csak egyszerű függvényhívásokat tartalmaz. Mindez azt jelenti, hogy az említett példák csak az első 4 bonyolultsági szintet fedik le az általunk használt 10 szintes rendszerben.

A fentebb említettek miatt, hasonlóan a TraFixhoz [7], egy adatgenerálót hoztunk létre, mely C kódokat generál és fordít le assemblyre ekvivalens módon.

Mivel a modellünk teljesítményét valós kódokon is szeretnénk volna lemérni, létrehoztunk egy emberek által írt programokat tartalmazó adathalmazt (Codeforces² versenyprogramozási weboldalon található beküldött megoldások). A realisztikus adathalmaznak a részletes leírását és a modellünk ezen elért eredményeit a 4. részben fejtjük ki.

3.2. Előfeldolgozás

Az assembly kód C kódból való előállításához GCC fordítót használtunk [16], mely az AT&T x86-os assembly szabványt alkalmazta. A GCC több száz optimalizációs szabályt alkalmaz fordítás közben, még a -O0 (optimalizációk kikapcsolása) kapcsoló megadása mellett is. Az előfeldolgozás során arra kellett törekednünk, hogy a C kódot ekvivalens transzformációk sorozatával olyan alakra hozzuk, mely a lehető legjobban hasonlít az assembly kódra. Annak érdekében, hogy ezt elérjük, utánoznunk kellett a fordító által használt optimalizációkat. Ezek alapján az optimalizációk három típusát különböztettük meg.

Az első típusú optimalizáció az utasításokon van végrehajtva, ahol a tisztán konstansokkal végzett műveletek kiértékelődnek, illetve olyan alakra transzformáljuk az utasításokat, hogy azok későbbi optimalizációkat lehetővé tegyenek, vagy csökkentsék a futási időt. Ezen optimalizációkra található egy példa a 3.2. ábrán: a zárójelek átrendeződnek, a kommutatív operátorok kiértékelésének sorrendje megváltozik, majd a csak konstansokat tartalmazó rész kiértékelődik.

¹<https://www.cs.mtsu.edu/~untch/karel/> 2022

²<http://codeforces.com/> 2022

A második típusa az optimalizációknak a változók olyan szorzására, osztására, maradékszámítására vonatkozik, melyeknek egyik operandusa konstans érték. Ezek egy jelentős része biteltolásokká alakul át, ezzel növelve a számítások numerikus stabilitását, és a kiértékelés sebességét.

A harmadik optimalizációtípus egyes logikai kifejezések negálása. Speciális esetekben a GCC a logikai kifejezések operátorának az ellentétét használja. A későbbiekben kifejtjük az említett transzformációk szükségességét.

Az előfeldolgozás első lépéseként a kezdeti C kódot ($C_{original}$) átalakítjuk egy absztrakt szintaxisfává (AST) egy parser [17] segítségével (C_{genAST}).

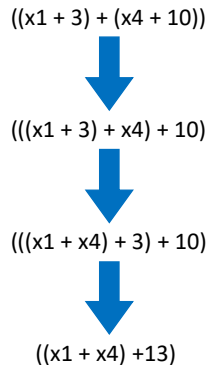
Garantálnunk kellett, hogy ugyanazok a literálok és változók jelennek meg az assembly kódban és a C kódban. Ennek érdekében a GCC fordítás során használt optimalizációs szabályainak jelentős részét implementálnunk kellett. A változók esetében kihasználtuk, hogy a memóriában sorfolytonosan helyezkednek el és mindegyikhez tartozik egy regiszter offset, mely a változó első bájtját indexeli. Tehát minden lokális változó a C kódban megfeleltethető egy regiszter offsetnek az assemblyben, mellyel az elvárt kimenetben a változókat azonosítjuk.

3.2.1. Első típusú optimalizációk

Az első típusú optimalizációk szabályai aritmetikai optimalizációkat hajtanak végre az AST struktúráján és tartalmán. AST alakban ezek a módosítások sokkal egyszerűbben végrehajthatók a példákon, mivel a fa alakja jelzi a kapcsolatot az egyes csúcsok között, szemben a C nyelvű kóddal, ahol a lineáris alakban ezek az információk csak nehezen elérhetőek, habár a szabályok esetében kulcsfontosságúnak bizonyulnak. Fontos megemlíteni, hogy habár ezek az átalakítások mind ekvivalensek, mégsem lehet őket egyértelműen visszaalakítani az eredeti formájukra. Példának okáért a kifejezés (9) lehet egy optimalizáció utáni alakja az $(5 + 4)$ -nek, a $(3 * 3)$ -nak, stb.

A szerkezeti optimalizációs algoritmus (első típusú optimalizációk) akkor ér véget, ha már nem tud több optimalizációt végrehajtani egyetlen részfán se. Miután az AST elérte végleges alakját (C_{optAST}), visszaalakítjuk C programozási nyelven írt kóddá (C_{opt}).

C_{opt} -ot ezt követően lefordítjuk assembly kódra (A_{opt}), melyet ezután kanonikalizálunk. Kanonikalizáció alatt azt értjük, hogy az A_{opt} -ot tokenizáljuk és eltávolítjuk belőle a vesszőket. A kanonikalizáció végén a kódvisszafejtés szempontjából szükségtelen részeit



3.2. ábra. Egy példa, amin látható pár GCC által használt optimalizáció, melyeket mi is alkalmazunk.

az assemblynek (például a program fejlécet illetve a lábfejet) eltávolítjuk. Az így kapott kanonikalizált assembly kód lesz a végső formája a modellünk bemenetének (A_{fin}).

3.2.2. Második típusú optimalizációk

Az előfeldolgozás célja, hogy olyan adatokat hozzunk létre, amelyekben az A_{fin} és az elvárt C kód megegyezik az utasítások jellegében és tartalmában (literálok és változók) amennyire csak lehetséges. Ennek érdekében a C_{optAST} egy posztorder bejárással lesz átalakítva, így elérve a módosított lengyel jelölés formát (C_{RPN}), ahogy a TraFix [7] is csinálta. Ez a módosított RPN megkönnyíti a modell számára, hogy összefüggéseket találjon az assembly és C nyelvű utasítások között, mivel az assembly szerkezete nagy mértékben hasonlít az RPN-re. Egy ilyen RPN-re történő átalakítás látható a 3.3. ábrán. A módosított RPN ekvivalens módon visszaalakítható a kezdeti C alakra (C_{optAST} , majd C_{opt}), így nem veszítünk információt a transzformációval.

A posztorder bejárás közben végrehajtjuk a második típusú optimalizációkat. Ekvivalens transzformációkkal a speciális konstansokkal való szorzást, osztást és maradékszámítást átalakítjuk, hogy az assemblyvel megegyező formát vegyen fel a C kód. Ez azt jelenti, hogy biteltolások és túlcsoordulások jelennek meg a kódban, hasonlóan az assemblyben történő számításokhoz.

3.2.3. Harmadik típusú optimalizációk

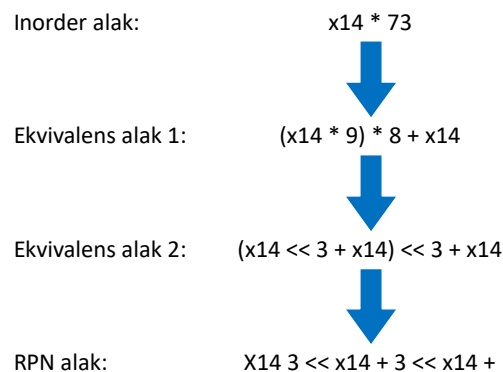
Nem csak aritmetikailag ekvivalens transzformációkat hajtunk végre a posztorder bejárás során. Bizonyos logikai kifejezések operátorát negáljuk annak érdekében, hogy az

assemblyben lévő feltételes ugrások típusával megegyezzenek. Például egy ($<$) reláció átalakulhat ($>=$)-re, ha az assemblyben is `jge` (jump if greater than or equal, azaz ugrás, ha nagyobb vagy egyenlő) utasítás jelenik meg a `j1` (jump if less than, azaz ugrás, ha kisebb) helyett az optimalizáció következtében.

Az assemblyben megjelenő ugrások jelentése nem mindig konzisztens. Ennek oka, hogy különböző módokon kell kezelni a feltételeit egy elágazásnak és egy ciklusnak. Elágazás esetén az assemblyben csak akkor következik be a feltételes ugrás, ha a feltétel hamis, különben a kód a következő sorral folytatódik. Ezzel szemben a ciklusoknál a feltételes ugrás pont fordítva működik, és csak akkor következik be, ha a feltétel igaznak bizonyult. Így a modell tanítása érdekében a ciklusok feltételeiben lévő operátorokat negáltuk az elvárt kimenetben.

Ez a transzformáció egyszerűen visszaalakítható az eredeti alakra, miután a modell végzett a kimenetének prediktálásával. Az átalakítás lehetővé teszi, hogy ezekben az esetekben is kimásolhassuk az operátort a másoló mechanizmus segítségével a bemenetből.

Bár a harmadik típusú optimalizációk egyidejűleg történnek a második típusú optimalizációkkal, mégis megkülönböztetjük őket, mivel a kódon végzett módosítások jellege különbözik bennük.



3.3. ábra. Egy példa a literállal való szorzás ekvivalens módon történő átalakítására biteltolások által.

3.2.4. A generált példák szűrése

Az adathalmazba tevés előtt minden generált példának át kell mennie egy szűrőrendszeren. Amennyiben egy generált adat fennakad a szűrőn, úgy el lesz vetve és nem használjuk fel a későbbiekben. Az alábbi szabályoknak kell megfelelnie a generált adatoknak:

- A literáloknak és darabszámuknak meg kell egyeznie C_{RPN} -ben és A_{fin} -ben. Csak minimális eltérést engedünk meg, ellenkező esetben a példát elvetjük: ha egy adott literál n -szer szerepel az optimalizált C kódban, akkor legalább n -szer kell szerepelnie a végső assembly kódban (A_{fin}) is. Ez a szűrés elengedhetetlen a copy működése szempontjából, ugyanis ha egy literál nem jelenik meg az assembly kódban, viszont az elvárt C kódban megtalálható, úgy az nem tud mit lemásolni, és a modell képtelen a literált prediktálni.
- A logikai kifejezéseket is teszteljük. Ha C_{opt} -ban a különböző relációk ($\geq$, $>$, stb.) és az assemblyben nekik megfelelő tethető feltételes ugrások darabszáma eltér, akkor a példa fennakad a szűrőn és elvetjük.
- Korlátoztuk A_{fin} maximális hosszát 750 tokenre. Ez az érték szignifikánsan magasabb, mint amivel a korábbi munkák dolgoztak, és véleményünk szerint kellően magas, hogy valós programok is beleférjenek. Fontosnak tartjuk megemlíteni, hogy az előfeldolgozás során töröltük a vesszőket és a visszaalakításhoz szükségtelen részeket az assembly kódból, így az eredeti bemeneti alacsony absztrakciós szintű kód sokkal hosszabb volt. A leghosszabb sikeresen visszafordított assembly állomány (előfeldolgozási átalakítások nélkül) 808 token hosszú volt. Ez összevetve a TraFix [7] eredményével, ahol 668 token volt a leghosszabb sikeresen visszaalakított kód tokenszáma, 20%-os javulást jelent.

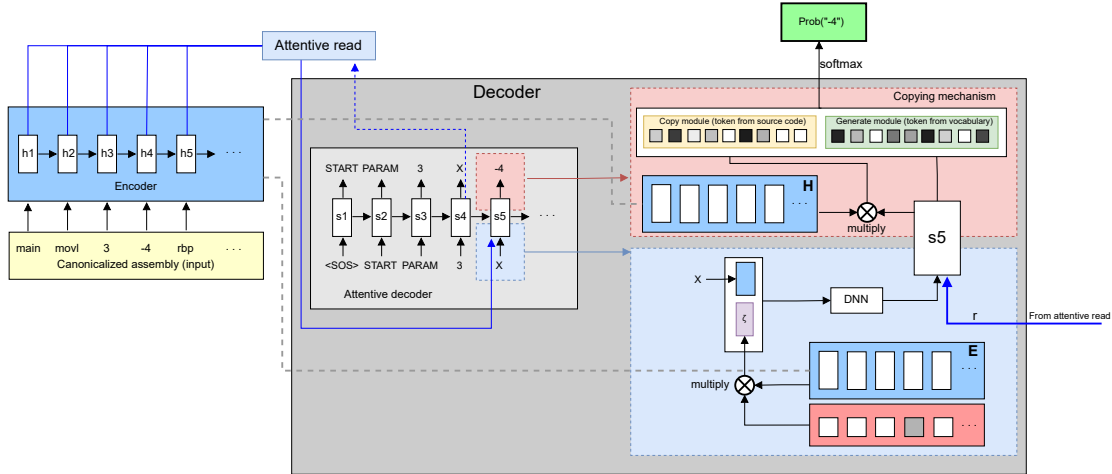
Minden adathalmazunk ($C_{original}$, C_{RPN} , A_{fin}) hármaskból épül fel (beleértve a valódi kódokat tartalmazó adathalmazunkat is). A kezdeti kódot, $C_{original}$, eltároltuk, mivel C_{RPN} -t csak C_{opt} alakra lehet ekvivalens módon egyértelműen visszaalakítani a korábban tárgyalt okok miatt.

A példákat addig generáltuk, amíg a sikeresen generált példák száma el nem érte a kezdetben kijelölt értéket. Amikor ezt a darabszámot elértük, az adathalmazt késznek nyilvánítottuk.

Fontos megemlíteni, hogy a main függvény a példáinkban az összes lehetséges változó definíciójával kezdődik (egész számok (int) X0-tól X19-ig, feltételezésünk szerint az általános kódok nem használnak 20-nál több lokális változót egyetlen függvényben). A változókat mind a volatile kulcsszó segítségével definiáltuk, mely garantálta, hogy az első változó (X0) kapja meg a -4-es regiszter offsetet az assembly kódban, míg a második változó (X1) kapja meg a -8-as regiszter offsetet, stb. Ez biztosította a kapcsolatot a változók nevei és a regiszterek offsetei között, ezáltal lehetővé téve a másolást.

Az egész folyamat a 3.1. ábrán látható.

3.3. A használt modell



3.4. ábra. A modellt, ahogy egy kanonikalizált assembly kódot (bemeneti kódot) dolgoz fel. A bemenetben szereplő tokeneket egyesével dolgozza fel az encoder, mely tartalmaz egy beágyazási réteget és egy rekurrens réteget (GRU). Az attentive decoder tartalmaz selective readet, melynek kimenetét összefűzzük az beágyazási réteg kimenetével. Az összefűzést követően az érték sűrű rétegeknek lesznek átadva. Az attentive decoder GRU rétege kiszámítja az új rejtett reprezentációt a korábbi számított értékek alapján: az attentive read eredményét összefűzzük a sűrű rétegek eredményével, ez alkotja a GRU bemenetét. A GRU által adott új rejtett reprezentáció és az encoder eredményeül kapott rejtett reprezentációk felhasználásával megkapjuk a logitokat mind a copy modulból, mind a generate modulból. A két logit vektort összefűzzük és softmax függvényt alkalmazunk rajta, így megkapva a következő token valószínűségi eloszlását, melyből a legnagyobb valószínűségű token lesz a következő kimeneti token.

A modellünk (3.4. ábra) egy seq2seq háló Bahdanau attentionnel [10] és másoló mechanizmussal [9] kiegészítve. A decoder a modell teljesítményének növelése céljából tartalmaz két különböző attentiont: Bahdanau attention [10] és a másoló mechanizmus [9] saját attentionjét. A decoder egy lépésében a decoder átadja az aktuális rejtett állapotot (hidden) a másoló mechanizmus [9] két moduljának. Ezek a modulok a logitokat prediktálják a másolható és a nem másolható részből a szókészletnek. Softmax nemlineáris függvényt alkalmazva a két logit vektor összefűzésén, megkapjuk a valószínűségeit a következő generált tokennek.

3.3.1. Encoder

Az encoder összetömöríti a bemeneti sorozatban ($X = [x_1, \dots, x_L]$) lévő információt egy beágyazási (embedding) réteg és egy rekurrens réteg (GRU [3])) felhasználásával egy rejtett vektor reprezentációba ($\mathbf{H}$).

A **beágyazási réteg** minden bemenethez egy adott méretű vektort rendel hozzá, a vektorban lévő értékeket a modell tanulja.

$$\mathbf{h}_t, \mathbf{e}_t = \text{GRU}(\text{Emb}(x_t), \mathbf{h}_{t-1}) \quad (3.1)$$

$$\mathbf{H} = [\mathbf{h}_1 \dots \mathbf{h}_L] \quad (3.2)$$

$$\mathbf{E} = [\mathbf{e}_1 \dots \mathbf{e}_L] \quad (3.3)$$

ahol $\mathbf{h}_t$ a rejtett állapot, $\mathbf{e}_t$ pedig az encoder GRU-jának a kimenete a t időpillanatban (3.1 egyenlet).

3.3.2. Decoder

A decoder felhasználja az encoder által összetömörített információt és ez alapján prediktálja a modell kimenetét. A decoderünk tartalmaz egy beágyazási réteget, egy dropout réteget (csak tanításkor használt) és egy rekurrens réteget (GRU [3]), melyeken felül még ki van egészítve Bahdanau attentionnel [10] és másoló mechanizmussal [9] (generate modul és copy modul) és az ahhoz tartozó külön attentionnel. Az említett mechanizmusoknak a lényege, hogy lehetővé teszik a modell számára, hogy a megfelelő tokeneket prediktálja elsőre, ezzel mindenféle utólagos hibajavítási technikát szükségtelenné téve.

A **dropout réteg** egy előre meghatározott p valószínűséggel nullázzuk ki a paraméterként kapott tömb (vektor, tenzor, mátrix) elemeit, a változatlanokat pedig megszorozzuk $\frac{1}{1-p}$ -vel, ezzel az átlagos összérték egy soron/oszlopon belül nem változik.

A Bahdanau attention és a selective read (copy mechanizmushoz külön kifejlesztett attention) az alábbi módon működik:

$$\zeta = \text{Sel}(\mathbf{y}_{t-1}, X, \mathbf{l}_{t-1}, \mathbf{E}) \quad (3.4)$$

$$\mathbf{r} = \text{Attn}(\mathbf{s}_{t-1}, \mathbf{E}) \quad (3.5)$$

ahol ζ a selective read (a copy cikkjében olvasható módon [9]) eredménye, $\mathbf{r}$ az attentive read eredménye (Bahdanau attention [10]), $\mathbf{y}_t$ a prediktált token t időpillanatban (3.9 egyenlet), $\mathbf{s}_t$ a GRU (3.6 egyenlet) rejtett állapota t időpillanatban, $\mathbf{l}_t$ tartalmazza a má-

solási logitokat a bemenetre vonatkoztatva ($\mathbf{l}_{t(i)}$ reprezentálja a logitját az i_{th} tokennek a bemenetből, 3.8 egyenlet). A decoder többi része az alábbiak szerint működik:

$$\mathbf{s}_t = GRU(Emb(\mathbf{y}_{t-1}), \zeta, \mathbf{r}) \quad (3.6)$$

$$\mathbf{g} = G(\mathbf{s}_t) \quad (3.7)$$

$$\mathbf{c}, \mathbf{l}_t = C(X, \mathbf{s}_t, \mathbf{H}) \quad (3.8)$$

$$\mathbf{y}_t = Softmax(\mathbf{g} \oplus \mathbf{c}) \quad (3.9)$$

ahol ζ a selective read (3.4 egyenlet), $\mathbf{r}$ az attentive read (3.5 egyenlet), G és C a generate modul és a copy modul és $\oplus$ jelenti az összefűzést a két eredménynek, melyekből a softmax függvény használatával kapjuk meg az eloszlását a következő tokennek.

Generate modul A generate modul (G) felelős a szókészlet nem másolható szavainak prediktálásáért. Egyetlen DNN-t tartalmaz, így a működése az alábbiak szerint írható le:

$$\mathbf{g} = DNN(\mathbf{s}_t) \quad (3.10)$$

ahol $\mathbf{s}_t$ a decoder hidden állapota, mely megegyezik a 3.6 egyenletben lévővel és a 3.7. egyenletben paraméterként van átadva.

Copy modul A copy modul (C) felelős a szókészlet másolható tokenjeinek prediktálásáért. Egyetlen DNN-t tartalmaz, a működése pedig a következők szerint zajlik:

$$\mathbf{D} = DNN(\mathbf{H}) \quad (3.11)$$

$$\mathbf{l}_t = \mathbf{D} \cdot \mathbf{s}_t \quad (3.12)$$

$$\mathbf{c}_{(i)} = \sum_{j: X_{(j)}=V_{(i)}} \mathbf{l}_{t(j)} \quad (\forall i \in [1, \dots \|V\|]) \quad (3.13)$$

ahol $\mathbf{H}$ az encoder hidden kimeneteit jelöli a 3.2 egyenletből, $\mathbf{s}_t$ a decoder rejtett állapota a 3.6 egyenletből mely a 3.8 egyenletben lett paraméterként átadva, valamint V az X szókészlet azon részhalmaza, mely tartalmazza a másolható tokeneket.

3.3.3. A hibafüggvény és a hiperparaméterek

Hibafüggvényként a cross-entropy loszt használtuk, ennek eredményén futtattuk a backpropagationt is.

A tanítás közben az ADAM [18] adaptív optimalizációs algoritmust használtuk, melynek segítségével a tanulási ráta paraméterenként külön-külön szabályozható, ezzel még

hatékonyabbá téve a tanulás folyamatát. A használt hiperparaméterek a 3.1. táblázatban találhatók.

3.1. táblázat. A modell hiperparaméterei

Név	Érték
Batch méret	32
Hidden méret	512
Begyazási méret	128
Tanulási ráta	0.0001
Dropout ráta	0.4

3.4. C kód előállítás

Miután a modell feldolgozta a bemenetet és befejezte a kimenet generálását, a kapott RPN formát még vissza kell alakítani C programozási nyelven írt kódra. Ennek érdekében az előfeldolgozási lépéseket fordított sorrendben kellett leimplementálni ellentétes működéssel.

Első lépésként a biteltolásokat szűrjük ki, és alakítjuk vissza, melyek eredetileg szám-literálokkal való szorzások, osztások, maradékszámítások lehettek. Ez az átalakítás egyértelműen végezhető el, hiszen az RPN-ben a megfelelő literálok találhatóak meg, melyek az assembly kódból lettek kimásolva. Egy mintaillesztő algoritmust használunk a megfelelő biteltolási kifejezések felismerésére. Az osztások fordítása során az assemblyben utasítások hosszabb sorozata jelenik meg, melyek nagy, látszólag véletlenszerű számokkal dolgoznak. Ezen jellegű osztásokat visszaalakítjuk eredeti alakra az eltolások visszaalakítása előtt.

Miután a konvertálható bitműveleteket visszaalakítottuk, a keletkezett RPN-t egy AST-vé konvertáljuk. Ezalatt a konverzió alatt a módosított logikai kifejezéseket (ismételten negálva a jelentésüket) visszatranszformáljuk eredeti jelentésükre egyértelmű módon.

Az RPN tokenjeit egyesével dolgozzuk fel az AST-vé alakítás során, mindegyik egy új csúcsot eredményez. Ezen csúcsok belekerülnek egy verem adatszerkezetbe. Egyes speciális tokenek (mint például IF vagy WHILE), esetén a korábbi csúcsokat ki kell venni a veremből és összefogni egy új részfává, melyet ezután a verem tetejére teszünk. Miután minden tokent feldolgoztunk, a verem egyetlen tokent tartalmaz: az AST reprezentációját az egész programnak.

A `pyparsert` [17] használva a keletkezett AST-t visszaalakítjuk C programozási nyelven írt kóddá, melyet ezután meg lehet mutatni a felhasználónak.

4. fejezet

Kiértékelés

A következő szakaszban bemutatjuk a kiértékelési módszereinket és a modellünk eredményét: a teljesítményét a 10 bonyolultsági szinten, a különböző nyelvi elemeken, valamint az egymásba ágyazott bináris kifejezések nélkül, amik a legnehezebben visszafejthető nyelvi elemek. A szakaszt azzal zárjuk, hogy bemutatjuk a modell teljesítményét a valós adathalmazon.

4.1. Módszerek a teljesítmény mérésére

Több mód is van arra, hogy a kódvisszafejtés sikerességét mérni tudjuk. Ebben a dolgozatban két fő kiértékelési módszert fogunk alkalmazni annak eldöntésére, hogy milyen jól teljesít a modellünk.

Az első a tokenenkénti pontosság. Ez a technika azt méri, hogy hány token egyezik meg az elvárt RPN kimenetben és a modell prediktált kimenetében, osztva az elvárt kimenet hosszával. Egyezésnek csak azt tekintjük, ha ugyanazon a pozíción szerepel a két token. Tehát egy megegyező, de eltoltszakasza a kimeneti tokeneknek nem tekinthető egyezésnek. A tokenenkénti pontosság képlete alább látható:

$$\frac{\sum_{i=1}^{|y|} \text{ind}(\mathbf{y}_{(i)} = \hat{\mathbf{y}}_{(i)})}{|y|}$$

A második módszer a tökéletesen visszafejtetéseket méri, ahol az elvárt és a tényleges kimenet tokenjei megegyeznek minden indexen, és a hosszuk is azonos.

4.2. Teljesítmény a bonyolultsági szinteken

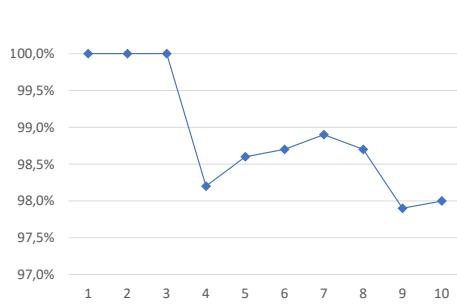
A kódvisszafejtés problémáját 10 bonyolultsági szintre osztottuk. Ezek a szintek hasonlóak ahhoz, amiket a TraFix [7] szerzői bevezettek, de van egy pár kisebb eltérés, valamint mi két új célkitűzést is hozzáadtunk (9. és 10. szint).

1. Számliterálok értékül adása
2. 1. szint + Változók értékül adása
3. 2. szint + Unáris operátorok ($++$, $--$, $\sim$), mint önálló utasítások és értékadásokon belül
4. 2. szint + Értékadások, amiknek a jobb oldala tartalmazhat bináris kifejezéseket ($+$, $-$, $*$, $/$, $\%$, $|$, $\&$, $^$, $<<$, $>>$, ahol az első 5 operátor egymásba ágyazva is szerepelhet)
5. A 3. és a 4. szint együttese
6. 5. szint + If és If-else elágazások (egymásba ágyazhatóság nélkül) és ternáris operátorok (egymásba ágyazhatóság nélkül).
7. 6. szint + While ciklusok (egymásba ágyazhatóság nélkül)
8. 7. szint + Egymásba ágyazhatóság
9. 8. szint + Switch utasítások (egymásba ágyazhatóság nélkül, és más utasításoknak sem lehetnek részei)
10. 9. szint + Függvények paraméterekkel, visszatérési értékkel és függvényhívások

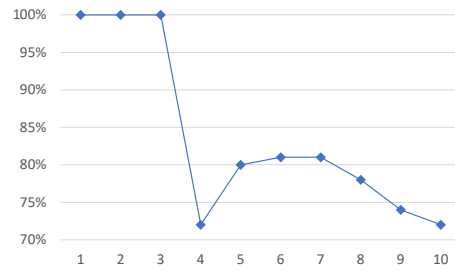
A különbség az első 8 szint meghatározása és a TraFix [7] által definiált szintek között a következő: a 6. szinten, valamint afelett lehetnek ternáris operátorok is.

A generált adathalmazunk tartalmaz kódokat minden bonyolultsági szintről. A szintenkénti példaszám és az egyes nyelvi elemek száma a tanítási adathalmazunkban a Függelékben látható (A.1. és A.2. Táblázat). Összességében az adathalmazunk 260161 tanítási példát tartalmaz. Az egyes bonyolultsági szinteket reprezentáló példák mellett az adathalmazban szerepelnek olyan példák is, amik főként egy adott nyelvi elemet használnak, vagy valamilyen speciális kódot tartalmaznak, amit a GCC különleges módon optimalizál. Ez utóbbiból 161 darab található az adathalmazban.

A modell által elért tokenenkénti pontosság és a tökéletesen visszafejtett kódok aránya a különböző szinteken az 4.1. ábrán láthatók.



(a) A pontosság a 10 bonyolultsági szinten. Az y tengely nem 0-tól kezdődik.



(b) A tökéletes kódvisszafejtés mértéke a 10 bonyolultsági szinten. Az y tengely nem 0-tól kezdődik.

4.1. ábra. A modellünk pontossága a bonyolultsági szinteken. A teljesítményt a tokenenkénti pontossággal és a tökéletes kódvisszafejtéssel ábrázoljuk.

4.3. Teljesítmény az egyes nyelvi elemeken

A modellünk pontosságát megmértük olyan adathalmazokon is, amik csak egy-egy nyelvi elemre koncentrálnak. Ezek az adathalmazok az adott típusú nyelvi elemen kívül a többi sokkal kisebb mértékben tartalmazzák (csak értékadásokat literálokkal vagy változókkal).

Ezek az eredmények a 4.1. táblázatban láthatók: BINARY egymásba ágyazható két-operandusú operátorokat, BITWISE értékadásokat bitműveletekkel, IF egymásba ágyazható elágazásokat, TERNARY értékadásokat ternáris operátorokkal, WHILE egymásba ágyazható while ciklusokat, SWITCH egymásba nem ágyazható switch utasításokat és FUNCTION értékadásokat tartalmaz függvényhívásokkal, valamint függvénydefiníciókat paraméterekkel és visszatérési értékkel.

4.1. táblázat. A modellünk teljesítménye olyan adathalmazokon, amik csak egyes nyelvi elemeket tartalmaznak.

Szint	Tokenenkénti pontosság	Tökéletes kódvisszafejtés
BINARY	97.45%	62%
BITWISE	100.00%	100%
IF	99.08%	90%
TERNARY	99.04%	86%
WHILE	99.70%	97%
SWITCH	96.34%	75%
FUNCTION	99.95%	98%

4.4. Teljesítmény egymásba ágyazott bináris operátorok nélkül

A modellünk teljesítményét olyan adatokon is megmértük, ami nem tartalmaz egymásba ágyazott bináris operátorokat. Ezen adathalmaz elkészítésének oka az, hogy az előzetes méréseink alapján ez a nyelvi elem bizonyult a legnehezebbnek a modell számára. Az elért eredmények a 4.2. táblázatban olvashatók.

4.2. táblázat. A modellünk eredménye az egymásba ágyazott bináris operátorokat nem tartalmazó adathalmazon.

Szint	Tokenenkénti pontosság	Tökéletes kódvisszafejtés
4	99.81%	99%
5	99.76%	97%
6	99.82%	98%
7	99.81%	98%
8	99.56%	96%
9	98.82%	91%
10	98.88%	88%

4.5. Eredmények valós adathalmazokon

A kód példák a realisztikus adathalmazunkban Codeforces-ról¹ lettek összegyűjtve, egy weboldalról, ami online programozási versenyeket szervez. 2000 különböző feladatból 10582 darab beadott, C-ben írt fájlt gyűjtöttük össze. Minden feladatnál a 10 legrövidebb, helyes megoldást töltöttük le (amennyiben nem volt 10 ilyen, akkor annyit, amennyi volt).

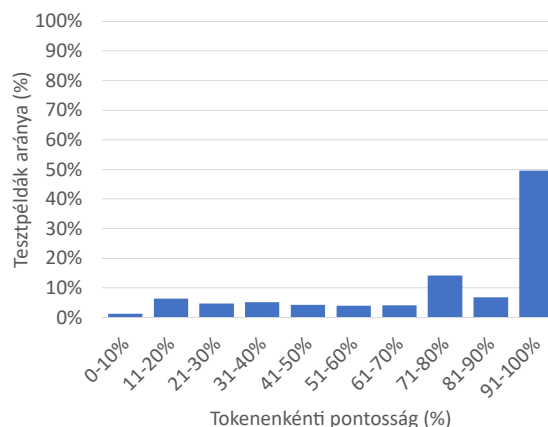
Mielőtt még hozzáadtuk volna az adathalmazhoz őket, a beadott fájlokat megszűrtük. Minden fájl, ami a következő szabályok bármelyikét teljesítette, elvetettük:

- tartalmaz olyan operátorokat, amiket nem láttunk tanítás közben (pl. +=, &&, &=)
- tartalmaz nem tanított beépített típusokat (pl. struct, enum, float, tömbök)
- tartalmaz nem tanított beépített kulcsszavakat (pl. continue, goto, exit)
- tartalmaz nem tanított beépített függvényeket (pl. strcpy, abs)

¹<http://codeforces.com/> 2022

- használ olyan könyvtárakat, amik nem tanított beépített függvényeket tartalmaznak (pl. math.h, string.h, stdlib.h)
- egy másik beadott megoldás duplikációja
- üres fájl
- tartalmaz scanf függvényhívást egy ciklus feltételében (ennek az az oka, hogy a többi scanf hívást lecseréljük értékadásokra, ahogy azt később el is magyarázzuk)
- tartalmaz értékadásokat ternáris operátorokon belül vagy láncolt értékadásokat
- tartalmaz több függvénydefiníciót is vagy globális változót
- tartalmaz void kulcsszót

A szűrés után megmaradt fájlok még mindig tartalmazhattak olyan ciklusokat, amiket tanítás közben nem ismert meg a modell (for ciklus, do-while ciklus). A for ciklusok esetében ekvivalensen át tudjuk őket alakítani while ciklusokká, amiket tanult a modell. Ezzel szemben a do-while ciklusok legalább egyszer lefuttatják a ciklus belsejét. Ez a probléma úgy lett megoldva, hogy a ciklus belsejének egy másolatát helyeztük a ciklus elé, és a ciklust egy egyszerű while ciklussá alakítottuk. Fontos megjegyezni, hogy ez nem ugyanazt az assembly kódot generálja, mintha közvetlenül egy do-while ciklust fordítanánk le GCC-vel. Ezek mellett a változók és a függvények nevét is megváltoztattuk, hogy konzisztensek legyenek a tanítópéldákkal.



4.2. ábra. Az elért pontosságok eloszlása a realisztikus adathalmazon.

Végül az átalakított fájlok szintén átestek azokon a teszteken, mint amiken a tanító-példák (ld. a Módszerekben szereplő szűrésről szóló szakaszt). A fájlok, amik sikeresen átmentek ezen a szűrőn hozzá lettek adva az adathalmazhoz. A végleges adathalmaz 669 darab példát tartalmaz.

Habár a modellünk generált adatokon volt tanítva, ami a különböző nyelvi elemeket (pl. while ciklusok, ternáris operátorok) sokkal nagyobb gyakorisággal tartalmazta, mint az emberek által írt kód, így is jól teljesít a realisztikus adathalmazon. 76%-os tokenenkénti pontosságot és 35%-os tökéletes kódvisszafejtést ért el. A visszafejtés pontosságának eloszlása a 4.2. ábrán látható.

5. fejezet

Diszkusszió

A modellünk felülmúlta azokat az eredményeket, amiket az alapozó cikk [6], a TraFix [7], és a Coda [8] elértek (Coda esetében csak az ott mutatott hibajavító technikájuk alkalmazása nélkül). A tökéletes visszafordítási arányban 8%-ot javítottunk a TraFix-hoz képest a számukra legnehezebb 8. komplexitási szinten (4.1 Ábra), ezzel cáfolva a TraFix szerzőinek korábbi kijelentését, miszerint a feladat nem megoldható utólagos hibajavítások nélkül. Emellett eredményesen növeltük a sikeresen visszafordított kód maximális hosszát, mely esetünkben 808 token hosszú volt az előfeldolgozást megelőzően, ez 21%-os növekedést jelent a TraFix korábbi eredményével szemben, az alapozó cikkhez képest pedig 621%-os javulást jelent (az alapozó cikk leghosszabb kódja 112 tokenből állt). Habár a Coda tökéletes visszafordítási arányban 90% felett teljesített, az ő példáikban a függvényhívások domináltak, így a számliterálok (és az ezeken értelmezett operátorok), valamint a ciklusok elenyésző mértékben jelentek meg a példáikban. Hasonló példákön arhitektúránk hasonlóan jó visszarodítási arányt ért el utólagos iteratív hibajavítások nélkül, ez látható a 4.1 táblázat IF, FUNCTION soraiban.

Az eredményeink azt mutatják, hogy a modell számára a visszafejtés során a legnehezebbek az egymásba ágyazott bináris operátorok, amik szorzást, osztást vagy maradékszámítást írnak le egy számliterállal a jobb oldalukon, valamint a switch utasítások (4.1 táblázat). A bináris kifejezések azért okoznak problémát, mert gyakran bitenkénti eltolásokra konvertálódnak fordítás során, még a GCC -O0-s beállításának használatával is, és a keletkező hosszú kódban nehéz számontartani a szabályokat. A switch utasítások pedig azért nehezek, mert viszonylag magas az absztrakciós szintjük, és a fordításkor keletkező kód hossza nagyobb, mint a többi nyelvi elem esetében.

A 4.1 ábrán látható, hogy a 4. szint nehéz a modell számára, mivel ebben a szintben

jelennek meg a bináris operátorok. Úgy gondoljuk, hogy a modell azért teljesít jobban a későbbi szinteken, mert ott kisebb a gyakorisága az egymásba ágyazott bináris operátoroknak, mivel több nyelvi elem típus van a példákban.

6. fejezet

Összefoglalás

Ebben a dolgozatban bemutattuk az end-to-end, single-pass kódvisszafejtő eszközünket, amely másoló mechanizmust alkalmaz. Megmutattuk, hogy egy neurális kódvisszafejtő iteratív hibajavító algoritmusok használata nélkül is képes megtanulni helyesen prediktálni a literálokat és a változókat.

A modellünk nagy pontosságot ért el a különböző kódvisszafejtési feladatokon, miközben idiomatikus kódot generált anélkül, hogy imitálta volna a regisztereket vagy goto utasításokat használt volna. A bonyolultsági szintrendszerünk a TraFix [7] hasonló szintrendszerén alapult, de kiegészítettük két új szinttel, hogy újabb nyelvi elemeket is megtanuljon a modell (ternáris operátorok, switch utasítások, függvények), amiket a programozók nap mint nap használnak.

Azt is megmutattuk, hogy egy end-to-end, single-pass modell képes hatékonyan visszafejtetni alacsony absztrakciós szintű kódot anélkül, hogy iteratív hibajavítási módszereket alkalmazna. A másoló mechanizmus hozzáadása a modellünkhöz lehetővé tette, hogy helyesen prediktálja a literálokat és a változókat, ami nehéz feladat a tudományterületen.

Kutatásunk közben azt tapasztaltuk, hogy az egymásba ágyazott bináris kifejezések visszafejtése a legnehezebb feladat. Ezek a kifejezések olyan köztes értékeket használnak fel az alacsony szintű számításokban, amik lényegesen meghosszabbítják a kódot.

Úgy gondoljuk, hogy a jövőben érdemes lenne megnézni, hogy egy tree LSTM [19] copy-val hogyan kezelné a kódvisszafejtés problémáját. Ahelyett, hogy lineáris szöveggént tárolná az adatokat, ezek a hálók egy faszerkezetet alkalmaznak a program leírásához, ami sokkal egyszerűbbé teszi a nyelvi elemek közti kapcsolatok felismerését.

Jelenleg készül egy nemzetközi folyóiratba szánt cikk a munkánkról.

Köszönetnyilvánítás

Köszönjük Csertán András és Buzga Levente segítségét a program implementációjában, valamint témavezetőink útmutatását.

Az Innovációs és Technológiai Minisztérium ÚNKP-21-1 kódszámú Új Nemzeti Kiválóság Programjának a Nemzeti Kutatási, Fejlesztési és Innovációs Alapból finanszírozott szakmai támogatásával készült.



NEMZETI KUTATÁSI, FEJLESZTÉSI
ÉS INNOVÁCIÓS HIVATAL

6.1. ábra. ÚNKP és NKFIH logója

Irodalomjegyzék

- [1] Zhibo Liu és Shuai Wang. “How Far We Have Come: Testing Decompilation Correctness of C Decompilers”. *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2020. Virtual Event, USA: Association for Computing Machinery, 2020, 475–487. ISBN: 9781450380089. DOI: 10.1145/3395363.3397370. URL: <https://doi.org/10.1145/3395363.3397370>.
- [2] 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>.
- [3] 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>.
- [4] Sepp Hochreiter és Jürgen Schmidhuber. “Long short-term memory”. *Neural computation* 9.8 (1997), 1735–1780. old.
- [5] Jiajun Zhang és Chengqing Zong. “Deep Neural Networks in Machine Translation: An Overview”. *IEEE Intelligent Systems* 30.5 (2015), 16–25. old. DOI: 10.1109/MIS.2015.69.
- [6] 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.
- [7] Omer Katz és tsai. “Towards Neural Decompilation”. *CoRR* abs/1905.08325 (2019). arXiv: 1905.08325. URL: <http://arxiv.org/abs/1905.08325>.

- [8] 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>.
- [9] 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>.
- [10] 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>.
- [11] Ruigang Liang és tsai. “Neutron: an attention-based neural decompiler”. *Cybersecurity* 4.1 (2021), 1–13. old. DOI: 10.1186/s42400-021-00070-0.
- [12] Yan Shoshitaishvili és tsai. “SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis”. *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, 138–157. old. DOI: 10.1109/SP.2016.17.
- [13] 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>.
- [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] H. S. Warren. “Hacker’s delight”. *Pearson Education* (2013).
- [16] GNU Project. *GCC 7.2.0*. <https://github.com/gcc-mirror/gcc/tree/releases/gcc-7>. 2019.
- [17] Eli Bendersky. *pycparser*. <https://github.com/eliben/pycparser>. 2021.
- [18] 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>.
- [19] Kai Sheng Tai, Richard Socher és Christopher D. Manning. “Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks”. *CoRR* abs/1503.00075 (2015). arXiv: 1503.00075. URL: <http://arxiv.org/abs/1503.00075>.

A. függelék

Mellékletek

A.1. táblázat. Az egyes C nyelvi elemek száma a generált adathalmazban.

Típus	Darabszám
Értékkadás	2 211 424
Literál	2 149 270
Változó	4 744 408
Unáris operátor	311 530
Bináris operátor	748 692
Bitoperátor	247 126
Logikai kifejezés	252 116
Ternáris operátor	69 442
If elágazás	93 194
While ciklus	89 480
Switch utasítás	39 510
Case blokkok	117 587
Függvényhívások	66 203
Függvénydefiníciók	38 990

A.2. táblázat. Az egyes nyelvi elemek generáláshoz használt eloszlási súlyai az 1-10. szinteken.

Szint	1	2	3	4	5	6	7	8	9	10
Értékadás	5	5	5	5	5	5	5	5	5	5
Inkrementálás / dekrementálás	0	0	1	0	1	1	1	1	1	1
If elágazás	0	0	0	0	0	1	1	1	1	1
While ciklus	0	0	0	0	0	0	1	1	1	1
Switch utasítás	0	0	0	0	0	0	0	0	1	1
Literál	2	2	2	2	2	2	2	2	2	2
Változó	0	2	2	2	2	2	2	2	2	2
Unáris operátor	0	0	2	0	2	2	2	2	2	2
Bináris operátor	0	0	0	8	8	8	8	8	8	8
Bitoperátor	0	0	0	2	4	4	4	4	4	4
Ternáris operátor	0	0	0	0	0	1	1	1	1	1
Függvényhívás	0	0	0	0	0	0	0	0	0	1
Utasítások számának minimума	4	4	4	3	3	3	3	3	3	3
Utasítások számának maximuma	9	9	8	7	7	6	6	6	6	6
Generált adatok száma (* 1000)	5	5	5	50	15	5	5	5	5	100

A.3. táblázat. A tokenenkénti pontosság és a tökéletes visszafordítás aránya az egyes szinteken.

Szint	Pontosság	Tökéletes
1	100%	100%
2	100%	100%
3	100%	100%
4	98.2%	72%
5	98.6%	80%
6	98.7%	81%
7	98.9%	81%
8	98.7%	78%
9	97.9%	74%
10	98.0%	72%