TDK-dolgozat

Mucsányi Bálint Gyarmathy Bálint Czapp Ádám Tibor

FlexCoder: Gyakorlati programszintézis flexibilis inputhosszokkal és kifejező lambdafüggvényekkel

EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

PROGRAMOZÁSELMÉLET ÉS SZOFTVERTECHNOLÓGIAI TANSZÉK

Szerzők:

Mucsányi Bálint	Gyarmathy Bálint	Czapp Ádám Tibor
programtervező informatikus BSc	programtervező informatikus MSc	programtervező informatikus MSc
3. évfolyam	1. évfolyam	1. évfolyam

Témavezetők:

dr. Pintér Balázs	dr. Gregorics Tibor
egyetemi adjunktus	egyetemi docens

Budapest, 2020

Lezárás dátuma: 2020. 12. 16.

Tartalomjegyzék

1. fejezet

Bevezetés

Dolgozatunkban egy flexibilis programszintézis-modellt mutatunk be, melynek feladata egy olyan függvénykompozíció szintetizálása, amely a megadott bemeneti példákat a hozzájuk tartozó kimeneti példákra transzformálja. A bemeneti példák rendszerünkben minden esetben listák, a kimeneti példák lehetnek egyaránt listák és skalárértékek is.

A programszintézisnek két fő ágát különböztethetjük meg [1]. Deduktív programszintézis esetén egy igazolhatóan helyes program generálását tűzzük ki célként, amely megfelel egy formális specfikációnak. Induktív programszintézis esetén a szintetizálandó program elvárt működését csak példákkal demonstráljuk, vagy egy szöveges leírással [2].

A Példaalapú Programozás (Programming by Examples, PbE) a programszintézis egy demonstratív megközelítése, mellyel dolgozatunk is foglalkozik. A példák PbE esetén rendezett párok, melyek bemenetekből és az azokra elvárt kimenetekből állnak [3]. Ezekből állítjuk elő a programot (1.1 ábra) úgy, hogy az a bemenetekből előállítsa a kimeneteket (1.2 ábra). A PbE rendszereket leggyakrabban string-transzformációkra [4, 5, 6] és listamanipulációkra [7, 8, 9] alkalmazzák.

A mély tanulás egyre növekvő népszerűségével a programszintézis kutatási területe nagy áttöréseket ért el a modellek pontosságában és sebességében egyaránt [7]. A gépi tanulási algoritmusok heurisztikaként történő felhasználása az egyik legprominensebb megközelítése az ezen módszerek integrálásának a szintetizálási folyamatba. Ezek a heurisztikák például a feladat állapotgráfján alkalmazható gráfkereső algoritmusokban használhatók fel. Érdekes megközelítés a népszerű, jól ismert heurisztikák gépi tanulási módszerekkel

1.1. ábra. Az ábrán egy PbE feladat látható. Bemenetként egy párt kapunk: az elvárt bemenet(ek)et és az elvárt kimenet(ek)et. A szintetizáló rendszer feladata, hogy megadja azt a programot (jelen esetben függvénykompozíciót), amely az összes megadott bemenetre a hozzák tartozó megadott kimenetet állítja elő. A példában az egyszerűség kedvéért egyetlen bemenetet (az $[1,5,4,-2,6]$ listát) és egyetlen kimenetet (a $[2,10,8]$ listát) adtunk meg. Ennek a specifikációnak eleget tesz a $map(*2,take(3,array))$ függvénykompozíció.

1.2. ábra. Az ábrán a 1.1 ábrán megadott $map(*2,take(3,array))$ függvénykompozíciót láthatjuk. Az ezt alkotó függvényeket a megfelelő sorrendben (először a *take*, majd a *map* függvényt) végrehajtva az $[1,5,4,-2,6]$ listán valóban a $[2,10,8]$ listát kapjuk eredményül, így a szintetizálási folyamat sikeres volt.

történő kiegészítése is. Az ilyen módszerek predikciói tehát jól ki is tudják egészíteni a feladat sajátosságait kihasználó ismereteket [6, 5]. Erre a 2. fejezetben mutatunk példát.

A kutatási terület alapvető publikációja a DeepCoder [7] cikk, amely számos újabb cikk alapjául szolgált [8, 6, 9, 5]. Példaként szolgál a PCCoder [8] rendszer, mely nagy előrelépést mutatott a szintézis teljesítményében.

Az ezen munkák által bemutatott jelentős továbbfejlesztések ellenére még nem láthattunk sok példát a terület legjobb rendszereinek valós környezetben való alkalmazására. Ennek egyik oka a meglévő rendszerek limitációi, például a programot alkotó függvények lehetséges paramétereinek szűkössége, vagy a bemenetek hosszára tett szigorú megszorítások.

Dolgozatunkban ezen rendszerek limitációit szeretnénk csökkenteni és flexibilitásukat növelni, hogy így egy lépést tehessünk a valóélet-beli feladatok felé. Rendszerünket ezen megfontolásból *FlexCoder*-nek neveztük el.

A dolgozatunkban közölt eredményeink egy cikk alapját is képzik, melyet teljes cikként elfogadtak a 2021-es International Conference on Pattern Recognition Applications and Methods (ICPRAM) konferenciára [10].

2. fejezet

Kapcsolódó munkák

A DeepCoder [7] a neurális programszintézis szakterületének alapvetése és számos későbbi publikáció alapja. Rendszerük neurális hálója bemeneti-kimeneti párok alapján prediktálja, hogy mely függvények vannak jelen a szintetizálandó programban, amely segít a keresési algoritmus irányításában. Hálójukat azonban nem használják a keresés minden lépésében, csak a folyamat kezdetén.

A PCCoder rendszer [8] nagy előrelépést mutatott a szintézis teljesítményében a DeepCoder által definiált szakterület-specifikus nyelven (DSL). A PCCoder lépésenkénti keresést alkalmaz, amely az egyes lépésekben az aktuális állapotot használja a program következő függvényének prediktálására, beleértve mind a függvényekhez tartozó operátorokat és azok paramétereit. A keresésre a Complete Anytime Beam Search (CAB) [11] keresési algoritmust használják, és a futásidőt két nagyságrenddel csökkentik a DeepCoderhez képest, miközben kiemelkedően jobb eredményeket érnek el ugyanazokon az adathalmazokon.

Feng és tsai. rendszere [9] jó példa a függvénykompozíciók sikeres felhasználására a programok szintetizálásához. Ez a konfliktusvezérelt tanuláson alapuló módszer képes lemmákat tanulni, amik segítségével a keresendő programteret fokozatosan csökkenti. Ezen újítások bevezetésével jobb eredményeket képes elérni, mint a DeepCoder reimplementációjuk. (A DeepCoder rendszerhez nem került kiadásra hivatalos forráskód.)

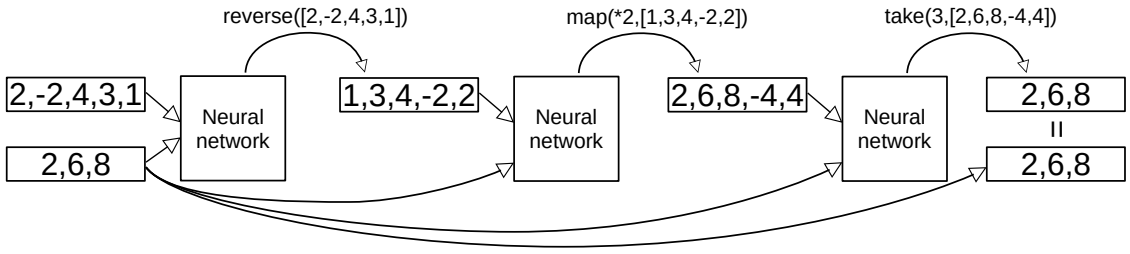
Kalyan és tsai. munkája [6] bemeneti-kimeneti példákon mutat be egy neurális háló által támogatott deduktív keresést, kombinálva a heurisztikákat és a neurális háló predikcióit

a szintetizálási folyamatban. Rangfüggvényük ugyanazt a szerepet tölti be, mint rendszerünkben a neurális háló. A jelentős különbség a FlexCoder-hez képes az, hogy ezt a rendszert szövegátalakításokra használják.

3. fejezet

Módszer

Módszerünk a neurális háló által irányított programszintézis, amely egy optimalizált nyalábszélességekkel dolgozó nyalábkeresés algoritmusra támaszkodik. A nyalábkeresés minden egyes lépését a neurális háló irányítja. A függvénykompozíciók DSL-je a funkcionális programozásból ismert, gyakran magasabbrendű függvényekből épül fel. A FlexCoder-nek két fő egysége van: a *nyalábkeresés algoritmus* és a *neurális háló*. Azonban van egy harmadik fontos egység is, a *nyelvtan*, ami definiálja a DSL-t.



3.1. ábra. A programszintézis folyamata. A feladat a bemeneti listá(ka)t a kimeneti listá(k)ra transzformáló függvénykompozíció megadása. Ebben az esetben a bemenet a $[2,-2,4,3,1]$ lista, a kimenet pedig a $[2,6,8]$ lista. Az ábra bemutatja, hogy a becsült függvényt hogyan hajtjuk végre a bemeneten, majd hogyan tápláljuk vissza az eredményt a neurális hálóba. A reverse függvény megfordítja a listában található elemek sorrendjét. A map függvény az adott operátorból és numerikus paraméterből (a példában '$*$', illetve '2') álló függvényt hajtja végre a bemeneti lista összes elemén. A take függvény a bemenete első, paramétereként megkapott számú elemét (a példában '3') adja vissza egy új listaként. Amennyiben ezen függvényeket a megfelelő sorrendben hajtjuk végre a bemeneti listán, láthatjuk, hogy a kurrens bemeneti lista megegyezik az elvárt kimeneti listával, amely azt jelenti, hogy egy megoldást találtunk: $take(3, map(*2, reverse(input)))$.

A 3.1 ábra a rendszer működésébe ad betekintést. A neurális hálónak minden egyes lépésben átadunk egy bemeneti-kimeneti párt, ami ezek után a kompozícióban soron következő

függvényt prediktálja. Ezt a függvényt ezután alkalmazzuk az aktuális bemeneti-kimeneti párra, ezáltal létrehozzuk az algoritmus következő iterációjának a bemenetét. Ezt addig folytatjuk, amíg meg nem találjuk a megoldást, vagy el nem érjük az iterációs korlátot.

3.1. Példagenerálás és nyelvtan

A függvénykompozíciókat környezetfüggetlen nyelvtan (CFG) segítségével reprezentáljuk [12]. Ez a letisztult és egyértelmű megadási mód lehetővé teszi a nyelvtan könnyű bővíthetőségét, legyen szó akár új függvényekről, akár meglévő függvények paramétereiről. A környezetfüggetlen nyelvtant a Natural Language Toolkit [13] segítségével implementáljuk. A nyelvtan teljes szerkezetét (a függvények rövid leírásával) a 3.2 ábra tartalmazza.

függvény	map	filter	take	drop	search
operátorok	+, -, /, *, %	<, >, ==, %	-	-	-
paraméterek	{-8, -7, . . . , 8}	{-8, -7, . . . , 8}	{1, 2, . . . , 8}	{1, 2, . . . , 8}	{-8, -7, . . . , 8}
kosár	map	filter	takedrop	takedrop	search

3.1. táblázat. A függvények az operátoraikkal, a paramétereik intervallumával és a függvénykompozíciók felépítése során használt súlyozott véletlen kiválasztáshoz alkalmazott kosarakkal.

függvény	**min**	**max**	**reverse**	**sort**	**sum**	**count**
kosár	one_param	one_param	one_param	one_param	one_param	one_param

3.2. táblázat. A paraméterek és operátorok nélküli függvények a megfelelő kosarakkal.

Amennyiben nem adunk felső korlátot a programkompozíciók hosszára, a lehetséges programok terének számossága megszámlálhatóan végtelen. Adathalmazunk generálása során csak megoldható példákat hozunk létre, így az optimális, azaz minimális számú függvényből álló megoldás létezése garantált.

A számparamétereket a $\{-8,-7,\ldots,8\}$ halmazból választjuk, amely jóval nagyobb, mint a PCCoder által használt $\{-1,0,\ldots,4\}$ halmaz. Az átmeneti, köztes listák elemei a $\{-1024,-1023,\ldots,1024\}$ halmazból kerülnek ki (ez a PCCodernél $\{-256,-255,\ldots,256\}$), míg a bemeneti listák esetében ez a $\{-256,-255,\ldots,256\}$ halmaz (amely megegyezik a PCCoder által használttal).

A lambdafüggvényeket a visszatérési értékük típusa alapján két kategóriába sorolhatjuk: boolean lambdafüggvények, melyeket a filter használ és numerikus lambdafüggvények,

$$
\begin{aligned}
S &\to ARRAY\_FUNCTION \\
S &\to NUMERIC\_FUNCTION \\
ARRAY\_FUNCTION &\to sort(ARRAY) \\
ARRAY\_FUNCTION &\to take(POS, ARRAY) \\
ARRAY\_FUNCTION &\to drop(POS, ARRAY) \\
ARRAY\_FUNCTION &\to reverse(ARRAY) \\
ARRAY\_FUNCTION &\to map(NUM\_LAMBDA, ARRAY) \\
ARRAY\_FUNCTION &\to filter(BOOL\_LAMBDA, ARRAY) \\
NUMERIC\_FUNCTION &\to max(ARRAY) \mid min(ARRAY) \\
NUMERIC\_FUNCTION &\to sum(ARRAY) \mid count(ARRAY) \\
NUMERIC\_FUNCTION &\to search(NUM, ARRAY) \\
NUM &\to NEG \mid 0 \mid POS \\
NEG &\to -8 \mid -7 \mid ... \mid -2 \mid -1 \\
POS &\to 1 \mid GREATER\_THAN\_ONE \\
GREATER\_THAN\_ONE &\to 2 \mid 3 \mid ... \mid 8 \\
BOOL\_LAMBDA &\to BOOL\_OPERATOR\ NUM \\
BOOL\_LAMBDA &\to MOD \\
BOOL\_OPERATOR &\to == \mid < \mid > \\
MOD &\to \%\ GREATER\_THAN\_ONE == 0 \\
NUM\_LAMBDA &\to *\ MUL\_NUM \\
NUM\_LAMBDA &\to /\ DIV\_NUM \\
NUM\_LAMBDA &\to +\ POS \\
NUM\_LAMBDA &\to -\ POS \mid \%\ GREATER\_THAN\_ONE \\
MUL\_NUM &\to NEG \mid 0 \mid GREATER\_THAN\_ONE \\
DIV\_NUM &\to NEG \mid GREATER\_THAN\_ONE \\
ARRAY &\to list \mid ARRAY\_FUNCTION
\end{aligned}
$$

3.2. ábra. A függvénykompozíciók generálására használt nyelvtan. Mindenekelőtt fontos megjegyezni, hogy a függvények nem módosítják a bemeneti listát, ahelyett új listát adnak vissza. A sort függvény az ARRAY elemeit rendezi növekvő sorrendbe. A take megtartja, míg a drop eldobja az ARRAY első POS darabnyi elemét. A reverse függvény megfordítja a bemeneteként kapott lista elemeit. A map és a filter is magasabbrendű függvények. A map alkalmazza a NUM_LAMBDA lambdafüggvényt a paraméteréül kapott lista minden egyes elemére. A filter csak azokat az elemeket tartja meg az ARRAY listából, amelyekre a BOOL_LAMBDA predikátum igazat ad vissza. A min és max függvények a legkisebb és legnagyobb elemét adják vissza az ARRAY-nek. A sum függvény az ARRAY elemeinek összegét, míg a count az elemeinek számát adja vissza. A search annak az ARRAY listaelemnek az indexét adja vissza, amely egyenlő a NUM paraméterrel. A generált példák közül csak azokat fogadjuk el, amelyeknél a NUM valóban eleme az ARRAY-nek.

melyeket a map kaphat paraméteréül. A lambdafüggvényekhez meghatározunk néhány szabályt, hogy elkerüljük a hibás kifejezéseket, vagy az identitásfüggvényeket. Hibás kifejezés például a nullával való osztás; identitásfüggvények például a $(+0)$ és a $(*1)$ függvények.

A nyelvtant a lehetséges függvények generálására használjuk, melyek a kompozícióink építőelemei. Mivel a függvényeknek jelentősen eltérő számú lehetséges paramétere van, ezért biztosítanunk kell, hogy az összes lehetséges függvénytípus hasonló számban szerepel az adathalmazban. Ez azért fontos, mert a neurális hálónknak mindegyik, a nyelvtanban található függvényről megfelelő tudást kell gyűjtenie a tanulási fázis során, és a példák kiegyensúlyozása nélkül lehetőség nyílna arra, hogy bizonyos függvényekre túl kevés példát lásson a háló. Ennek érdekében a kompozíciókat felépítő függvényeket súlyozott véletlen kiválasztással adjuk meg a lehetséges 151 felparaméterezett függvényből.

A függvényeket öt kategóriába soroljuk az alapján, hogy a nyelvtanban hány különböző lehetséges felparaméterezése szerepel. Ahogy a 3.1 és 3.2 táblázatok mutatják, a különböző kategóriákba tartozó függvényeknek változó számú lehetséges argumentumai vannak, és az intervallumok is eltérőek lehetnek, melyekből a számparamétereket választjuk. Mivel minden kategóriából egyenlő valószínűséggel szeretnénk választani, ezért először súlyozott véletlen kiválasztással egy kategóriát választunk, az egyes kategóriáknak a 3.1 egyenlet által meghatározott súlyozását figyelembe véve. Ezután a kategóriában szereplő függvények közül azonos valószínűséggel választunk egyet.

$$weight(function) = \begin{cases} 6/11, & \text{ha } function \in one\_param \\ 2/11, & \text{ha } function \in takedrop \\ 1/11, & \text{ha } function \in map \cup filter \cup search \end{cases} \tag{3.1}$$

Az így generált függvényeken különböző szűrési technikákat alkalmazunk, hogy elkerüljük a redundáns függvényeket vagy szuboptimális felparaméterezéseket.

Az első szűrési technika, ami a függvények paramétereit optimalizálja:

- map(+ 1, map(+ 2, [1, 2])) $\rightarrow$ map(+ 3, [1, 2])
- filter($>$ 2, [1, 6, 7]) $\rightarrow$ filter($>$ 5, [1, 6, 7])

A második szűrési technika az adott bemeneti-kimeneti párra identitásként ható függvényeket távolítja el:

- sum(filter($>$ 5, [6, 7, 8])) $\rightarrow$ sum([6, 7, 8])

A nyelvtan alapvetően nem engedélyezi azokat a függvényeket, amelyek minden bemeneti-kimeneti párra identitásként hatnának (például a *map(+ 0, [2, 3, 5]))* függvény).

A harmadik szűrő azon kompozíciókat távolítja el, amelyek üres listát eredményeznének:

- filter($<$ 1, [2, 3, 4]) $\rightarrow$ [ ]
- drop(4, [1, 2, 3]) $\rightarrow$ [ ]

A negyedik szűrő eltávolít minden olyan példát, amelyik tartalmaz a $\{-1024, -1023, \ldots, 1024\}$ halmazon kívül eső értékeket:

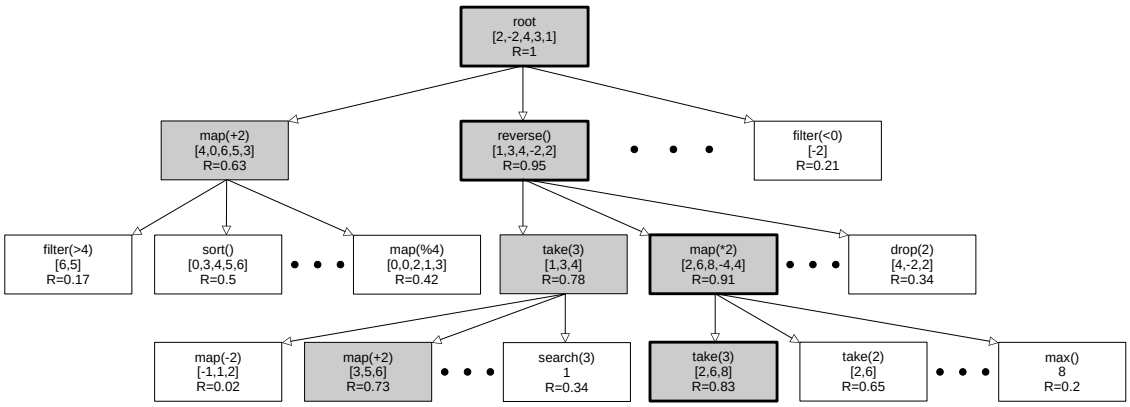
- map($*$ 8, [1, 2, 255]) $\rightarrow$ [8, 16, 2040]

3.2. Nyalábkeresés (Beam Search)

Ha a programtér felsorolásával keressük a megoldást, egy, a kompozíciók hosszával exponenciálisan skálázódó problémával állunk szemben. A kezdeti programszintézisrendszerek tételbizonyító algoritmusokat és körültekintően megválasztott heurisztikákat alkalmaztak a keresési tér redukálása céljából [14]. Rendszerünkben egy mély neuronhálót használunk a keresés irányításához a heurisztikák helyett.

A korábban említett módon a programszintézisre optimális függvénykompozíció kereséseként tekintünk. A kompozíciókat függvényről-függvényre építjük fel. A nyalábkeresést a Complete Anytime Beam Search (CAB) [11] algoritmus alapján implementáltuk.

Az algoritmus csúcs objektumok irányított fáját építi fel. Minden egyes csúcs három mezőt tartalmaz: egy függvényt, ennek a függvénynek a szülő csúcsának kimenetére történő alkalmazásának az eredményét és a csúcs rangját. A fa minden szintjén a csúcsokat a rangjuk szerint csökkenő sorrendbe rendezzük. Ezt a rangot a felparaméterezett függvények rangjaként értelmezzük, melyet a neurális háló predikciói alapján állapítunk meg.



3.3. ábra. Az ábrán egy sikeres nyalábkeresés látható, amely során azon függvénykompozíciók valamelyikét keressük amely a $[2,-2,4,3,1]$ bemeneti listából a $[2,6,8]$ kimenetet állítja elő. Az irányított keresési fa minden csúcsa három mezőt tartalmaz: egy függvényt, ennek a függvénynek a szülő csúcs kimenetére való alkalmazásának eredményét, és a csúcs rangját. A szürke négyzetek azok a csúcsok, melyeket az algoritmus működése során kiterjesztünk. Ezeket a rangjuk alapján választjuk ki, amelyet R-el jelölünk. A fekete kerettel kiemelt csúcsok az eredményt mutatják: $take(3, map(*2, reverse(input)))$.

A fa minden egyes csúcsának függvényét (azaz a szintetizálandó kompozícióban soron következő lehetséges függvényeket) a hálónak a csúcs szülőjében tárolt aktuális bemeneti-kimeneti listákra vett predikciója határozza meg. Abban az esetben, ha több bemeneti-kimeneti példa is van, ezeket kötegelve adjuk át a hálónak, majd a predikciókat átlagoljuk.

Miután az összes gyerek csúcsot legeneráljuk, megtartjuk az első $\nu_i \in \mathbb{N}$ $(i \in 1..\vartheta)$ darabot (ahol ν_i az aktuális szinten lévő nyalábszélességet és ϑ a mélységi korlátot jelöli) a csúcsok rendezett listájából, majd a kiértékelésükkel megadjuk ezek eredénymezőjét is. Ezt mindaddig ismételjük, amíg nem találunk megoldást, vagy az algoritmus el nem éri az iterációs limitet (vagy opcionálisan az időlimitet). Amennyiben megoldást találunk, elég csak a szülő pointereket követnünk a helyes kompozíció meghatározásához. Amikor elérjük a ϑ mélységi limitet, de nem találtunk megoldást, megduplázzuk az összes ν_i értéket és újrakezdjük a keresést. Mivel a neurális hálót egy keresés során többször is meghívjuk, a folyamat gyorsítása érdekében cache-elést használunk, hogy eltároljuk az egyedi bemeneti-kimeneti párok rangját. Ez azért lehetséges mert a neurális háló egy tiszta függvény.

Az Algoritmus 1 minden egyes mélységéhez előre optimalizált nyalábszélességeket használ. Ezeket a ciklus minden egyes iterációjában megduplázzuk. Az első lépésben a 3.1 alfejezetben említett módon a kiértékelt kimenetekre, vagy ezek hosszára tett megszorítá-

1. Algoritmus Nyalábkeresés.

Funct nyalábkeresés()

```
 1: i = 0
 2: found = false
 3: while i < max_iter and not found do
 4:    depth = 0
 5:    nodes = [root]
 6:    while depth < ϑ and not found do
 7:       beam_size = v[depth] * 2^i
 8:       output_nodes = process(filter(nodes))
 9:       found = check_solution(output_nodes)
10:       nodes = take_best(beam_size, output_nodes)        rang hozzárendelése
11:       depth += 1
12:    end while
13:    i += 1
14: end while
```

sokat ki nem elégítő programokat távolítjuk el. A kimenetek esetében a hosszra definiált megszorítás biztosítja számunkra, hogy csak azokat a csúcsokat tartjuk meg, melyek kimeneti listájában legalább annyi elem van, mint az eredeti kimeneti listában. A szűrést követően a neurális háló segítségével minden egyes hátramaradó programhoz hozzárendeljük a megfelelő rangot.

A programokat végre is hajtjuk, hogy a megoldásra tett megszorításokat is leellenőrizhessük. Ha megoldást találunk, az algoritmus véget ér. Ellenkező esetben az első *nyalábméret*nyi (beam size) legnagyobb ranggal rendelkező kompozíciót kiterjesztjük, és a belső ciklus következő iterációjában egy új függvénnyel kibővítjük. Ha a belső ciklusból kilépünk, a nyalábszélességeket megduplázzuk, de a már korábban meghatározott rangokat nem számoljuk ki újra a feljebb említett cache-elésnek köszönhetően.

A nyalábszélességeket a validációs adathalmazon végzett tesztfuttatások alapján optimalizáltuk úgy, hogy minden egyes szinten azt az átlagos nyalábszélességet (nyalábszélesség $\in \{v_1, v_2, ..., v_n\}$) határoztuk meg, ami tartalmazza a kompozíció következő függvényét. Ezáltal nagyobb esélyünk van arra, hogy már az első, vagy korai iterációkban találjunk megoldást. Minden egyes szinten azokat az átlagos nyalábszélességeket választottuk, ami 90%-ban tartalmazta az eredeti megoldásban szereplő aktuális függvényt. Ez a 90%-os határ megfelelő arányt biztosít a pontosság és sebesség között.

3.3. Az alkalmazott rekurrens neurális hálók

Mivel neurális hálónk architektúrájában központi szerepet kapnak a rekurrens rétegek, ismertetjük az általunk kipróbált változatoknak fontosabb tulajdonságait és főbb különbségeit.

3.3.1. Egyszerű rekurrens neurális háló

Az általunk használt rekurrens hálók ismertetéséhez érdemes ezek architektúrái taglalása előtt létrehozásuk motivációját bemutatni.

Legyen T a feldolgozott sorozat hossza, $\mathbf{h} \in \mathbb{R}^s$ a rejtett reprezentáció, $\mathbf{W_x} \in \mathbb{R}^{s \times n}$ a bemenet súlymátrixa, $\mathbf{W_h} \in \mathbb{R}^{s \times s}$ és $\mathbf{b_h} \in \mathbb{R}^s$ a rejtett reprezentáció súlymátrixa és eltolásvektora, $\mathbf{W_y} \in \mathbb{R}^{k \times s}$ és $\mathbf{b_y} \in \mathbb{R}^k$ a kimenet súlymátrixa és eltolásvektora. Jelöljük továbbá σ_h-val a rejtett reprezentáció[1], σ_y-val pedig a predikció aktivációs függvényét, $\mathbf{x} \in \mathbb{R}^n$-val a bemeneti listát, $\hat{\mathbf{y}}$-val pedig a predikciót.

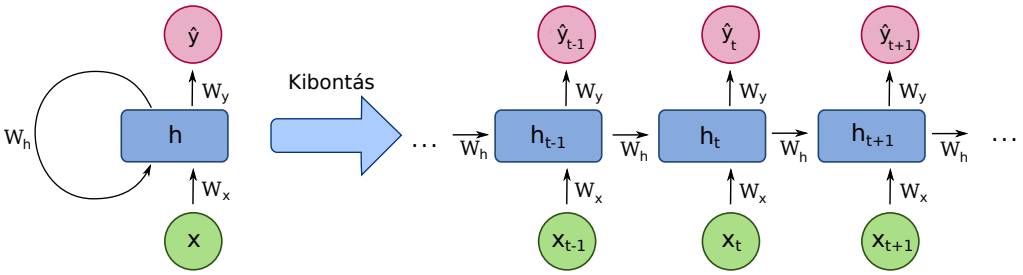
Ekkor a legegyszerűbb visszacsatolt neurális háló az alábbi egyenletekkel írható le (*Elman-féle architektúra*):

$$\begin{aligned} & \mathbf{h_0} = \mathbf{0} \\ \forall t \in \{1, 2, \ldots, T\} : & \\ & \mathbf{h_t} = \sigma_h(\mathbf{W_h h_{t-1}} + \mathbf{W_x x_t} + \mathbf{b_h}) \\ & \hat{\mathbf{y}}_\mathbf{t} = \sigma_y(\mathbf{W_y h_t} + \mathbf{b_y}), \end{aligned} \tag{3.2}$$

ahol a t alsó index az egyes tenzorok t. időlépésben felvett értékét jelöli.

Minden rekurrens neurális háló egy állapotleírást tanul, amit az előző egyenletekben a $\mathbf{h}$ vektor reprezentál. Ezt az RNN a bemeneteként kapott sorozat feldolgozása közben az addig látott, általa fontosnak ítélt információk tárolására használja. Minden lépésben a sorozat következő elemét vizsgálja, és minden elem feldolgozásához ugyanazokat a súlyokat és eltolásvektorokat használja ($\mathbf{W_x}, \mathbf{W_h}, \mathbf{W_y}, \mathbf{b_y}, \mathbf{b_h}$). A hálóban összesen 5 paramétertenzor található: 3 súlymátrix és 2 eltolásvektor. Az architektúra kompakt, illetve időben kiterített formában a 3.4 ábrán látható.

[1] A rejtett reprezentáció aktivációs függvényét leggyakrabban a hiperbolikus tangens (tanh) függvénynek választják meg.



3.4. ábra. Az ábrán az Elman-féle egyszerű rekurrens neurális háló architektúra látható, bal oldalon kompakt, jobb oldalon pedig időben kiterített formában. Forrás: Wikimedia Commons[2]

Az architektúrát leggyakrabban a gradiensmódszer alkalmazásával tanítjuk. Legegyszerűbb formájában ezt az alábbi algoritmussal írhatjuk le:

2. Algoritmus Gradiensmódszer.

Funct gradiensmódszer()

1: A modell költségének kiszámítása az adathalmazon.
2: **for** $i \leftarrow 1 \ldots n$ **do**
3: $\quad \theta_i = \theta_i$ - $\alpha * \frac{\partial}{\partial \theta_i} J(\theta_i)$
4: **end for**

Az algoritmusban θ-val jelöltük a modell összes tanítandó paraméterét, J-vel a modell költségét, α-val pedig a tanulási rátát. Az algoritmusban felhasznált parciális deriváltakat a gradiens-visszaterjesztés módszerével határozhatjuk meg, amely a láncszabály sorozatos alkalmazásából áll (*Backpropagation Through Time*). Ennek szemléltetéséhez válasszuk meg σ_h-t a hiperbolikus tangens függvénynek:

$$\forall i \in \{1,2,\ldots,s\} : \sigma_h(\mathbf{x})_i = tanh(\mathbf{x})_i = \frac{e^{x_i} - e^{-x_i}}{e^{x_i} + e^{-x_i}}, \tag{3.3}$$

illetve legyen σ_y a softmax függvény:

$$\forall i \in \{1,2,\ldots,k\} : \sigma_y(\mathbf{x})_i = softmax(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}}. \tag{3.4}$$

[2]Módosított változat. Szerző: fdeloche. `https://commons.wikimedia.org/wiki/File:Recurrent_neural_network_unfold.svg`

Legyen továbbá a kereszt-entrópia a választott költségfüggvényünk:

$$\mathrm{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{i=1}^{k} y_i log(\hat{y}_i), \tag{3.5}$$

ahol $\mathbf{y} \in \mathbb{R}^k$ egy one-hot vektor, azaz egyetlen nemnulla eleme van, melynek értéke 1, azon az indexen, amely a klasszifikációs problémában a megfelelő osztályt jelöli.

A deriváltak egyszerűbb meghatározásának érdekében bővítsük a 3.2 egyenleteket köztes állapotokkal, illetve a költségek kiszámolásával:

$$\begin{aligned}
&\mathbf{h_0} = \mathbf{0} \\
\forall t \in \{1,2,\ldots,T\}: & \\
&\mathbf{z_t} = \mathbf{W_h h_{t-1}} + \mathbf{W_x x_t} + \mathbf{b_h} \\
&\mathbf{h_t} = \tanh(\mathbf{z_t}) \\
&\mathbf{g_t} = \mathbf{W_y h_t} + \mathbf{b_y} \\
&\hat{\mathbf{y}}_\mathbf{t} = \mathrm{softmax}(\mathbf{g_t}) \\
&J_t = \mathrm{CE}(\hat{\mathbf{y}}_\mathbf{t}, \mathbf{y_t}).
\end{aligned} \tag{3.6}$$

Egy példa teljes költsége az egyes időpillanatokban számolt költségek összege:

$$J = \sum_{t=1}^{T} J_t. \tag{3.7}$$

Ebből következik, hogy

$$\begin{aligned}
\frac{\partial J}{\partial \mathbf{W_y}} &= \sum_{t=1}^{T} \frac{\partial J_t}{\partial \mathbf{W_y}} \\
\frac{\partial J}{\partial \mathbf{W_h}} &= \sum_{t=1}^{T} \frac{\partial J_t}{\partial \mathbf{W_h}} \\
\frac{\partial J}{\partial \mathbf{W_x}} &= \sum_{t=1}^{T} \frac{\partial J_t}{\partial \mathbf{W_x}}.
\end{aligned} \tag{3.8}$$

Először határozzuk meg $\frac{\partial J_t}{\partial \mathbf{W_y}}$ ($t \in \{1,2,\ldots,T\}$) láncszabállyal történő felbontását. A $\mathbf{W_y}$

súly csak a t. időlépésben befolyásolja J_t értékét, így a felbontás egyszerűen megadható:

$$\frac{\partial J_t}{\partial \mathbf{W_y}} = \frac{\partial J_t}{\partial \hat{\mathbf{y}}_\mathbf{t}} \frac{\partial \hat{\mathbf{y}}_\mathbf{t}}{\partial \mathbf{g_t}} \frac{\partial \mathbf{g_t}}{\partial \mathbf{W_y}} \tag{3.9}$$

Folytassuk a levezetést $\frac{\partial J_t}{\partial \mathbf{W_h}}$ ($t \in \{1,2,\ldots,T\}$) felbontásának megadásával. Itt már egy bonyolultabb kifejezést kapunk eredményül, ugyanis a $\mathbf{W_h}$ súly a t. és az azt megelőző időlépésekben is befolyásolja J_t értékét:

$$\frac{\partial J_t}{\partial \mathbf{W_h}} = \sum_{i=0}^{t} \frac{\partial J_t}{\partial \hat{\mathbf{y}}_\mathbf{t}} \frac{\partial \hat{\mathbf{y}}_\mathbf{t}}{\partial \mathbf{g_t}} \frac{\partial \mathbf{g_t}}{\partial \mathbf{h_t}} \frac{\partial \mathbf{h_t}}{\partial \mathbf{h_i}} \frac{\partial \mathbf{h_i}}{\partial \mathbf{z_i}} \frac{\partial \mathbf{z_i}}{\partial \mathbf{W_h}}. \tag{3.10}$$

Végül $\frac{\partial J_t}{\partial \mathbf{W_x}}$ ($t \in \{1,2,\ldots,T\}$) felbontását adjuk meg. Hasonló kifejezést kapunk, mint $\frac{\partial J_t}{\partial \mathbf{W_y}}$ esetén, ugyanis $\mathbf{W_x}$ szintén befolyásolja J_t értékét a t. időlépés előtti lépésekben is:

$$\frac{\partial J_t}{\partial \mathbf{W_x}} = \sum_{i=0}^{t} \frac{\partial J_t}{\partial \hat{\mathbf{y}}_\mathbf{t}} \frac{\partial \hat{\mathbf{y}}_\mathbf{t}}{\partial \mathbf{g_t}} \frac{\partial \mathbf{g_t}}{\partial \mathbf{h_t}} \frac{\partial \mathbf{h_t}}{\partial \mathbf{h_i}} \frac{\partial \mathbf{h_i}}{\partial \mathbf{z_i}} \frac{\partial \mathbf{z_i}}{\partial \mathbf{W_x}}. \tag{3.11}$$

A láncszabályokban szereplő parciális derivált-értékek megadása következik. Kezdjünk $\frac{\partial J_t}{\partial \mathbf{g_t}} = \frac{\partial J_t}{\partial \hat{\mathbf{y}}_\mathbf{t}} \frac{\partial \hat{\mathbf{y}}_\mathbf{t}}{\partial \mathbf{g_t}}$-vel, melyhez először megadjuk a softmax függvény deriváltját:

$$\frac{\partial \text{softmax}(\mathbf{x})_i}{\partial x_j} = \frac{\partial \frac{e^{x_i}}{\sum_{s=1}^{k} e^{x_s}}}{\partial x_j} \quad (i,j \in \{1,2,\ldots,k\}). \tag{3.12}$$

$i = j$ esetén:

$$\begin{aligned} &\frac{\partial \frac{e^{x_i}}{\sum_{s=1}^{k} e^{x_s}}}{\partial x_j} = \frac{e^{x_i}\sum_{s=1}^{k} e^{x_s} - e^{x_i}e^{x_j}}{(\sum_{s=1}^{k} e^{x_s})^2} = \\ &\frac{e^{x_i}(\sum_{s=1}^{k} e^{x_s} - e^{x_j})}{(\sum_{s=1}^{k} e^{x_s})^2} = \text{softmax}(\mathbf{x})_i(1 - \text{softmax}(\mathbf{x})_j) = \\ &\text{softmax}(\mathbf{x})_j(1 - \text{softmax}(\mathbf{x})_j). \end{aligned} \tag{3.13}$$

$i \neq j$ esetén:

$$\frac{\partial \frac{e^{x_i}}{\sum_{s=1}^{k} e^{x_s}}}{\partial x_j} = \frac{0 * \sum_{s=1}^{k} e^{x_s} - e^{x_i} e^{x_j}}{(\sum_{s=1}^{k} e^{x_s})^2} = \\ - \frac{e^{x_i}}{\sum_{s=1}^{k} e^{x_s}} \frac{e^{x_j}}{\sum_{s=1}^{k} e^{x_s}} = -\text{softmax}(\mathbf{x})_i \text{softmax}(\mathbf{x})_j. \tag{3.14}$$

A Kronecker delta használatával egyben is felírhatjuk a két esetet:

$$\frac{\partial \frac{e^{x_i}}{\sum_{s=1}^{k} e^{x_s}}}{\partial x_j} = \text{softmax}(\mathbf{x})_i (\delta_{ij} - \text{softmax}(\mathbf{x})_j). \tag{3.15}$$

Ekkor tehát

$$\forall i \in \{1, 2, \ldots, k\}:$$

$$\begin{aligned} \frac{\partial J_t}{\partial g_{t_i}} &= -\sum_{s=1}^{k} y_{t_s} \frac{\partial log(\hat{y}_{t_s})}{\partial g_{t_i}} = \\ &- \sum_{s=1}^{k} y_{t_s} \frac{\partial log(\hat{y}_{t_s})}{\partial \hat{y}_{t_s}} \frac{\partial \hat{y}_{t_s}}{\partial g_{t_i}} = -\sum_{s=1}^{k} \frac{y_{t_s}}{\hat{y}_{t_s}} \frac{\partial \hat{y}_{t_s}}{\partial g_{t_i}} = \\ &- \sum_{s=1}^{k} \frac{y_{t_s}}{\hat{y}_{t_s}} \frac{\partial \hat{y}_{t_s}}{\partial g_{t_i}} = -\sum_{s=1}^{k} \frac{y_{t_s}}{\hat{y}_{t_s}} \hat{y}_{t_s} (\delta_{si} - \hat{y}_{t_i}) = \\ &- \sum_{s=1}^{k} y_{t_s} (\delta_{si} - \hat{y}_{t_i}) = \sum_{s=1, s \neq i}^{k} y_{t_s} \hat{y}_{t_i} - y_{t_i}(1 - \hat{y}_{t_i}) = \\ &\sum_{s=1, s \neq i}^{k} y_{t_s} \hat{y}_{t_i} - y_{t_i} + y_{t_i} \hat{y}_{t_i} = \sum_{s=1}^{k} y_{t_s} \hat{y}_{t_i} - y_{t_i} = \hat{y}_{t_i} - y_{t_i}, \end{aligned} \tag{3.16}$$

ugyanis az előbbiek alapján $\mathbf{y}$-nak egyetlen nemnulla eleme van.

Egyszerűbben:

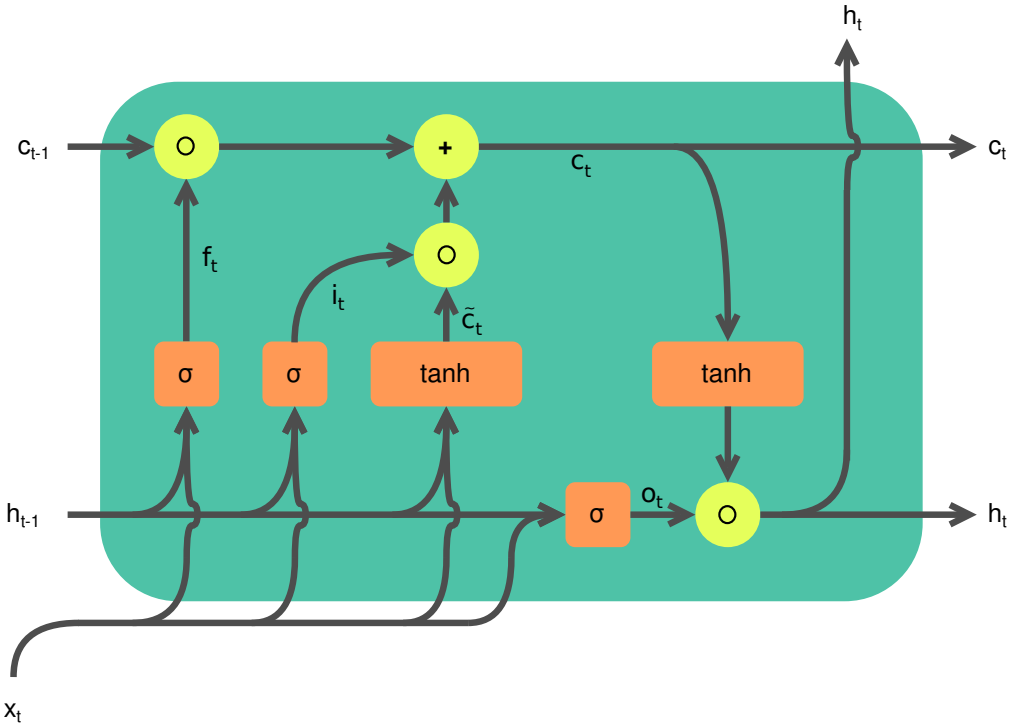
$$\frac{\partial J_t}{\partial \mathbf{g_t}} = \hat{\mathbf{y}}_\mathbf{t} - \mathbf{y_t}. \tag{3.17}$$

A maradék derivált megadása már egyszerű feladat:

$$\begin{aligned}
\frac{\partial J_t}{\mathbf{W_y}} &= (\hat{\mathbf{y}}_\mathbf{t} - \mathbf{y_t})\mathbf{h_t} \\
\frac{\partial J_t}{\mathbf{h_t}} &= (\hat{\mathbf{y}}_\mathbf{t} - \mathbf{y_t})\mathbf{W_y} \\
\frac{\partial \mathbf{h_t}}{\partial \mathbf{z_t}} &= 1 - \tanh^2(\mathbf{z_t}) = 1 - \mathbf{h_t}^2 \\
\frac{\partial \mathbf{z_t}}{\partial \mathbf{h_{t-1}}} &= \mathbf{W_h} \\
\frac{\partial \mathbf{z_t}}{\partial \mathbf{W_x}} &= \mathbf{x_t} \\
\frac{\partial \mathbf{z_t}}{\partial \mathbf{W_h}} &= \mathbf{h_{t-1}}.
\end{aligned} \tag{3.18}$$

Érdemes megfigyelni a $\frac{\partial \mathbf{h_t}}{\partial \mathbf{h_{t-1}}} = \frac{\partial \mathbf{h_t}}{\partial \mathbf{z_t}} \frac{\partial \mathbf{z_t}}{\partial \mathbf{h_{t-1}}} = (1 - \mathbf{h_t}^2)\mathbf{W_h}$ parciális deriváltat: $\mathbf{W_h}$ a bemeneti sorozat feldolgozása során minden lépésben ugyanaz (*weight sharing*), és a gradiensek visszaterjesztésekor $(1 - \mathbf{h_t}^2)\mathbf{W_h}$-val minden visszafele tett lépésben beszorzunk, ahogy bejárjuk a számítási fát a láncszabály sorozatos alkalmazásával. Ekkor egy, a mély tanulás tudományágában jól ismert problémával találkozhatunk: az *instabil gradiensek* jelenségével. Amennyiben T értéke meglehetősen nagy, azaz egy hosszú szekvenciát kell feldolgoznia a neurális hálónak, sokszor szorzunk be $(1 - \mathbf{h_t}^2)\mathbf{W_h}$ -val. Ha W_h Frobenius-normája 1-nél kisebb, akkor a kezdeti időlépésekhez egy várhatóan nagyon kicsi gradiensérték érkezik, azaz a tanulás lehetetlenné válik. Könnyebben megoldható probléma, ha a mátrix Frobenius-normája nagyobb 1-nél: ekkor a kezdeti időlépésekhez egy várhatóan nagyon nagy gradiensérték érkezik, azonban a gradienskorlátozás (*gradient clipping*) módszerével a visszaterjesztés minden lépésében megadhatunk egy felső korlátot a gradiensnek: $grad := min\{grad, max\_grad\}$, így a tanulás továbbra is kivitelezhető marad.

Mivel egy számítógépen véges aritmetikával dolgozunk, így ezen két esetben $\exists T \in \mathbb{N}^+$, melytől kezdve vagy túlcsordulás, vagy alulcsordulás figyelhető meg a probléma tanítása során. Mindkét esetben megakad a tanulás folyamata. A nagyobb problémát általában a gradiensek eltűnése jelenti (*vanishing gradients*), hiszen a túlcsordulásra láthattunk megoldást, így foglalkozzunk ennek elkerülésével.



3.5. ábra. Az ábrán az LSTM architektúra működésének egyszerűsített változata látható. A jelölések a 3.3.2 alfejezetben találhatóak. Az ábra tetején levő $\mathbf{h_t}$ értéket $\hat{\mathbf{y}}_\mathbf{t}$ kiszámításához használjuk fel. Forrás: Wikimedia Commons[3]

3.3.2. Long Short-Term Memory

Az architektúra Hochreiter és Schmidhuber nevéhez fűződik, melyet 1997-ben mutattak be. A sejt működésének egyszerűsített folyamata a 3.5 ábrán látható.

Az eltűnő gradiensek problémáját nagy mértékben csillapítja az LSTM architektúra, melyben a rejtett reprezentáció mellett egy sejtreprezentációt is alkalmazunk. Ez egy olyan útvonalat is biztosít a gradiensnek, melyen változatlanul haladhat tovább, így nagyságrendje nem változik. Megjelenik továbbá 3 gát is (felejtő gát, aktualizáló gát, kimeneti gát), melyek az információk elfelejtését, illetve megjegyzését egyensúlyozzák. Ezen tulajdonságok miatt az LSTM sejt egy jóval kedvezőbb választás, amennyiben hosszabb sorozatokat szeretnénk feldolgozni, azonban általánosan is jobb teljesítményt nyújt az egyszerű rekurrens sejtnél a gátak által bevezetett információszelekció miatt.

Az előző jelöléseket kiegészítve legyen $\mathbf{f} \in \mathbb{R}^s$ az úgynevezett felejtő gát aktivációs vektora, $\mathbf{i}, \mathbf{o} \in \mathbb{R}^s$ az aktualizáló gát és a kimeneti gát aktivációs vektorai, $\tilde{\mathbf{c}} \in \mathbb{R}^s$ a sejtbe-

[3]Módosított változat. Szerző: Guillaume Chevalier. `https://commons.wikimedia.org/wiki/File:LSTM_cell.svg`

menet aktivációs vektora, $\mathbf{c} \in \mathbb{R}^s$ pedig a sejt bemeneti vektora. A tanítható paraméterek legyenek $\mathbf{W_f}, \mathbf{W_i}, \mathbf{W_o}, \mathbf{W_c} \in \mathbb{R}^{s \times n}$, $\mathbf{U_f}, \mathbf{U_i}, \mathbf{U_o}, \mathbf{U_c} \in \mathbb{R}^{s \times s}$, és $\mathbf{b_f}, \mathbf{b_i}, \mathbf{b_o}, \mathbf{b_c} \in \mathbb{R}^s$. Jelölje σ_g az összes gát aktivációs függvényét, σ_c a sejtreprezentáció kiszámításánál használt aktivációs függvényt, illetve σ_h a rejtett reprezentáció aktivációs függvényét. Az eredeti publikációban [15] σ_g-t a sigmoid függvénynek, σ_c-t és σ_h-t pedig a hiperbolikus tangens függvénynek választották meg.

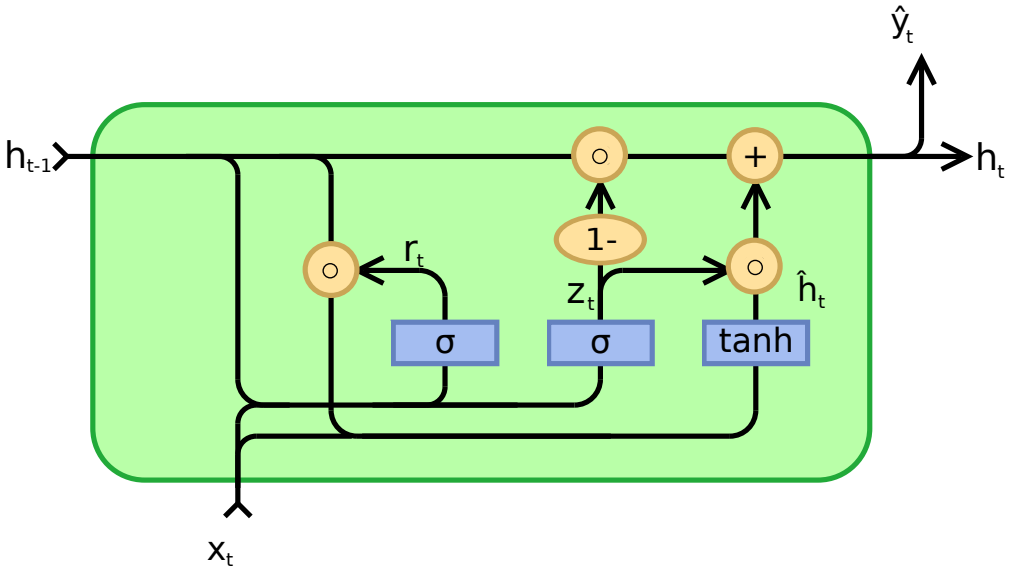
Az LSTM architektúra működését leíró egyenletek:

$$
\begin{aligned}
&\mathbf{h_0} = \mathbf{0} \\
&\mathbf{c_0} = \mathbf{0} \\
\forall t \in \{1, 2, \ldots, T\}: & \\
&\mathbf{f_t} = \sigma_g(\mathbf{W_f x_t} + \mathbf{U_f h_{t-1}} + \mathbf{b_f}) \\
&\mathbf{i_t} = \sigma_g(\mathbf{W_i x_t} + \mathbf{U_i h_{t-1}} + \mathbf{b_i}) \\
&\mathbf{o_t} = \sigma_g(\mathbf{W_o x_t} + \mathbf{U_o h_{t-1}} + \mathbf{b_o}) \\
&\mathbf{\tilde{c}_t} = \sigma_c(\mathbf{W_c x_t} + \mathbf{U_c h_{t-1}} + \mathbf{b_c}) \\
&\mathbf{c_t} = \mathbf{f_t} \circ \mathbf{c_{t-1}} + \mathbf{i_t} \circ \mathbf{\tilde{c}_t} \\
&\mathbf{h_t} = \mathbf{o_t} \circ \sigma_h(\mathbf{c_t}),
\end{aligned}
\tag{3.19}
$$

ahol $\circ$ a Hadamard-szorzatot jelöli.

3.3.3. Gated Recurrent Unit

A GRU sejtek az LSTM sejtek után 17 évvel, 2014-ben jelentek meg. Az architektúra működésének folyamata a 3.6 ábrán látható. Ezek az LSTM sejtekkel ellentétben csak két gátat tartalmaznak (aktualizáló gát, visszaállító gát), illetve nem tartanak számon egy külön sejtállapotot, így számítási igényük jóval kisebb. Ez azonban nem minden feladat esetén jár a teljesítmény romlásával az LSTM-hez képest: olyan eseteket is láthatunk, ahol azonos dimenzionalitással a GRU sejtek jobb teljesítményt érnek el, mint az LSTM sejtek. Ez azonban nem olyan gyakori: a legtöbb esetben azonos dimenzionalitás esetén valamivel gyengébben teljesít a GRU, viszont propagációs és tanítási ideje jóval kedvezőbb. Emiatt a két architektúrát a mi feladatunk esetén úgy érdemes összehasonlítani, hogy a GRU dimenzionalitását növeljük, ugyanis egy, a mi feladatunk szempontjából igazságos össze-



3.6. ábra. Az ábrán a GRU sejtet leíró egyenletek vizuális megjelenítése látható. A jelölések a 3.3.3 alfejezetben találhatóak. Forrás: Wikimedia Commons[4]

hasonlítás a tanítási és propagációs időt közel egyenlővé teszi a két architektúra esetén. Ilyen feltételek mellett már gyakran tapasztalhatjuk azt, hogy a GRU sejt jobban teljesít az LSTM-nél.

A GRU sejtek a gradiensek visszaterjesztése során az LSTM sejtekhez hasonlóan szintén képesek a gradienseknek egy olyan útvonal nyújtására is, melyen keresztül nem változik a nagyságrendjük még hosszabb szekvencia esetén sem.

Megintcsak az egyszerű rekurrens hálónál ismertetett jelöléseket egészítjük ki: legyen $\hat{\mathbf{h}} \in \mathbb{R}^s$ a rejtettreprezentáció-jelölt, $\mathbf{z} \in \mathbb{R}^s$ az aktualizáló gát vektora, $\mathbf{r} \in \mathbb{R}^s$ a visszaállító gát vektora, a tanítható paraméterek pedig: $\mathbf{W_x}, \mathbf{W_r}, \mathbf{W_h} \in \mathbb{R}^{s \times n}$, $\mathbf{U_z}, \mathbf{U_r}, \mathbf{U_h} \in \mathbb{R}^{s \times s}$, illetve $\mathbf{b_z}, \mathbf{b_r}, \mathbf{b_h} \in \mathbb{R}^s$. Jelöljük továbbá egységesen σ_g-vel az egyes gátak aktivációs függvényeit, σ_h-val pedig a rejtett reprezentáció kiszámításához használt aktivációs függvényt. Az architektúrát először bemutató publikációban [16] σ_g-t a sigmoid függvénynek, σ_h-t pedig a hiperbolikus tangens függvénynek adták meg.

[4]Módosított változat. Szerző: Jeblad. `https://commons.wikimedia.org/wiki/File:Gated_Recurrent_Unit,_base_type.svg`

Ezen jelölésekkel a GRU sejt működése az alábbi egyenletekkel írható le:

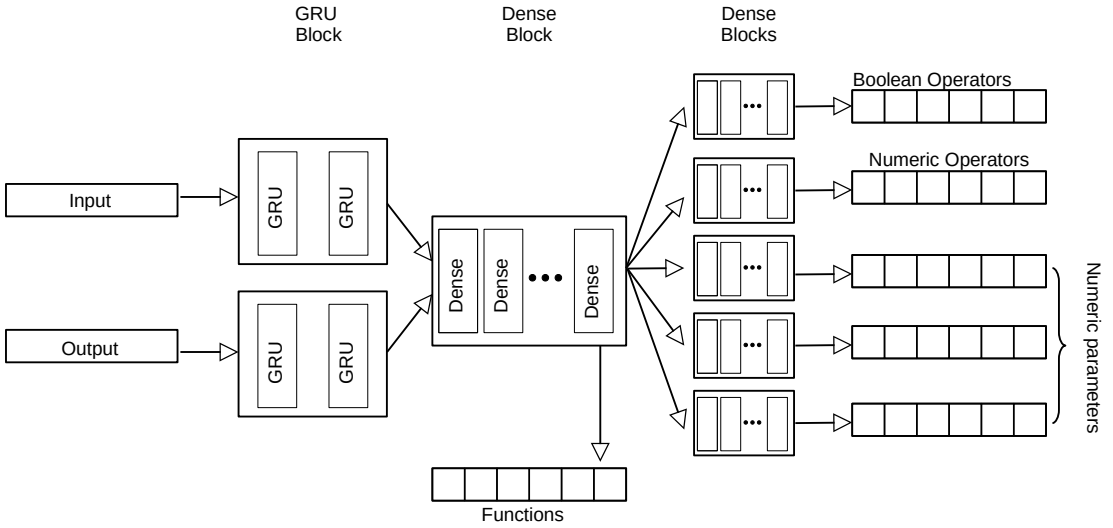
$$
\begin{aligned}
&\mathbf{h_0} = \mathbf{0} \\
&\forall t \in \{1, 2, \ldots, T\} : \\
&\mathbf{z_t} = \sigma_g(\mathbf{W_z x_t} + \mathbf{U_z h_{t-1}} + \mathbf{b_z}) \\
&\mathbf{r_t} = \sigma_g(\mathbf{W_r x_t} + \mathbf{U_r h_{t-1}} + \mathbf{b_r}) \\
&\hat{\mathbf{h}}_\mathbf{t} = \sigma_h(\mathbf{W_h x_t} + \mathbf{U_h}(\mathbf{r_t} \circ \mathbf{h_{t-1}}) + \mathbf{b_h}) \\
&\mathbf{h_t} = (\mathbf{1} - \mathbf{z_t}) \circ \mathbf{h_{t-1}} + \mathbf{z_t} \circ \hat{\mathbf{h}}_\mathbf{t}.
\end{aligned}
\tag{3.20}
$$

3.3.4. Kétirányú rekurrens hálók

Gyakori feladat a természetes nyelvfeldolgozás területén, hogy egy adott szó kontextusából következtetünk egy tulajdonságára. Például a *Named Entity Recognition* feladatban olyan nevesített entitásokat ismerünk fel, mint a személyek, cégek, helyek. Legtöbbször ahhoz, hogy egy rendszer megfelelő biztonsággal el tudja dönteni egy szóról, hogy az egy tulajdonnév-e, a szó kontextusát is alaposan meg kell vizsgálnia. Egy szó kontextusába azonban a vizsgált szó utáni szavak is beletartoznak, ezeket pedig az eddig taglalt modellek nem láthatják, ugyanis ezek csak a jelenlegi szó és az addigi szavak alapján következtethetnek. A kétirányú rekurrens rétegek erre adnak megoldást: a szöveget két oldalról egyszerre dolgozzák fel, és minden szónál az azt megelőző és azt követő szavak mindegyikéből ki tudnak nyerni hasznos információt. Ez implementáció szintjén azt jelenti, hogy a két irányból érkező rejtett reprezentációkat minden időlépésben konkatenáljuk, majd a fentebb látható egyenletekből már ismert további feldolgozó lépések után végül ebből a konkatenált mátrixból kapjuk meg a háló kimenetét. Ezt az architektúratípust is kipróbáltuk a neurális hálónkkal történő kísérletezés során, mind az LSTM, mind a GRU sejttel.

3.4. A neurális háló

A neurális háló bemenete egyetlen program bemeneti és kimeneti példáinak listája. Kimenete 6 vektor, melyek az egyes függvények, paraméterek és lambdafüggvények rangjait tartalmazzák.



3.7. ábra. A neurális háló architektúrája. A háló összes bemeneti példáját először a GRU blokk kapja meg, amely egy belső reprezentációt ad vissza. Ez a reprezentáció egy teljesen összekötött rétegekből álló blokknak kerül továbbításra. Ezután hat részre osztjuk a modell további komponenseit, melyek közül öt további, teljesen összekötött rétegekből álló blokkokkal folytatódik.

A felparaméterezett függvény rangját a komponensei rangjának mértani közepeként határozzuk meg.

A függvények halmazát F-fel jelöljük, melynek összes eleme egy pár (f_{class}, f_{arg}), ahol f_{class} a függvény neve, f_{arg} pedig a függvény argumentumainak listája. A függvény rangját az alábbi képlettel határozzuk meg:

$$R(f) = \sqrt[n+1]{R(f_{class}) * \prod_{i=1}^{n} R(f_{arg_i})}, \tag{3.21}$$

ahol $n \in N$ a paraméterek száma.

3.4.1. Architektúra

A bemeneti és kimeneti listák külön-külön haladnak át két rekurrens rétegekből álló blokkon, melyek lehetővé teszik a változó hosszúságú listák feldolgozását. Ezen blokkok két rétegnyi GRU [16] sejtet tartalmaznak, melyek rejtett reprezentáció mind 256 neuronból állnak.

A bemeneti és kimeneti listák GRU reprezentációit konkatenáljuk, majd ez végighalad egy teljesen összekötött blokkon, amely 7 rétegből áll. Ezek mindegyike a SELU [17]

aktivációs függvényt használja. Az egyes rétegek rendre 128, 256, 512, 1024, 512, 256 és 128 neuronnal rendelkeznek. A SELU aktivációval gyorsabb konvergenciát tapasztaltunk a háló tanítása közben a függvény által nyújtott belső normalizáció miatt.

$$SELU(x) = \lambda \begin{cases} x, & \text{ha } x > 0 \\ \alpha(e^x - 1), & \text{ha } x \leq 0 \end{cases} \tag{3.22}$$

Ezen blokk után található az első kimeneti réteg, ami sigmoid aktivációs függvény segítségével a kompozícióban soron következő lehetséges függvények valószínűségeit prediktálja. Ezen első, teljesen összekötött rétegekből álló blokk öt kisebb, hasonló blokkba is csatlakozik, melyek mindegyike öt réteget tartalmaz SELU aktivációs függvényekkel és rendre 128, 256, 512, 256 és 128 neuronnal.

A kisebb sűrű blokkok állítják elő a modell maradék öt kimenetét. Ezek vektorok, melyek mindegyike a kompozíció következő függvénye lehetséges paramétereinek valószínűségeit tartalmazzák. Ez az öt vektor a (1) bool lambda operátor, (2) numerikus lambda operátor, a (3) bool lambda numerikus argumentum, a (4) numerikus lambda numerikus argumentum és a (5) nemmagasabbrendű függvények paramétereinek feleltethető meg. Az összes kimeneti rétegen sigmoid aktivációs függvényt használunk a vektorok minden elemére. A háló legkisebb kimeneti vektorának 4 eleme van a legnagyobbnak pedig 17. A háló loss (L) értéke a 6 L_i-vel jelölt kimeneti loss érték összege, melyek mindegyike kereszt-entrópia függvény.

$$L(Y, \hat{Y}) = \sum_{i=1}^{6} L_i(Y_i, \hat{Y}_i) \tag{3.23}$$

3.4.2. Tanítás

A tanítási fázis megkezdése előtt a kompozíciókat szétbontjuk különálló függvényekre és ezeket mind one-hot vektorrá alakítjuk, hogy mindegyikhez tudjunk egy címkét rendelni a modell tanításához. A generált példák 98%-át használjuk tanítóhalmazként, a maradék 2%-ot pedig validációs halmazként. A teszthalmazokat kísérletenként külön generáljuk.

Az Adam optimalizációs algoritmust [18] használjuk ennek alapértelmezett hiperparamétereivel: $\beta_1 = 0.9$, $\beta_2 = 0.999$ és $\varepsilon = 10^{-8}$. A neurális háló egy Intel i5-7600K processzor-

ral és NVIDIA GTX 1070 GPU-val ellátott számítógépen volt tanítva öt epoch türelmű early stopping módszerrel. A hálót maximum 30 epoch-ig tanítottuk, amely átlagosan megközelítőleg 6 órát vett igénybe.

4. fejezet

Kísérletek

Az elkövetkezőkben taglalt kísérletek mindegyikét egy, a Google Cloud Platformon bérelt c2-standard-16 virtuális számítógépen futtattuk, amely egy 16 virtuális maggal rendelkező, Intel Cascade Lake architektúrájú processzorral volt ellátva. A számítógépben 64 GB memória volt, és nem rendelkezett dedikált GPU-val. A kísérletekhez felhasznált neurális háló öt hosszúságú példakompozíciókon volt tanítva, a bemeneti listák hossza pedig 15 és 20 közé esett. Minden programhoz 1 bemeneti-kimeneti példát biztosítottunk a tanítási folyamat során. Mindegyik kísérlethez egy egyenletes eloszlásból mintavételeztük az eredeti programtér elemeit, 1000-es mintamérettel.

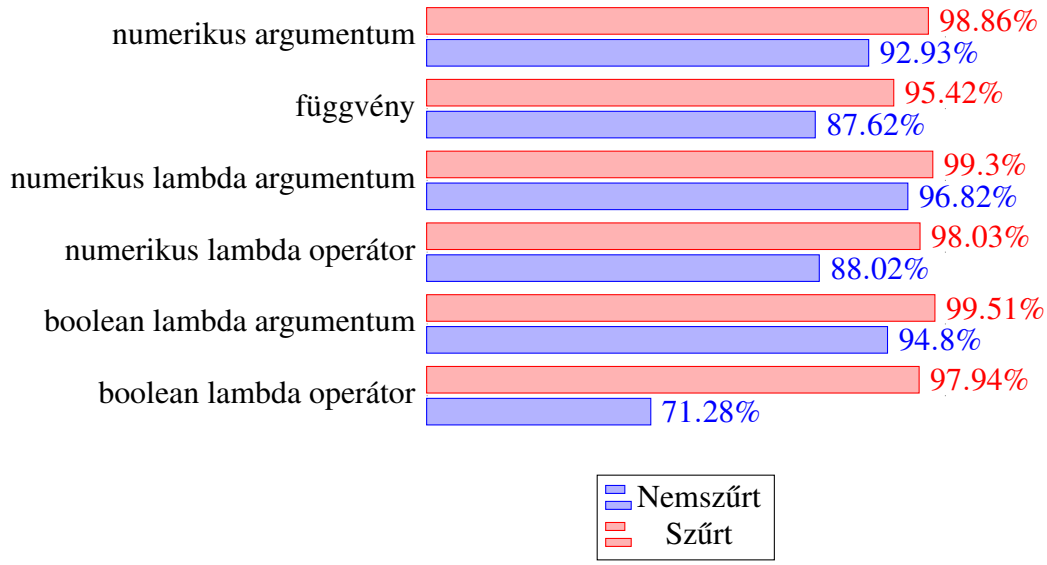
4.1. Az adathalmazok szűrése

A 3.1 fejezetben leírt adathalmaz-szűrési módszer nagy mértékben taníthatóbbá tette a feladatot a neurális háló számára, a háló összes kimeneti fejének pontosságát jelentősen növelte. A pontosságbeli változások a 4.1 ábrán láthatóak.

Ezt a szűrést a kísérletekhez generált összes adathalmazon elvégezzük.

4.2. Különböző rekurrens rétegek

Ebben a fejezetben különböző népszerű rekurrens rétegeknek a háló pontosságára kifejtett hatását vizsgáljuk. A kísérletekben a 3.3 alfejezetben már ismertetett Long Short-



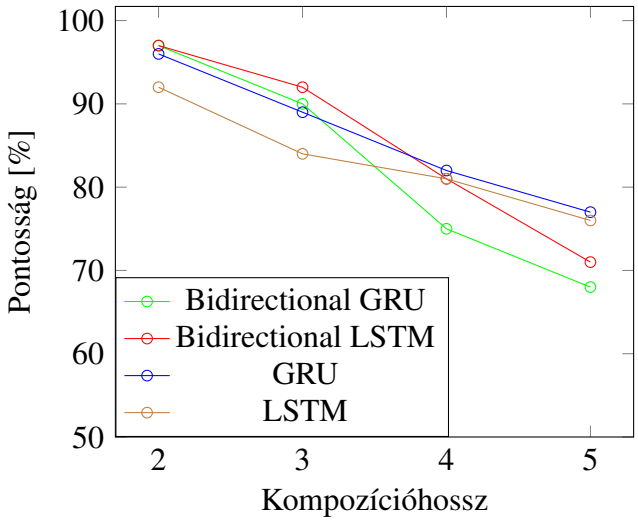
4.1. ábra. A tanítóhalmazon végrehajtott példaszűrés utáni javulás a neurális háló kimeneti fejeinek pontosságában. Látható, hogy a szűrés egy jóval taníthatóbb feladatot eredményez. Az eredményeket a GRU modellünk tanítása során mértük. Az egyes oszlopok a háló megfelelő kimeneti fejének a tanítás végén mért validációs pontosságát mutatják.

Term Memory (LSTM) [15], kétirányú LSTM [19], Gated Recurrent Unit (GRU) [16] és kétirányú GRU rétegeket használtunk fel.

Az első kísérletben azt vizsgáltuk, hogy az egyes rétegfajták pontossága hogyan változik a kompozícióhossz növelésével (4.2 ábra). A kétirányú rétegek szuboptimálisnak bizonyultak, ugyanis ezek – némileg meglepő módon – nem pontosították a modell predikcióit hosszabb függvénykompozíciók esetén, de a tanítás és tesztelés jelentősen több időt vett igénybe.

Annak ellenére, hogy a kétirányú LSTM éri el a legjobb eredményeket rövidebb kompozíciók esetén, ennek pontossága az egyirányú LSTM és GRU sejtek pontossága alá esik, ahogy a kompozícióhossz növekszik. Futásidő tekintetében a kétirányú LSTM a leglassabb. A kétirányú GRU modell a második leglassabb, és ennek teljesítménye a legrosszabb hosszabb kompozíciókra a vizsgált rétegek közül. Az egyirányú LSTM és GRU sejtek közül a GRU az általunk preferált, ugyanis konzisztensen jó eredményeket adott mind futásidő, mind pontosság tekintetében.

A második kísérletben a bemeneti és kimeneti listák hosszának növelésének a FlexCoder rendszer pontosságára kifejtett hatását vizsgáltuk. Tíz és ötven közötti hosszúságú bemeneti és kimeneti listákat generáltunk ötös lépésközzel a teszt végrehajtásához. A 4.3 ábra



4.2. ábra. A különböző rekurrens rétegek teljesítménye a kompozícióhossz függvényében. Mind a kétirányú modellek, mind az LSTM modell esetén 200 dimenziós rejtett állapotot adtunk meg. A GRU modell esetén ezt 256-ra növeltük, ugyanis ezek kevesebb számítást igényelnek a gátak kevesebb számának köszönhetően. Bár a kétirányú LSTM éri el a legjobb eredményt rövid függvénykompozíciók esetén, ennek teljesítménye jelentősen csökken, ahogy a kompozícióhossz növekszik. Emellett ez a modell bizonyult a leglassabbnak is. A sebességet és pontosságot is figyelembe véve a GRU modell a legkedvezőbb számunkra.

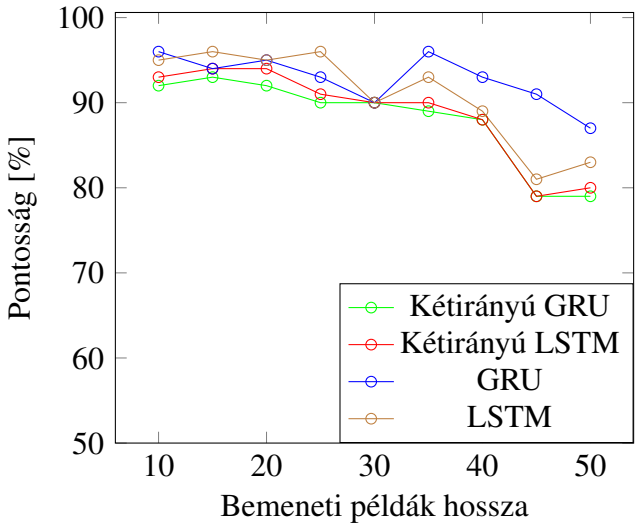
megmutatja, hogy a FlexCoder GRU rekurrens réteggel képes volt hosszabb listahosszokra is általánosítani.

Az első kísérlethez hasonlóan a GRU-t használó modell bizonyult a legpontosabbnak, és ez rendelkezett a legrövidebb futásidővel is. A kétirányú modellek hasonlóan teljesítettek, de a kétirányú LSTM-et használó neurális háló jelentősen lassabb volt. A GRU modell konzisztensen jobb eredményt tudott nyújtani hosszabb bemenetek esetén, mint az LSTM-alapú modell.

Ezen két kísérlet eredményei alapján a GRU sejtet használó neurális hálót választottuk a további kísérletek elvégzéséhez.

4.3. Pontosság és futási idő

A 4.1 és 4.2 táblázatok a FlexCoder rendszer pontosságát és futásidejét mutatják a bemeneti és kimeneti listák számának, illetve a kompozícióhossz függvényében. A bemeneti és kimeneti listák számának növelésével a feladat jóval specifikusabbá válik: egy olyan program szintetizálása, amely mindegyik bemeneti-kimeneti párt kielégít, egy komple-



4.3. ábra. A FlexCoder rendszer pontossága négy különböző rekurrens réteg használatával, a bemeneti lista hosszának függvényében. A GRU-t használó modell általánosít legjobban a hosszabb bemenetekre.

komp. hossz \ **I/O párok**	1	2	3	4	5
2	97%	97%	97%	96%	97%
3	99%	94%	92%	95%	93%
4	97%	89%	86%	83%	85%
5	96%	88%	81%	79%	78%

4.1. táblázat. A kompozícióhossz, a bemeneti-kimeneti párok száma és a pontosság közötti összefüggés. A kompozícióhossz, illetve a bemeneti-kimeneti párok számának növelése – várható módon – majdnem minden esetben csökkenti a pontosságot.

xebb feladat, ugyanis a lehetséges megoldások halmaza egyre kisebb. Hasonlóan, ahogy a kompozícióhosszt növeljük, a lehetséges programok tere exponenciálisan növekszik.

A FlexCoder által elért pontosság összehasonlítható a PCCoder-rel, amennyiben az utóbbi rendszerben kicseréljük az időlimitet a FlexCoder rendszerben használt iterációs limitre. A végrehajtási idő terén a FlexCoder néha visszaesik a PCCoder-hez képest, azonban általánosan elmondható, hogy a két rendszer teljesítménye hasonló.

4.4. Összehasonlítás a PCCoder-rel

Rendszerünket a PCCoder-rel [8] hasonlítjuk össze, amely teljesítményben több nagyságrenddel felülmúlta a DeepCoder rendszert.

komp. hossz \ I/O párok	1	2	3	4	5
2	211 s	255 s	264 s	277 s	274 s
3	524 s	1082 s	1239 s	1141 s	1291 s
4	1224 s	2900 s	3250 s	3923 s	3620 s
5	1520 s	3445 s	4986 s	5537 s	5530 s

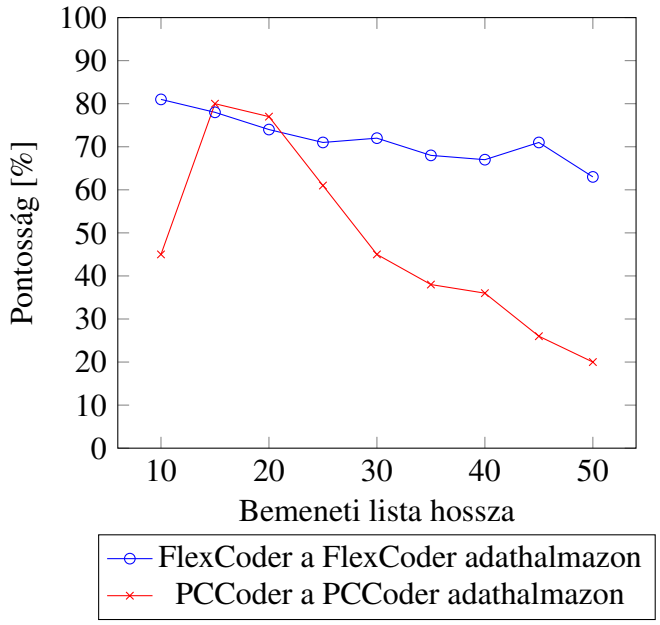
4.2. táblázat. A kompozícióhossz, a bemeneti-kimeneti párok száma és a végrehajtási idő közötti összefüggés. A kompozícióhossz, illetve a bemeneti-kimeneti párok számának növelése – várható módon – majdnem minden esetben növeli a futásidőt.

A programszintézis problémakörét jelentősen más szemszögből közelítjük meg, mint a PCCoder (lásd 1 fejezet és 3.1 alfejezet). Mi függvénykompozíciókat szintetizálunk, ők utasítások sorozatát. A nyelvtanaink kifejezőereje szintén különbözik: Egyrészről a nyelvtanunk nem tartalmaz néhány függvényt, melyeket a PCCoder által használt nyelvtan igen (például `ZipWith`, vagy `Scanl1`). Másrészről azonban a nyelvtanunk kifejezőereje sokkal nagyobb a lehetséges paraméterértékeket tekintve.

Nyelvtanunk 151 különböző függvényt képes kifejezni, melyek közül 130 a kompozícióban bárhol elhelyezkedhet, míg 21 csak a legkülsőbb függvény lehet, ugyanis ezek skaláris értéket adnak vissza, és minden függvényünk listát vár bemenetként. A PCCoder által használt DSL 105 különböző függvényt képes kifejezni. Az öt hosszúságú lehetséges programok száma megközelítőleg $43,13$ millió a FlexCoder rendszer esetén, és hozzávetőleg $12,76$ millió a PCCoder esetén, amely azt jelenti, hogy a FlexCoder programtere $3,38$-szorosa a PCCoder-ének, öt hosszúságú programok esetén.

Annak céljából, hogy ezen különbségek ellenére korrekt és hiteles összehasonlítást végezhessünk a két rendszer között, olyan kísérleteket is végeztünk, melyekben mindkét rendszer teljesítményét a saját adathalmazán vizsgáljuk, illetve olyanokat is, melyekben egymás adathalmazán vetjük össze teljesítményeiket.

Az első kísérletben megfigyelhetünk egy általunk bármilyen programszintézis-rendszer esetén kifejezetten fontosnak vélt aspektust: milyen sikeresen általánosít különböző bemeneti hosszokra. Mind a FlexCoder-t, mind a PCCoder-t 15 és 20 közötti hosszúságú bemeneti és kimeneti listákon tanítottuk, öt hosszúságú kompozíciókkal. Mivel a PCCoder esetén kénytelenek voltunk megadni egy maximális hosszt a bemeneti és kimeneti listákra, ezt számukra a kísérletben a legkedvezőbb módon 50-nek határoztuk meg. A rend-

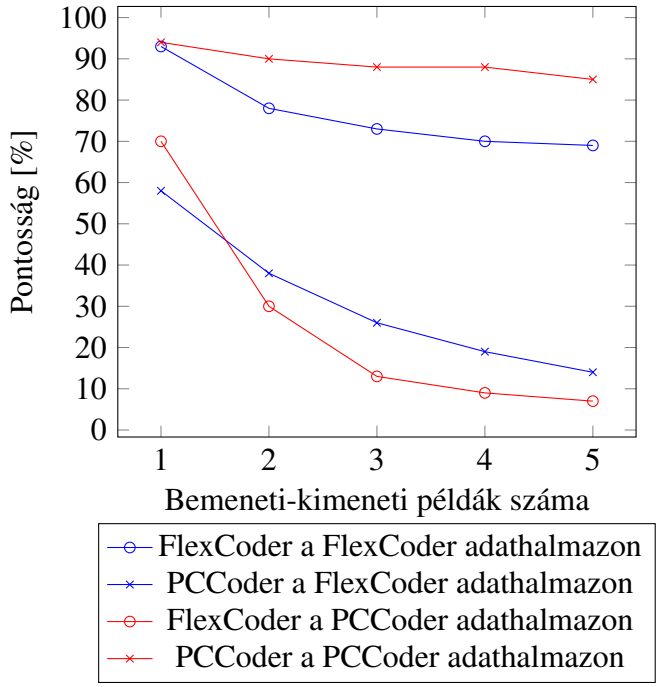


4.4. ábra. A FlexCoder és PCCoder rendszerek pontossága a bemeneti listák hosszának függvényében. Mindkét rendszer 15 és 20 közötti hosszúságú bemeneti-kimeneti párokon volt tanítva, ötös kompozícióhosszal. A PCCoder esetén a maximális listahosszt 50-re állítottuk. A rendszereket 10 és 50 közötti hosszúságú bemenetű bemeneti-kimeneti párokon teszteltük ötös lépésközzel, programonként 5 bemeneti-kimeneti párral, mindkét rendszert a saját adathalmazán.

szereket 10-től ötösével 50-ig terjedő hosszúságú bemenetű bemeneti-kimeneti példákon teszteltük, programonként 5 bemeneti-kimeneti példával. Ezen kísérletben mindkét rendszert a saját adathalmazán teszteltük. Az eredmények a 4.4 ábrán láthatóak.

A második kísérletben a FlexCoder és PCCoder rendszereket egy kevésbé valósághű esetben teszteltük le, melyben a PCCoder a legjobban teljesít: olyan bemenetilista-hosszokon, amelyeken tanítva lettek a rendszerek. Ebben a kísérletben a PCCoder-nek nem kell különböző bemenetilista-hosszokra általánosítania. A két rendszert saját adathalmazukon és egymás adathalmazán is összehasonlítjuk a 4.5 ábrán.

Rendszerünk egy mélységi korlátot definiál a keresési algoritmus esetén, míg a PCCoder által használt algoritmus időlimitet vezet be. Ahhoz, hogy az összehasonlítás korrekt és hiteles legyen, algoritmusunk mélységi korlátját időlimitre cseréltük, és 60 másodperces korlátot adtunk meg, a PCCoder-hez hasonlóan. Algoritmusunkban az időlimit bevezetése pár százalékkal rontotta rendszerünk pontosságát a 4.3 ábrához képest. Ez azt jelenti, hogy a FlexCoder mégjobban tudna teljesíteni az eredeti mélységi korláttal. A kísérletben használt paraméterek megegyeznek az első kísérletben használtakkal a bemenetilista-



4.5. ábra. A FlexCoder és PCCoder rendszerek pontossága a saját adathalmazukon és egymás adathalmazán. A paraméterek megegyeznek az első kísérletben használtakkal, kivéve, hogy most nem vizsgáljuk a bemenetilista-hosszokon történő általánosítást: a bemenetilista-hosszok megegyeznek a tanításnál használtakkal. A bemeneti-kimeneti párok száma 1-től 5-ig terjed.

hosszokon kívül, melyek megegyeznek a tanításnál használt hosszokkal. A bemeneti-kimeneti példák száma 1-től 5-ig terjed.

5. fejezet

Diszkusszió

A kutatási terület ismertetett munkái között még nem láthattunk sok példát a valóélet-beli környezetben való alkalmazására.

Úgy gondoljuk, hogy ennek oka leginkább (i) a statikus, vagy felülről korlátos bemenetilista-hosszok limitációja; (ii) a nyelvtan tokenjeinek agglutinációja, úgymint a lambdafüggvényekben az operátorok és ezek integer paramétereinek együttes kezelése (például $(+1)$ és $(*2)$); (iii) a lambdakifejezések limitált integer intervalluma, és (iv) a bemenetek, közbülső értékek és kimenetek limitált integer intervalluma.

A második pont következménye, hogy a szükséges lambdafüggvények száma a lambdaoperátorok és ezek lehetséges paraméterei számának szorzata. Például külön lambdafüggvények szükségesek a $(+1)$, $(+2)$, $\ldots$ $(+n)$ transzformációk kifejezéséhez. Ez a rendszerek által megengedett lehetséges lambdafüggvény-kombinációk számát jelentősen redukálja. Az első pontból következik, hogy a rendszerek nem általánosítanak megfelelően egy konstans L felső korlátnál hosszabb bemenetekre.

Dolgozatunkban kiemelt figyelmet fordítottunk ezen limitációk feloldására. E célból létrehoztunk egy függvénykompozíció alapú DSL-t, amely a DeepCoder által felhasználtakhoz hasonló függvényekből építkezik, és melyben a lambdakifejezések operátorait és ezek paramétereit külön kezeljük. Ez lehetővé tette számunkra, hogy nagy mértékben csökkentsük a szükséges lambdakifejezések számát, a lambdaoperátorok és ezek paraméterei számának szorzatáról ezek összegére (amennyiben a paramétereket nulláris függvényekként tekintjük), és így bővítsük a rendszerünk számára elérhető lehetséges lambdakife-

jezések halmazát a kutatási terület eddigi munkáihoz képest: a megengedett numerikus értékek halmazát a PCCoder rendszerben látott {-1, 0, . . . , 4} halmazról {-8, -7, . . . , 8}-ra terjesztettük ki. Ez a megközelítés megoldja a paraméterfüggvények kötött jellegét, ugyanis a terület ismert rendszerei csak néhány előre definiált lambdafüggvénnyel rendelkeznek, összetartozó operátorral és operandussal. Jelenleg rendszerünkben csupán két magasabbrendű függvény (**map** és **filter**) használja ezeket a lambdafüggvényeket. Azt gondoljuk, hogy a FlexCoder még jobban teljesítene az ismert rendszerekhez képest, ha kiterjesztenénk a nyelvtanunkat több magasabbrendű függvénnyel.

Kiterjesztettük továbbá a köztes eredménylistákban és kimeneti listákban található egész számok halmazát is az eddigi munkákban látható $\{-256, -255, \ldots, 256\}$-ról $\{-1024, -1023, \ldots, 1024\}$-re, így a példagenerálási fázisban jóval kevesebb programot szűrünk ki a lehetséges intervallum megszorításainak megsértése miatt. Fontos megemlítenünk, hogy nem használtunk a természetes nyelvfeldolgozás területén népszerű beágyazásokat a bemeneti lista elemeire, ugyanis ez megszorításokat vezetne be a bemeneti és kimeneti listák elemeinek intervallumát illetően.

Egy több komponensből álló, mély neurális hálót alkalmaztunk a gráfkereső algoritmusunk navigálásának segítésére, a DeepCoder és PCCoder rendszerekhez hasonlóan. A főbb eltéréseket a keresési algoritmus bemutatásánál taglaltuk. A neurális háló kötetlen hosszúságú bemeneti-kimeneti párokat fogad el bemenetként, és a problémát megoldó függvénykompozíció következő paraméterezett függvényét becsüli meg. Így a háló heurisztikaként szolgál a gráfkereső algoritmusunknak, melyet a nyalábkeresésre alapoztunk. Ez előre definiált, előzetes futási statisztikák alapján optimalizált nyaláb méreteket használ a keresőfa minden egyes szintjén. A dolgozatban bemutatott első kísérleteinkben olyan függvénykompozíciókat használtunk, melyekben az egyes függvények egyetlen listát várnak paraméterül további paramétereik mellett. A kompozíció következő függvényét az addigi részkompozíció lista eredményére alkalmazzuk. Ebben az értelemben a DSL-ünk kifejezőkészsége gyengébb a DeepCoder rendszerben használt nyelvtannál: például nem tartalmazza a `ZipWith`, vagy a `Scanl1` függvényeket.

A FlexCoder a PCCoder-rel ellentétben jól általánosított a különböző bemenetilista-hosszokra. A PCCoder rendszer csak azokon a bemenetilista-hosszokon teljesített jól, melyeken tanítva volt. Mivel a DeepCoder és a PCCoder által használt neurális hálók

nem szekvenciálisan dolgozzák fel a bemenetet, ezért maximum L hosszúságú bemenetet fogadnak el. (A bemenet L-nél rövidebb is lehet, az általuk használt padding miatt.) Az alapértelmezett maximális hossz mindkét rendszer esetében $L = 20$, azonban rendszerünk PCCoder-rel történő összehasonlítása során a korrekt összehasonlíthatóság érdekében ezt a korlátot 50-nek határoztuk meg. A FlexCoder ezt a problémát azzal oldja meg, hogy GRU [16] rétegeket használ a bemenetek feldolgozásához, így bemeneti méretek széles tartományával képes dolgozni. A PCCoder 50-nél hosszabb bemenetekre való alkalmazása a rendszer újratanítását igényelné, melynek elején egy nagyobb felső korlátot kellene meghatározni.

Olyan esetben is összehasonlítottuk a két rendszer teljesítményét, melyben a teszt során olyan bemenetilista-hosszokon kellett jól teljesítenie az egyes rendszereknek, melyeken előzőleg tanítva lettek. Ebben az esetben a PCCoder-nek nem kellett általánosítania különböző hosszokra. Ebben a könnyebb és kevésbé realisztikus esetben mindkét rendszer jobban teljesített a másiknál a saját adathalmazukon. Mindkét rendszer rosszul teljesített a másik DSL-jén.

Úgy gondoljuk, hogy ez utóbbi esetben az áll a FlexCoder rosszabb teljesítménye mögött, hogy ezen első kísérleteinkben olyan függvénykompozíciókat használtunk, melyek esetén – a rögzített paramétereken kívül – mindegyik függvény egyetlen listát kapott bemenetként. Legtöbb függvényünk esetén a kimenet is egyetlen lista volt. Ezen megszorítás relaxálása a FlexCoder rendszer kifejezőkészségét jelentősen magasabb szintre emelné, és lehetővé tehetné a rendszernek azt, hogy felülmúlja a PCCoder teljesítményét a legtöbb kísérletben.

6. fejezet

Konklúzió

A DeepCoder által megalkotott DSL kifejezőereje limitált, ahogy maguk a szerzők is kimondják ezt cikkükben [7]. Cikkünk elsődleges motivációja az volt, hogy ezt kibővítsük és a valóélet-beli alkalmazások felé mozduljunk el.

Bemutattuk a FlexCoder-t, melyben a fő kontribúcióink az alábbiak:

- Egy rekurrens neurális háló architektúrát mutattunk be, amely jól általánosít a különböző bemeneti hosszokra.
- Nyelvtanunkban a lambdakifejezések operátorait külön kezeltük ezek paramétereitől. Ez lehetővé teszi a paraméterek intervallumának szignifikáns növelését.
- Architektúránk nem szabott mesterséges határt a bemeneti, közbülső és kimeneti listák elemeinek intervallumára. Az eddigi munkákhoz képest kiterjesztettük a közbülső és kimeneti listák elemeinek intervallumát.

Az elsődleges limitációt és a legígéretesebb jövőbeli munkát abban látjuk, hogy lehetővé tegyük olyan függvénykompozíciók használatát, melyekben az egyes függvények több nemrögzített paramétert is kaphatnak. Ez lehetővé tenné a nyelvtanunk számára, hogy kifejezze a `ZipWith` és a `Scanl1`, illetve további érdekes függvényeket, például azt, mely a bemeneti lista első n elemét adja vissza új listaként, ahol n a bemeneti lista maximális eleme. Ez utóbbi függvényt a $take(max(arr), arr)$ formában fejezhetnénk ki, amennyiben megengednénk kifejezéseket rögzített paraméterként.

Neurális hálónkkal foytatott kísérletezéseinkben érdemes lehet NTM sejteket [20] bevezetni a GRU sejtek helyett, ugyanis az NTM sejtek kifejezetten hatékonyan képesek egyszerű algoritmusok megtanulására, például listák rendezésére, amely megtalálható az általunk használt függvények között is.

A FlexCoder pontosnak és hatékonynak bizonyult még ötven hosszúságú bemeneti listákra történő általánosítás esetén is, egy jóval szélesebb paraméterkészlettel, mint a jelenlegi rendszerek. Reméljük, hogy munkánkkal tettünk egy lépést a programszintézis széleskörű valóélet-beli alkalmazása felé.

A dolgozatunkban közölt eredményeink egy cikk alapját is képzik, melyet teljes cikként elfogadtak a 2021-es International Conference on Pattern Recognition Applications and Methods (ICPRAM) konferenciára [10].

Köszönetnyilvánítás

A szerzők szeretnék köszönetüket kifejezni Borsi Zsoltnak és Várkonyi Teréz Annának az értékes tanácsadásukért. A munka az EFOP-3.6.3-VEKOP-16-2017-00001: Tehetséggondozás az Autonóm Járművezérlési Technológiák projekt keretén belül készült. A projektet a Magyar Kormány támogatja, és az Európai Szociális Alap társfinanszírozza.

Irodalomjegyzék

[1] Sumit Gulwani, Oleksandr Polozov és Rishabh Singh. “Program Synthesis”. *Foundations and Trends® in Programming Languages* 4.1-2 (2017), 1–119. old. ISSN: 2325-1107. DOI: 10.1561/2500000010. URL: http://dx.doi.org/10.1561/2500000010.

[2] Pengcheng Yin és tsai. *StructVAE: Tree-structured Latent Variable Models for Semi-supervised Semantic Parsing*. 2018. arXiv: 1806.07832 [cs.CL].

[3] Sumit Gulwani. “Programming by Examples: Applications, Algorithms, and Ambiguity Resolution”. *Automated Reasoning*. Szerk. Nicola Olivetti és Ashish Tiwari. Cham: Springer International Publishing, 2016, 9–14. old. ISBN: 978-3-319-40229-1.

[4] Emilio Parisotto és tsai. *Neuro-Symbolic Program Synthesis*. 2016. arXiv: 1611.01855 [cs.AI].

[5] Woosuk Lee és tsai. “Accelerating search-based program synthesis using learned probabilistic models”. *ACM SIGPLAN Notices* 53.4 (2018), 436–449. old.

[6] Ashwin Kalyan és tsai. “Neural-guided deductive search for real-time program synthesis from examples”. *arXiv preprint arXiv:1804.01186* (2018).

[7] Matej Balog és tsai. “Deepcoder: Learning to write programs”. *arXiv preprint arXiv:1611.01989* (2016).

[8] Amit Zohar és Lior Wolf. “Automatic program synthesis of long programs with a learned garbage collector”. *Advances in Neural Information Processing Systems*. 2018, 2094–2103. old.

[9] Yu Feng és tsai. “Program Synthesis Using Conflict-Driven Learning”. *SIGPLAN Not.* 53.4 (2018. jún.), 420–435. ISSN: 0362-1340. DOI: 10.1145/3296979.3192382. URL: https://doi.org/10.1145/3296979.3192382.

[10] Gyarmathy Bálint és tsai. “FlexCoder: Practical program synthesis with flexible input lengths and expressive lambda functions”. (in press).

[11] Weixiong Zhang. “Search techniques”. *Handbook of data mining and knowledge discovery*. 2002, 169–184. old.

[12] Noam Chomsky és Marcel P Schützenberger. “The algebraic theory of context-free languages”. *Studies in Logic and the Foundations of Mathematics*. 26. köt. Elsevier, 1959, 118–161. old.

[13] Steven Bird, Ewan Klein és Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit*. " O’Reilly Media, Inc.", 2009.

[14] Ehud Y Shapiro. *Algorithmic Program Debugging. ACM Distinguished Dissertation*. 1982.

[15] Sepp Hochreiter és Jürgen Schmidhuber. “Long short-term memory”. *Neural computation* 9.8 (1997), 1735–1780. old.

[16] Kyunghyun Cho és tsai. “Learning phrase representations using RNN encoder-decoder for statistical machine translation”. *arXiv preprint arXiv:1406.1078* (2014).

[17] Günter Klambauer és tsai. “Self-normalizing neural networks”. *Advances in neural information processing systems*. 2017, 971–980. old.

[18] Diederik P Kingma és Jimmy Ba. “Adam: A method for stochastic optimization”. *arXiv preprint arXiv:1412.6980* (2014).

[19] Mike Schuster és Kuldip K Paliwal. “Bidirectional recurrent neural networks”. *IEEE transactions on Signal Processing* 45.11 (1997), 2673–2681. old.

[20] Alex Graves, Greg Wayne és Ivo Danihelka. “Neural turing machines”. *arXiv preprint arXiv:1410.5401* (2014).

Ábrák jegyzéke

Táblázatok jegyzéke