Eötvös Loránd Tudományegyetem

Informatikai Kar

Programozáselmélet és Szoftvertechnológiai Tanszék

Hatékony példaalapú programszintézis fastruktúrájú nyelvtanon

Témavezető:
dr. Pintér Balázs
egyetemi adjunktus, PhD

Szerző:
Mucsányi Bálint Hunor
programtervező informatikus BSc

Budapest, 2021

EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

SZAKDOLGOZAT TÉMABEJELENTŐ

Hallgató adatai:

Név: Mucsányi Bálint Hunor

Neptun kód: JLV5AE

Képzési adatok:

Szak: programtervező informatikus, alapképzés (BA/BSc)

Tagozat: Nappali

Belső témavezetővel rendelkezem

Témavezető neve: *dr. Pintér Balázs*

munkahelyének neve, tanszéke: ELTE IK, Programozáselmélet és Szoftvertechnológiai Tanszék

munkahelyének címe: 1117 Budapest, Pázmány Péter sétány 1/C

beosztás és iskolai végzettsége: egyetemi adjunktus, PhD

A szakdolgozat címe: Hatékony példaalapú programszintézis fastruktúrájú nyelvtanon

A szakdolgozat témája:

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

A programszintézis kutatási területe hosszú múltra tekint vissza. Már a mesterséges intelligencia kutatások kezdetén is felmerült a kérdés, hogy képes lehet-e egy program formális vagy informális specifikáció alapján egy annak megfelelő programot megadni. A kezdeti algoritmusok a problématér mérete miatt megfigyelhető kombinatorikus robbanást heurisztikákkal igyekeztek elkerülni.

Egy másik érdekes út erre a gépi tanulás használata. Az elmúlt pár évben jó pár új módszer született a programszintézis területén, melyek jelentős eredményeket értek el mély tanulási algoritmusok használatával. Ezek zöme, mint ahogy szakdolgozatom is, az informális specifikációkra fókuszál: a feladat bemeneti és kimeneti listapárokból álló példákhoz olyan programok (függvénykompozíciók) generálása, amik a bemenetekből képesek előállítani a kimeneteket.

Több eredményes módszerben is limitációt jelent a szintetizált programok vagy függvénykompozíciók szekvenciális szerkezete. Egy TDK dolgozat keretében hallgatótársaimmal közösen fejlesztettünk egy rendszert Python programozási nyelven, melyben szintén megjelenik ez a korlát. Szakdolgozatom célja ennek a korlátnak a feloldása: a meglévő rendszer kiterjesztése olyan programokra, ahol a függvénykompozíció tetszőleges fa lehet.

Ehhez tervem a kutatási terület további tanulmányozása és a lehetséges megoldások beépítése a rendszerbe. A területen már láthatunk ebben segítő, eredményes munkákat: ezek általában arra épülnek, hogy részlegesen felépített programokhoz is képesek megbecsülni a következő lépést a fa építésében.

Budapest, 2020.11.30.

Tartalomjegyzék

1. fejezet

Bevezetés

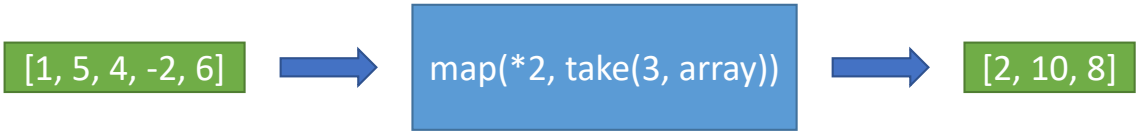
A programszintézisnek két fő ága ismeretes [1]. *Deduktív programszintézis* esetén egy igazolhatóan helyes program generálása a cél, amely eleget tesz egy formális specfikációnak. *Induktív programszintézis* esetén a szintetizálandó program elvárt működése csak példák által van demonstrálva, vagy egy informális szöveges leírással [2]. Ebben az esetben a siker kifejezetten fontos feltétele, hogy olyan sarkalatos példákkal is ellássuk a rendszert, melyek egyértelművé teszik a megadott problémát.

A Példaalapú Programozás (Programming by Examples, PbE) a programszintézis egy demonstratív megközelítése, mely a szakdolgozatomban bemutatott rendszerben is felhasználásra kerül. A példák PbE esetén rendezett párok (tuple), melyek bemenetekből és az azokra elvárt kimenetekből állnak [3]. A rendszer ezeket használja fel ahhoz, hogy egy olyan programot állítson elő (1.2 ábra), amely a bemenetekből előállítja a kimeneteket (1.1 ábra). Ez a program megadható utasítások sorozatával [4, 5], vagy akár egy függvénykompozícióval [6] is. Ezen rendszerek bemenetei általában bemeneti és kimeneti listapárok, melyekben a listák egész számokat tartalmaznak [4, 5, 6], vagy pedig bemeneti-kimeneti szövegpárok [7, 8, 9].

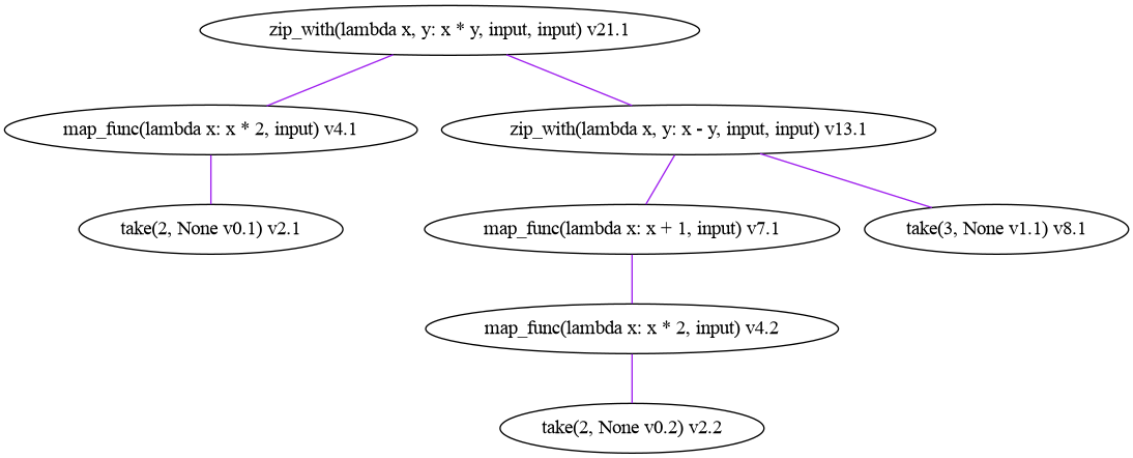
A lehetséges függvénykompozíciók vagy programok általában egy környezetfüggetlen nyelvtan (CFG) [10] által kerülnek megadásra, amelyből például a szakdolgozatban bemutatott rendszer esetén az 1.1 ábrán látható függvénykompozíció is levezethető.

A mélytanuló algoritmusokkal történő kisérletezés során a PbE kutatási területe nagy áttöréseket ért el, mind a modellek pontosságában, mind ezek sebességében [4]. Ezen algoritmusokat leginkább heurisztikaként használják fel a szintetizálási folya-

1.1. ábra. Az ábra egy PbE feladatot mutatja be egy bevezető példán keresztül, amely az egyszerűség kedvéért egyetlen bemeneti ($[1, 5, 4, -2, 6]$) és kimeneti ($[3, 15, 12]$) listát vonultat fel. A feladat tehát ebben az esetben egy PbE rendszer számára az, hogy egy olyan függvénykompozíciót adjon meg, amely a megadott inputlistát a megadott outputlistára transzformálja. Ennek a specifikációnak megfelel a $map(*3, take(3, array))$ függvénykompozíció. Fontos megemlíteni, hogy a ma ismert legjobb példaalapú programszintézis-rendszerek több input-output listapárt is képesek feldolgozni, amely esetben a feladat egy olyan függvénykompozíció szintetizálása, amely az összes inputlistát a hozzájuk tartozó kimenetekre képzi le.



1.2. ábra. Az ábrán a 1.1 ábrán a szintézisfolyamat eredményéül kapott $map(*3, take(3, array))$ függvénykompozíciót láthatjuk. Az ezt felépítő függvények megfelelő sorrendben (először a *take*, majd a *map* függvény) történő végrehajtása az $[1, 5, 4, -2, 6]$ listán valóban a $[3, 15, 12]$ listát eredményezi, így a programszintézis sikeres.



1.3. ábra. Az ábrán egy olyan, a szakdolgozatban bemutatott programszintézis- rendszer nyelvtanából levezethető kompozíció látható, mely magában foglalja a két listát, vagy listára kiértékelhető részkompozíciót paraméterül váró *ZipWith* függvényt, amely a két bemeneti lista azonos indexein található elemekre végrehajta a bináris operátorát, így képezve az outputlista ugyanazon indexén található elemet. Az ábrán None érték látható a bemeneti listák helyén, ugyanis a programcsomag lehetőséget ad kiértékelhetetlen, azaz bemeneti listákkal még nem ellátott függvénykompozíciók megalkotására is, melyeket később feltölthetünk tetszőleges listákkal.

matban. Ezek a heurisztikák például a feladat állapotgráfján alkalmazható gráfkereső algoritmusokban használhatók fel. Láthatunk példákat a kezdeti PbE rendszerekben található heurisztikák mélytanuló módszerekkel történő kiegészítésére is. A neurális hálók predikciói tehát jól ki tudják egészíteni a feladat sajátosságait kihasználó ismereteket [9, 8].

Szakdolgozatom témájának megválasztásában nagy segítséget nyújtott a dr. habil. Gregorics Tibor által vezetett MLforSE kutatócsoport, melyhez 2019-ben csatlakoztam. A kutatócsoport gépi tanulási módszerekkel foglalkozik a szoftvertechnológia területén, mint például mutability problémák detektálása Java programozási nyelven írt forráskódokban, Python kód idiomatizálása gépi tanulással, vagy példaalapú programszintézis mélytanuló algoritmussal segítve.

Én ez utóbbi rendszer implementálásához csatlakoztam, amely egy mai napig fejlesztett szoftver. A Gyarmathy et al. [11] cikkben bemutatott rendszer egy programszintézis-modell, melynek feladata egy olyan függvénykompozíció megtalálása, amely a rendszernek megadott bemeneti példákat az azokhoz tartozó, szintén megadott kimeneti példákra transzformálja. A bemeneti példák a rendszerben minden esetben listák, a kimeneti példák pedig lehetnek egyaránt listák és skalárértékek is. A rendszert FlexCoder-nek neveztük el, ugyanis az ELTE MLforSE kutatócsoportjában végzett munkám során arra a következtetésre jutottunk csapattársaimmal, hogy az jóval flexibilisebb az annak átadott listapárok hozzával szemben, mint az eddigi legjobb rendszerek. Ezt az eredményt azért tartjuk fontosnak, mert a valóéletbeli problémák esetén nem feltételezhetünk minden esetben egy felső korlátot a bemeneti és kimeneti listák hosszára. Programszintézis-modellünk nyelvtana lineáris, nincs kiterjesztve fastruktúrájú nyelvtanra.

Szakdolgozatom témája az *International Conference on Pattern Recognition Applications and Methods* konferencián csapattársaimmal bemutatott cikkben [11] taglalt rendszer továbbfejlesztése egy olyan környezetfüggetlen nyelvtanon történő felhasználásra, amely bevezet egy olyan függvényt is (*ZipWith*), amely egyszerre két listát, vagy listára kiértékelhető függvénykompozíciót kap paraméteréül. Az ezen függvényt tartalmazó kompozíciók így már nem szekvenciális, hanem fastruktúrájú formát öltenek, ahogy az 1.3 ábra is mutatja.

A szoftver fejlesztésében csapattársaim, Gyarmathy Bálint és Czapp Ádám is

részt vesznek. Ők a nyalábkeresési algoritmus implementálását végzik, amely így nem képzi szakdolgozatom szerves részét. A rendszer további komponensei önálló munkám eredményei.

Az eredeti rendszer a 2021-es Kari Tudományos Diákköri Konferencián bemutatásra került, ahol I. helyet kapott a "FlexCoder: Gyakorlati programszintézis flexibilis inputhosszokkal és kifejező lambdafüggvényekkel" című, Gyarmathy Bálint, Czapp Ádám és jómagam által írt dolgozat, és továbbküldésre került a 2021-es Országos Tudományos Diákköri Konferenciára. Itt II. helyet értünk el.

Az elkészült alkalmazás lehetőséget biztosít a programszintézis-rendszerben használt neurális háló számára tanítópéldák generálására, annak betanítására, majd a tanítás eredményességének ellenőrzésére is egy tesztadathalmazon. Az így betanított neurális hálót ezután fel is lehet használni egy, a felhasználó által megadott programszintézis-feladat megoldása során. Mindezen opciók egy grafikus felületen elérhetőek, amely a projekt többi részével együtt Python programozási nyelven íródott. A GUI fejlesztéséhez a PyQt5 programcsomagot használtam, a neurális háló tanításában pedig leginkább a PyTorch és PyTorch Lightning csomagok nyújtottak segítséget.

2. fejezet

Felhasználói dokumentáció

2.1. Az alkalmazás rövid ismertetése

A program lehetővé teszi, hogy a felhasználó megismerkedjen a programszintézis feladatával, illetve hogy a rendszer szerves részét képző neurális hálónak tanító, validációs és tesztelő adathalmazt generáljon, majd ezekkel betanítsa a hálót a választott optimalizációs módszerrel. Az így kapott, már optimalizált súlyokkal rendelkező neurális hálót a felhasználó perzisztálhatja, majd a rendszer további komponenseivel együtt felhasználhatja széleskörű programszintézis-problémák megoldására, melyekhez számos beállítható hiperparaméter tartozik. A grafikus felület törekszik az előzőleg említett funkciók használatának felhasználóbaráttá tételére, illetve statisztikát is készít a rendszer működésének egyes fejezeteiről, melyeket grafikonok segítségével mutat be a felhasználónak.

2.2. Hardveres követelmények és a felhasználói környezet kialakítása

Az alkalmazás fejlesztése során kiemelt figyelem irányult arra, hogy széles körben alkalmazható program keletkezzen, így a rendszerhez csak ajánlott hardverek tartoznak, egzakt minimális követelmények nem. A program mind Linux openSUSE és Manjaro disztribúciókon, mind Microsoft Windows 10 rendszeren futtatható a Python programozási nyelv és a felhasznált programcsomagok platformfüggetlen-

ségének köszönhetően. Fontos azonban megemlíteni, hogy a neurális háló tanítási folyamata lényegesen gyorsabb, amennyiben az egy, a CUDA technológiát támogató videokártyán megy végbe. Az alábbi hardverelemekkel teljesítményben megegyező, vagy azoknál jobb hardveres eszközök használata ajánlott:

- Intel®Core ™i7-7700HQ CPU @ 2.80 GHz processzor
- 8 GB @ 2400 MHz RAM
- 120 GB SSD
- NVIDIA®GeForce™GTX 1050 Ti 4 GB videokártya

A felhasználónak a hardveres követelmények vizsgálata után szüksége lesz egy Miniconda[1] felhasználói környezetre, melyből a Python 3.9-es verzió szükséges a szoftver megfelelő működéséhez. Linux, Windows és MacOSX rendszerekre is elérhető a Miniconda disztribúció. Miután a telepítési instrukciókat követve meggyőződtünk arról, hogy helyesen települt a program, az *Anaconda Prompt* terminálban adjuk ki az alábbi parancsokat abban a könyvtárban[2], ahova a szakdolgozat programját telepíteni szeretnénk:

```
$ git clone git@github.com:MucsanyiBalint/FlexCoderGUI.git
$ cd FlexCoderGUI
$ conda env create --file flexcoder.yml
$ conda activate flexcoder
```

Amennyiben a felhasználó rendelkezik egy, a CUDA[3] technológiát támogató videokártyával, a megfelelő teljesítmény eléréséhez szükséges a CUDA illesztőprogramok telepítése a felhasználó által preferált operációs rendszerre. A lábjegyzetben található linken válasszuk ki a CUDA Toolkit 11.1.0 verzióját, majd állítsuk be a megfelelő operációs rendszert, illetve az architektúrát, majd Linux rendszer esetén a disztribúciót, Windows esetén pedig a verziószámot. Ezután a telepítési instrukciókat követve installálhatjuk a CUDA Toolkit imént specifikált verzióját. Ekkor a telepítés sikerességét az alábbi kóddal ellenőrízhetjük:

[1] https://docs.conda.io/en/latest/miniconda.html

[2] A továbbiakban <install_path>.

[3] https://developer.nvidia.com/cuda-toolkit-archive

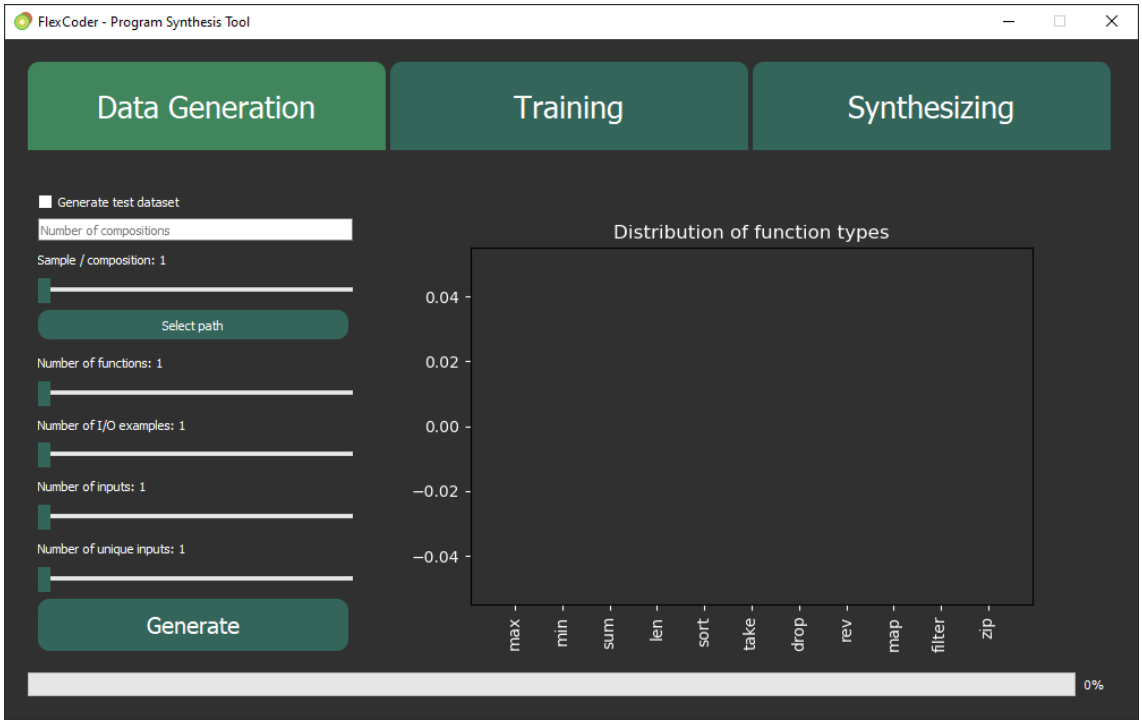
```
$ ipython
>>> import torch
>>> print(torch.cuda.is_available())
```

Amennyiben a parancs a *True* értéket írja ki a konzolra, a telepítés sikeres volt.

2.3. A program elindítása

A program elindítása az alábbi parancsok kiadásával lehetséges:

```
$ conda activate flexcoder
$ cd <install_path>/FlexCoderGUI
$ python flexcoder_gui.py
```



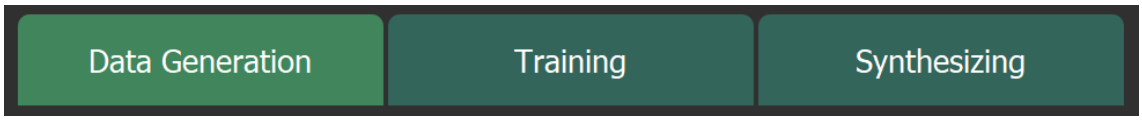
2.1. ábra. Az ábrán a szakdolgozat alapjául szolgáló szoftver látható, közvetlenül miután elindítjuk azt.

Amennyiben a program indítása sikeres, egy új ablak jelenik meg a képernyőn, melynek címe *FlexCoder - Program Synthesis Tool*. A grafikus felület az indítás után közvetlenül a 2.1 ábrán látható felépítéssel rendelkezik.

2.4. A program működése

A szoftver működésének taglalása a grafikus felület áttekintésével kezdődik, majd az egyes funkciók átfogó bemutatásával folytatódik.

2.4.1. A grafikus felület



2.2. ábra. A három gombból álló gombsor, melyekkel az applikáció lapjai között váltogathatunk.

Miután az előzőekben taglalt módon elindítjuk az applikációt, egy három egységre tagolt grafikus felülettel találjuk szemben magunkat. Az ablak tetején egy három gombból álló gombsáv jelenik meg, melyekkel a felület lapjai között váltogathatunk. Ez a 2.2 ábrán látható. Kezdetben az adatgenerálás lapja aktív, ugyanis mindenképp szükségünk van egy adathalmazra a további lapok használatához. Az aktív laphoz tartozó gomb élénkebb zöld színnel rendelkezik. Mindhárom lap esetén általánosan elmondható, hogy meglehetősen erőforrásigényes műveleteket foglalnak magukban, így a felületen nem engedélyezett egyszerre több lap funkcionalitásának futtatása. A grafikus felület mindhárom fülön reszponzív marad a hátérben futó folyamatok elindítása után is. Amint egy oldalon végeztünk a funkció használatával, a fenti gombsor ismét kattinthatóvá válik.

2.4.2. Adatgenerálás

Az adatgenerálás funkcióval lehetőségünk van tanító adathalmazt létrehozni a szoftver szerves részét képző neurális háló számára, amely a programszintézis folyamatát támogatja. A lap lehetőséget biztosít teszt adathalmaz generálására is, mellyel ellenőrizhetjük a programszintézis-rendszer teljesítményét. A teszteléshez használt adathalmazt a *Synthesizing* fülön használhatjuk fel, amely a *Programszintézis* alfejezetben kerül bemutatásra.

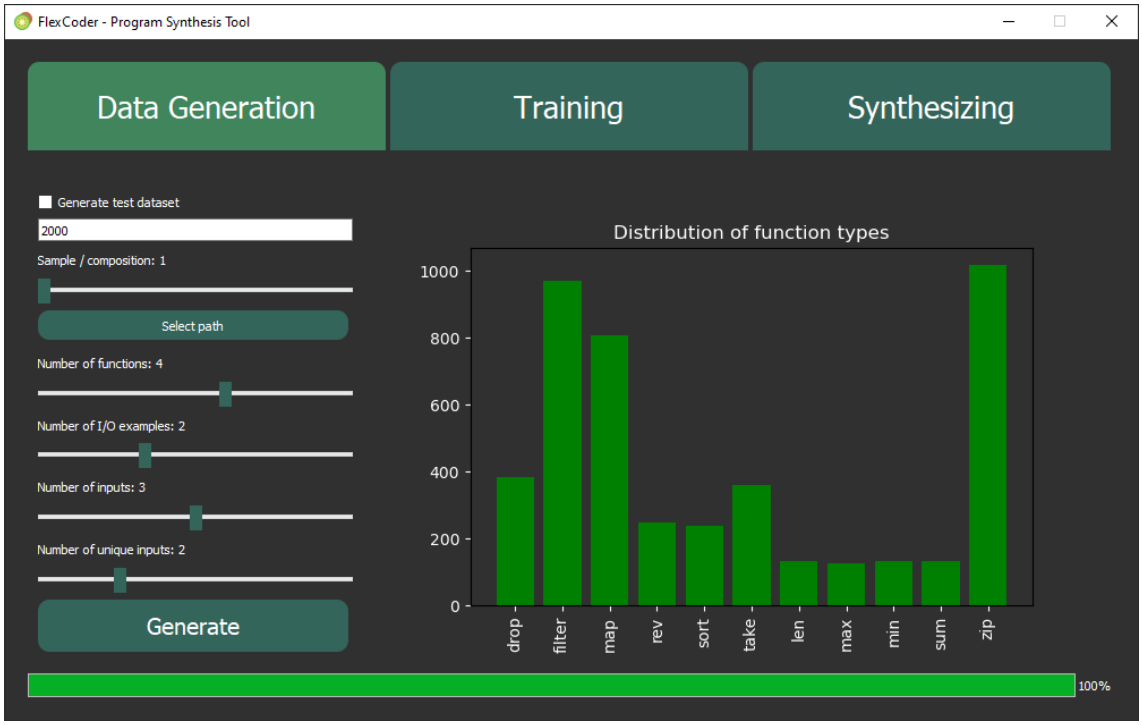
A lap bal oldalának felépítése:

- **Generate test dataset**: Ezen jelölőnégyzet aktiválásával generálhatunk teszt adathalmazt a rendszer teszteléséhez.
- **Number of compositions**: Beviteli mező, mellyel megadhatjuk a generálni kívánt függvénykompozíciók számát. A mező pozitív egész számokat fogad el, különben egy dialógusablak jelenik meg, ami tájékoztat a helytelen adatbevitelről.
- **Sample/composition**: Csúszka, mellyel azt szabályozhatjuk, hogy az egyes függvénykompozíciókból hány felparaméterezett kompozíciót állítson elő a szoftver, ahol a felparaméterezés a függvénykompozíció bemeneteinek behelyettesítését jelenti. Ezt az értéket 1-től 10-ig (inkluzív) állíthatjuk.
- **Select path**: Nyomógomb, amely a generálandó adathalmaz mentési útvonalának beállításáért felelős. Ehhez egy dialógusablak kerül megnyitásra, mellyel egy grafikus felületen, könyvtáraink között navigálva adhatjuk meg az útvonalat. Az adathalmazok kiterjesztése minden esetben *.DAT*, ez nem módosítható.
- **Number of functions**: Csúszka, ami lehetővé teszi a kompozíciót alkotó függvények számának megadását. Ez az érték 1 és 6 között (inkluzív) módosítható.
- **Number of I/O examples**: Csúszka, mellyel az egy kompozícióhoz tartozó bemeneti és kimeneti példák számát adhatjuk meg. A szintetizálási folyamat során egy oylan függvénykompozíció megadása a cél, amely az összes bemeneti példára végrehajtva a hozzájuk tartozó kimeneti példákat adja eredményül.
- **Number of inputs**: Csúszka, amellyel a függvénykompozíció bemeneteinek számát szabályozhatjuk. Ezt 1-től 5-ig (inkluzív) választhatjuk meg.
- **Number of unique inputs**: Csúszka, amely az egyes függvénykompozíciók egyedi bemenetei számának beállításáért felelős. A rendszer egy egyedi bemenetet a szintézis folyamata során többször is felhasználhat, ezért szükséges megkülönböztetni az egyedi bemenetek számát az összes bemenet számától. A paraméter értéke 1-től 5-ig (inkluzív) módosítható. Amennyiben az egyedi bemenetek számát nagyobbnak választjuk meg, mint a bemenetek számát (*Number of inputs*), akkor egy dialógusablak tájékoztat minket a helytelen paramétermegadásról.

- **Generate**: Nyomógomb, mellyel a a paraméterek megfelelő megadása után elindíthatjuk az adatgenerálási folyamatot. A folyamat elindulása után a gomb pirosra vált, és a *Stop* felirat jelenik meg rajta. Erre rákattintva leállíthatjuk a folyamatot. Ezzel minden addig generált példánk elvész.

A *Generate* gomb lenyomása után kezdetét veszi a generálási folyamat. Ennek előrehaladását a lap alján látható töltőcsík jelzi. Lehetőségünk van a folyamat befejezése előtt megszakítani azt a *Stop* gomb lenyomásával. Amennyiben hagytuk teljesen lefutni a generálást, a jobb oldalon látható oszlopdiagram megtelik a generálásból származó adatokkal. Ez a rendszer által támogatott egyes függvények előfordulásának számát foglalja magában a generált adathalmazban.

A generálást egymás után többször is lefuttathatjuk. Az oszlopdiagram minden esetben a legutóbbi, sikeresen végbement generálási folyamat eredményéül kapott adathalmaz statisztikáit tartalmazza. Amennyiben a generálást megszakítjuk, az oszlopdiagram nem frissül, ugyanis az addig generált adat elvész. A lap generálás utáni kinézetét az alábbi ábra mutatja be:



2.3. ábra. A *Data Generation* lap kinézete az adatgenerálás lefutása után. Az oszlopdiagram a legfrissebb eredményeket tartalmazza és automatikusan frissül.

2.4.3. A neurális háló tanítási folyamata

A neurális háló tanításának paraméterezésére és elindítására a *Training* lapon van lehetőségünk. Ehhez szükségünk van egy megfelelő formátumú adathalmazra, melyhez az imént taglalt *Data Generation* fülön férhetünk hozzá. Számos, a neuráls háló tanításánál felhasznált hiperparamétert beállíthatunk a lap bal oldalán.

A lap bal oldalának felépítése:

- **Select dataset**: Nyomógomb, amely a már legenerált tanító adathalmaz betöltéséért felelős. A gomb lenyomásakor egy dialógusablak nyílik meg, mellyel megadhatjuk a tanító adathalmaz útvonalát az operációs rendszer alapértelmezett fájlkezelőjének használatával. A kiválasztandó adathalmaz kiterjesztésének minden esetben *.DAT*-nak kell lennie, ez nem módosítható.

- **Patience of training**: Csúszka, mellyel a tanulási algoritmus türelmi intervallumát határozhatjuk meg. Ez az azon egymás utáni epoch-ok[4] maximális számát határozza meg a tanítás megállása előtt, melyekben a validációs költség nő. Ezt az értéket 1-től 10-ig (inkluzív) állíthatjuk.

- **Determine largest batch size**: Jelölőnégyzet, melynek kiválasztása esetén a tanítási folyamat elején lefuttatásra kerül egy algoritmus, amely meghatározza a legnagyobb kötegméretet a tanítópéldákhoz, ami még befér az aktuálisan elérhető VRAM-ba vagy RAM-ba. Amennyiben kiválasztjuk ezt a jelölőnégyzetet, a *Batch size* opció nem lesz elérhető. Az opció bejelölése ajánlott.

- **Batch size**: Csúszka, amely a tanítás során használt kötegméretet választja ki, azaz azt, hogy egyszerre hány példát adunk át a neurális hálónak a tanítási folyamatban. Ezen paraméter lényeges hatással lehet a tanítás konvergenciájára. Manuális meghatározása csak a gépi tanulásban jártas felhasználóknak javasolt. Minimális értéke 32, maximális értéke 1024. A lehetséges értékek kettőhatványok.

- **Number of CUDA GPUs**: Csúszka, mellyel kiválaszhatjuk, hogy a rendszerünkben elérhető CUDA-kompatibilis videokártyák közül mennyit szeretnénk

[4] A tanító adathalmaz teljes feldolgozása.

használni a tanításhoz. Az opció maximális értékét a szoftver a futtatási környezettől dinamikusan kéri le, így érvénytelen érték megadása nem lehetséges. A minimális szám 0, ebben az esetben CPU-n történik a tanítás.

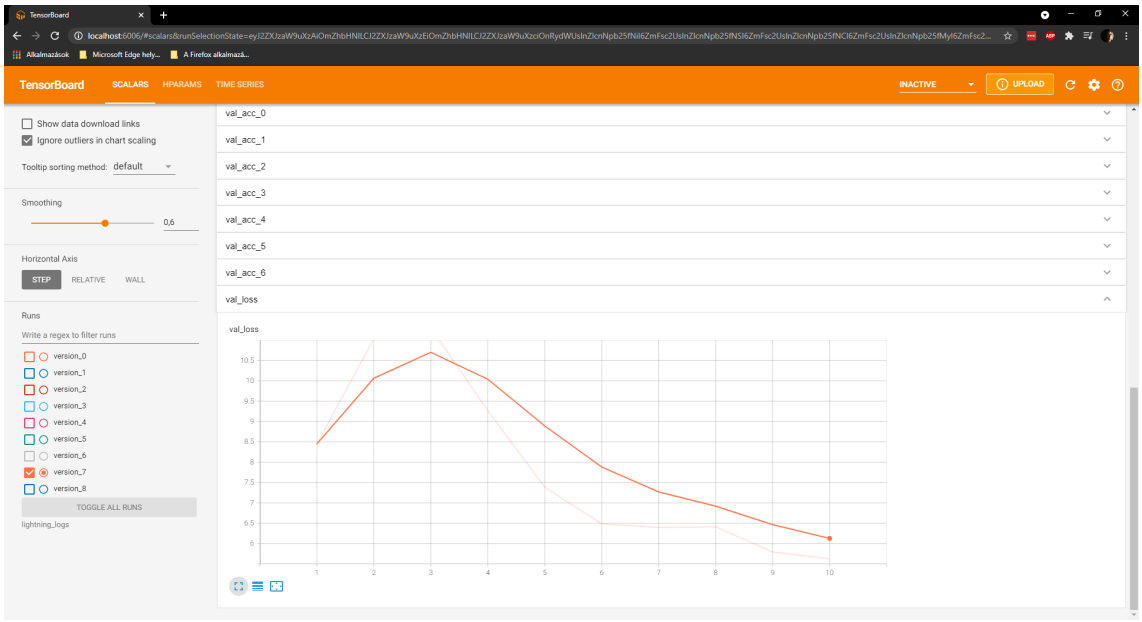
- **Maximum number of epochs**: Csúszka, amely lehetőséget ad a tanítás epoch-jaira egy felső korlát meghatározására. Az opció minimális értéke 5, maximális értéke pedig 100. A csúszka lépésköze 5.

- **Select path**: Nyomógomb, amely a tanítás során létrejövő modell checkpoint-ok mentési útvonalának kiválasztását teszi lehetővé.

- **Start training**: Nyomógomb, mellyel a a hiperparaméterek megadása után elindíthatjuk a tanítási folyamatot. A folyamat elindulása után a gomb pirosra vált, és a *Stop* felirat jelenik meg rajta. Erre rákattintva leállíthatjuk a folyamatot. Ezzel a tanítás során a leállításig létrehozott checkpoint-ok nem vesznek el, azokat később be lehet tölteni.

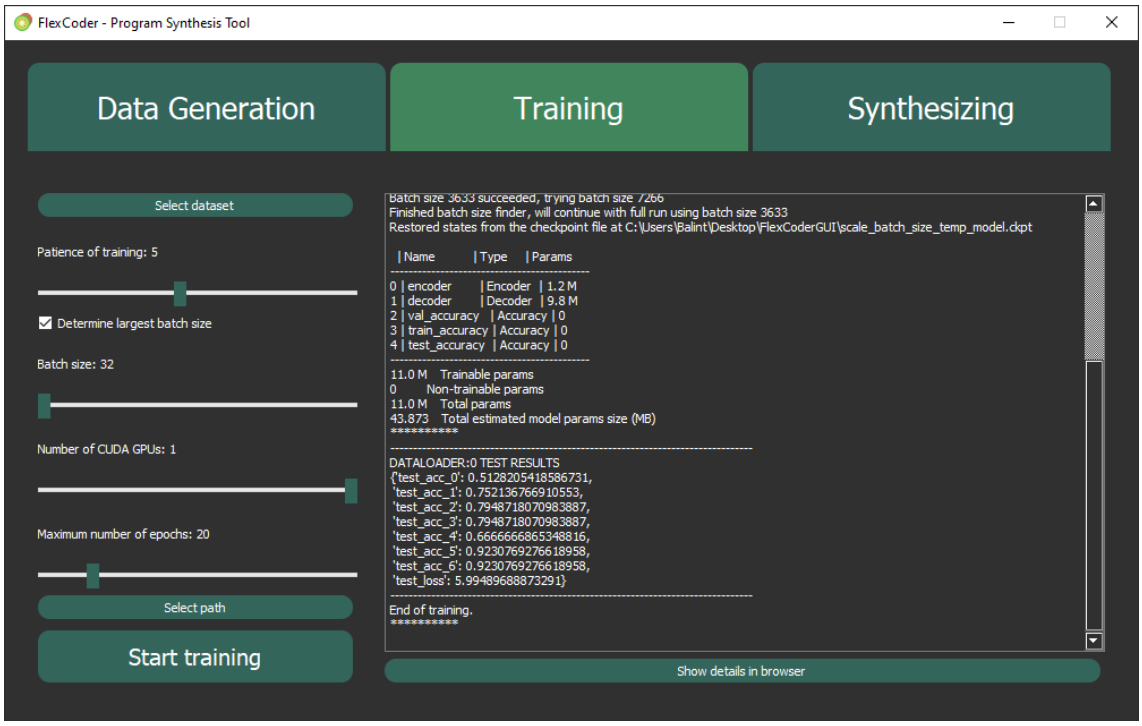
A *Start training* gomb lenyomása után elkezdődik a tanítási folyamat. Ennek állapotát az ablak jobb oldalán látható konzolablakban láthatjuk. Amennyiben megjelöltük a *Determine largest batch size* opciót, lehetőségünk van az algoritmus futását is nyomon követni a konzolban. A konzolban láthatunk egy rövid összefoglalót a tanítandó modellről, láthatjuk neuronjainak számát és becsült méretét megabájtban. A tanítási folyamat végén a tanítási adathalmazból automatikusan elszeparált teszthalmazon elért eredményét láthatjuk a neurális hálónak. Ezután a tanítás véget ér, a konzolon a *Finished training.* felirat jelenik meg.

A tanítást a generáláshoz hasonlóan szintén egymás után többször is lefuttathatjuk. A konzol megőrzni az előző tanítások eredményeit és logjait is, amennyiben összehasonlításhoz szükségünk lenne azokra. A tanítás során bármikor megnyomhatjuk a *Show details in browser* gombot, melynek hatására egy TensorBoard szerver indul el a háttérben, melyben a böngésző használatával nyomon követhetjük a tanulás folyamatát, a neurális háló egyes kimeneti fejeinek pontosságát, illetve az egyes loss értékeket. A megnyitott TensorBoard felület az alábbi ábrán látható:



2.4. ábra. A TensorBoard szoftver megjelenése a *Show details in browser* gombra való kattintás után. A modell egyes kimeneti fejeinek pontossága és loss értéke különkülön ábrákon megtekinthető, illetve a kumulált loss érték is logolásra kerül. Jobb oldalon választhatunk a neurális háló egyes verziói között, ha több betanított modell checkpoint-jával is rendelkezünk.

Amennyiben a tanítást megszakítjuk, a konzolon nem jelenik meg több információ. Ekkor az addig létrehozott checkpoint-ok továbbra is elérhetőek maradnak, így azokat felhasználhatjuk a későbbiekben. A felület tanítás utáni kinézetét az alábbi ábra mutatja be:



2.5. ábra. A *Training* fül megjelenése a tanítási folyamat végén. A konzolban a tanítás adatai és eredményei láthatók.

2.4.4. A neurális háló teljesítményének tesztelése

A neurális háló teljesítményének tesztelése kifejezetten fontos, ugyanis a *Training* fül hiperparamétereinek nem megfelelő beállítása könnyen egy gyengén teljesítő modellt eredményezhet. A *Training* fül konzoljában emiatt a tanítás végén megjelenik a neurális hálónak egy, a tanító adathalmazról automatikusan leválasztott teszt adathalmazon történő kiértékelése, melyben láthatjuk a kumulált loss értéket és az egyes outputfejek pontosságát a teszt adathalmazon. Fontos továbbá megjegyezni, hogy az előző alfejezetben bemutatott *TensorBoard* alkalmazásban részletesen nyomon tudjuk követni a tanítás során a szintén automatikusan leválasztott validációs adathalmazon mért validációs loss-t, illetve a kimeneti fejek pontosságát. Ezen adatok jó támpontot adhatnak, hogy a tanításhoz és generáláshoz használt hiperparamétereket milyen irányba módosítsuk.

2.4.5. Programszintézis

Kétségtelen, hogy a bemutatott program legfontosabb komponense a *Synthesizing* fül. Ezen ugyanis az általunk betanított neurális hálót felhasználhatjuk a programszintézis-rendszer komponenseként, és ezen rendszert függvénykompozíciók szintetizálására alkalmazhatjuk.

- **Select model**: Nyomógomb, mellyel a *Training* fülön betanított modellek közül választhatunk. A gomb lenyomásakor egy dialógusablak jelenik meg, melynek segítségével megadhatjuk a modell elérési útvonalát az operációs rendszer alapértelmezett fájlkezelőjének használatával. A kiválasztandó fájl kiterjesztésének minden esetben *.CKPT*-nak kell lennie, ez nem módosítható.

- **Select test dataset**: Nyomógomb, amely lehetőséget ad a teszt adathalmaz kiválasztására. Bár a modell kizárólag 1 bemeneti-kimeneti párt tartalmazó kompozíciókból álló adathalmazon tanítható, az képes a programszintézis-rendszer részeként több bemeneti-kimeneti példás feladatok megoldására is, így a teszt adathalmaz tartalmazhat ilyen példákat is. A gomb lenyomásakor egy dialógusablak nyílik meg, mellyel megadhatjuk a teszt adathalmaz útvonalát. A kiválasztandó adathalmaz kiterjesztésének minden esetben *.DAT*-nak kell lennie, ez nem módosítható. Fontos, hogy a teszt adathalmaz megadása nem kötelező! Amennyiben manuálisan szeretnénk megadni egy egzakt problémát a rendszernek, használhatjuk az *Input(s)* és *Output(s)* feliratú beviteli mezőket is.

- **Select save path**: Nyomógomb, mellyel megadhatjuk a feladatokat megoldó kompozíciók vizualizációjának mentési útvonalát. A felugró dialógusablakban létre is hozhatunk új könyvtárat.

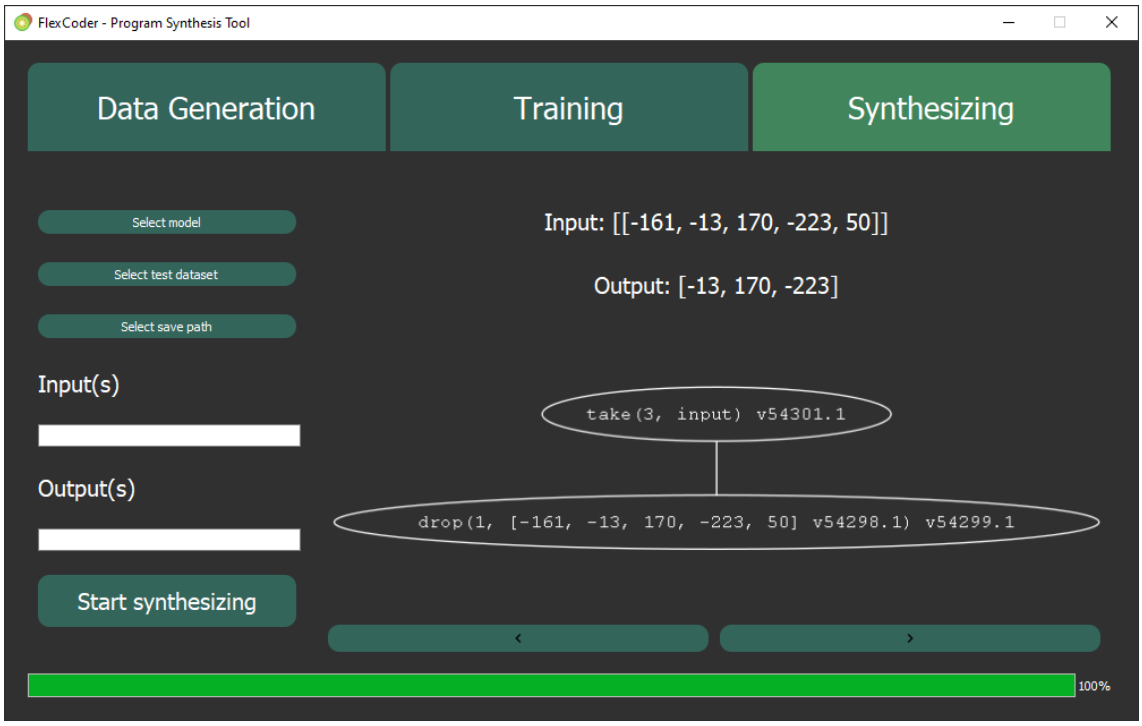
- **Input(s)**: Szövegmező, melyben egy egzakt programszintézis-probléma bemenetét adhatjuk meg. A probléma bemeneteit egy Python tuple-ben, azaz zárójelek között kell megadni, vesszőkkel elválasztva. Az egyes bemenetek egész értékek listái, vagy listák listái lehetnek, azonban ennek minden bemenet esetén egységesnek kell lennie. Az input nem tartalmazhat üres listát, ugyanis erre minden, a rendszer által támogatott transzformáció üres listát adna ered-

ményül, így a megoldás ez esetben triviális. A szövegmezőre vonatkozó megszorítások ellenőrzésre kerülnek az adatok megadásakor, a szoftver egy felugró dialógusablakkal figyelmeztet, amennyiben helytelen formátumú adat került megadásra.

- **Output(s)**: Szövegmező, melyben egy egzakt programszintézis-probléma kimenetét adhatjuk meg. A probléma kimenete egész értékek listája, vagy listák listája lehet. Fontos, hogy a bemeneti-kimeneti párok száma megegyezzen az *Input(s)* és az *Output(s)* szövegmezőben.

- **Start synthesizing**: Nyomógomb, mellyel a a modell, a probléma és a mentési útvonal megadása után elindíthatjuk. A folyamat elindulása után a gomb pirosra vált, és a *Stop* felirat jelenik meg rajta. Erre rákattintva leállíthatjuk a szintetizálási folyamatot. A szintetizálási folyamat leállítása nem törli ki az addig már sikeresen megoldott problémákra megadott függvénykompozíciókat, azok a képmegjelenítőben megtekinthetők.

A szintetizálási folyamat előrehaladását a fül alján látható töltőcsík jelzi. A felület jobb oldalán kezdetben fentről lefelé az *Awaiting inputs...*, azaz a *Várakozás a bemenetek megadására...*, az *Awaiting outputs...*, azaz a *Várakozás a kimenetek megadására...* és az *Awaiting results...*, azaz a *Várakozás az eredményekre...* feliratok olvashatók, illetve két nem kattintható gomb is látható a szöveg alatt. A szintetizálási folyamat befejeztével ezen feliratok helyén egy képmegjelenítő panel jelenik meg, amin a feladatok és az azokat megoldó szintetizált függvénykompozíciók láthatók. Amennyiben több, mint 1 problémát adtunk meg, a képmegjelenítő alatti nyomógombokkal tudunk váltogatni az egyes feladatok között. Erre példa az alábbi ábrán látható:



2.6. ábra. A *Synthesizing* fül megjelenése a szintézis folyamatának végén. A képmegjelenítő blokkban a programszintézis eredményéül szolgáló kompozíciók láthatók. Amennyiben egy feladatra nem talál megoldást a rendszer, kép helyett a *No solution for the given problem.*, azaz a *Nincs megoldás a megadott problémára.* felirat jelenik meg.

Az előző két funkcióhoz hasonlóan a szintetizálás is tetszőlegesen sokszor futtatható. A beállítások bármikor módosíthatóak, azonban a megadott paraméterek csak a *Start synthesizing* gombra rányomva lépnek érvénybe.

2.4.6. Fájlszerkezet

A felhasználók szemszögéből a program teljes mértékben rugalmas, ugyanis az általuk generált adatfájlokat, modelleket, illetve függvénykompozíció-vizualizációkat nem egy előre meghatározott helyre menti, hanem a felhasználónak lehetősége van minden elérési útvonalat közvetlenül megadni dialógusablakok segítségével. Az elmentett modellek esetén azonban a felhasználó csak az azokat tartalmazandó könyvtár útvonalát adja meg, az egyes checkpoint-ok nevét nem, ugyanis azt a PyTorch Lightning csomag automatikusan, jól átláthatóan kezeli. Az elmentett modellek fájlnevei *epoch=x-step=y.ckpt* alakúak, ahol x jelöli az aktuális epoch-számot, amiben

mentésre került a modell, y pedig az aktuális adatköteg indexét, amelyet a modell legutoljára feldolgozott. Egyetlen tanítási folyamat garantáltan egyetlen checkpointot ment le, méghozzá azt, melynél a validációs pontosság maximális volt a tanított modellnél.

A szoftver kezel *temporális* fájlokat is, melyek a belső működéshez szükségesek, és csak a program futása során léteznek, így csak a szemfüles felhasználó találkozhat velük, aki engedélyezte a rejtett fájlok megtekintését:

- **<install_path>/.sample.dat**: A szintetizálási folyamat egy tesztadathalmazt vár egy fájl formájában. Emiatt ha a felhasználó a programon belül ad meg a szövegdobozban egy programszintézis-problémát, úgy a szövegdoboz tartalma egy validációs eljárás után az *<install_path>/.sample.dat* rejtett fájlba íródik, amely a programszintézis folyamata után törlésre kerül.

- **<install_path>/.statistics.json**: Az adatgenerálási folyamat végén megjelenítésre kerül a támogatott függvények eloszlása a legutóbb generált adathalmazban. Ez a *<install_path>/.statistics.json* rejtett fájlba íródik, melyet a program felhasznál az oszlopdiagram megjelenítésére, majd törli azt.

- **<install_path>/.images.json**: A szintetizálási folyamat végén a megadott problémák az ezeket megoldó kompozíciók elérési útvonalaival együtt az *<install_path>/.statistics.json* fájlba íródnak, amely a képmegjelenítő által kerül beolvasásra, majd törlésre.

A *.DAT* és a *.CKPT* kiterjesztésű fájlok szabadon törölhetőek a felhasználó által, ugyanis ezek a fent említett eseten kívül közvetlenül általa kerülnek létrehozásra. **Azonban minden más kiterjesztésű fájl törlése az *<install_path>* könyvtárból hibákhoz, a szoftver instabilitásához és nemdefiniált viselkedéshez vezethet!**

2.4.7. Hibakezelés

A bemutatott szoftver számos paraméterrel rendelkezik, melyek befolyásolhatják a példagenerálást, a tanulási folyamatot és a programszintézist egyaránt. A program meglehetősen nagy komplexitása miatt könnyű hibát véteni az egyes folyamatok

felparaméterezésében, amely katasztrofális eredményekkel járna, ha nem lennének megfelelően ellenőrizve és kezelve az ilyen kivételes események.

A felhasználótól kapott adatok körültekintően ellenőrzésre kerülnek a hibák elkerülése végett. Ehhez a szoftver két fő stratégiát alkalmaz:

1. A felhasználót meggátolja a helytelen adatok bevitelében, illetve
2. a felhasználó által bevitt helytelen adatra, vagy egyes adatok hiányára dialógusablakkal figyelmeztet, a kért parancsot pedig nem hajtja végre.

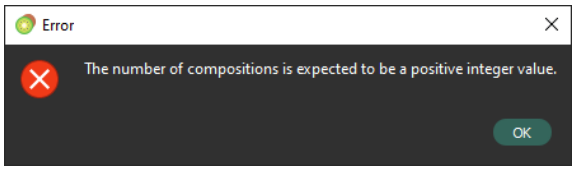
Mivel az első pont esetén a helytelen adat bevitele nem lehetséges, itt nem kell hibakezelésnek sem történnie. A második pont esetén azonban szükséges a bevitt adat validálása minden esetben, lefedve a sarkalatos eseteket is. A program által megjelenített, a figyelmet hibákra felhívó dialógusablakok lehetséges üzenetei, illetve az ezekhez fűzött rövid magyarázatok az alábbi listában olvashatók:

- **The number of compositions is expected to be a positive integer value.** Az adatgeneráló lapon a generálandó kompozíciók számának megadására szolgáló szövegmezőbe nem egy pozitív egész szám került beírásra. A példák száma nem lehet sem nempozitív, sem az egész számok halmazán kívül eső szám.
- **No filename provided. Please select the desired path.** Az adatgeneráló lapon nem került kiválasztásra a generálandó adatfájl elérési útvonala, illetve a fájl neve. Ez a *Select path* gomb megnyomásával lehetséges.
- **Number of unique inputs cannot be larger than the number of inputs.** A generálandó függvénykompozíciók esetén a *Number of inputs* feliratú csúszkával tudjuk megadni a bemenetek számát. Ezek azonban nem feltétlenül egyedi bemenetek, ugyanis a rendszer egy inputot többször is képes felhasználni. A *Number of unique inputs* feliratú csúszka teszi lehetővé az egyedi bemenetek számának egzakt specifikálását, amely így nem lehet több, mint a bemenetek összesített száma.
- **No dataset provided. Please select the desired path.** A tanítás lapján a felhasználó nem választott ki adathalmazt, melyen a neurális hálót tanítani szeretné. Ez a *Select dataset* feliratú gombbal tehető meg.

- **No save path provided.** A felhasználó nem jelölt ki mentési útvonalat a tanított modellnek. Ez a *Select path* gombbal tehető meg.
- **Please provide the inputs, too.** A *Synthesizing* lapon a kimenetek szövegmezője nem üres, azonban a bemeneteké igen. Ha a felhasználó a szövegmezőt használja egy példa megadásához, akkor az felülírja a kiválasztott fájlt, ha van ilyen. Ekkor tehát szükséges mind a bemenet, mind az elvárt kimenet biztosítása a problémához.
- **Please provide the outputs, too.** Hasonlóan az előző hibához a *Synthesizing* lapon a bemenetek szövegmezője nem üres, azonban a kimeneteké igen.
- **Please provide a dataset or an exact problem.** A felhasználó sem teszt adathalmazt nem adott meg a *Select test dataset* feliratú gomb megnyomásával, sem a szövegmezőket nem vette igénybe egy egzakt példa megadásához. Ezek közül az szükséges a szintetizáláshoz, ugyanis különben nincsenek bemenet-kimenet párok specifikálva, amikhez programot kéne szintetizálni.
- **Please provide a model checkpoint.** A felhasználó nem adott meg elérési útvonalat a szintetizálási folyamatban felhasználandó neurális hálóhoz. Ez a *Select model* feliratú gombbal tehető meg.
- **Please provide the save path of the synthesized compositions.** A felhasználó nem adott meg elérési útvonalat a szintetizált kompozíciók képeit tartalmazandó könyvtárhoz. Ez a *Select save path* feliratú gombbal tehető meg.
- **Invalid input state tuple provided.** A felhasználó a szövegdobozt választotta egy programszintézis-probléma megadásához, azonban az általa megadott bemenet nem megfelelő. Ennek elvárt formátuma a 2.4.5 alfejezetben kerül bemutatásra.
- **Empty inputs are not allowed.** A felhasználó vagy egy üres listát adott meg, vagy olyan listák listáját, melyek között van üres. Egy üres listát bemenetként megadva a nyelvtanunkból levezethető összes függvénykompozíció üres listát ad eredményül, így a feladat triviális.

- **Remove unnecessary parentheses from input.** A bemeneti szövegdobozban megadott bemeneten felesleges zárójelek találhatók az elvárt formátumhoz képest.
- **Mismatching number of I/O examples provided.** Vagy a bemenetben, vagy a kimenetben, vagy ezek között az egyes példákhoz tartozó bemeneti-kimeneti minták száma nem egyezik meg.
- **Invalid output state tuple provided.** Hasonlóan az *Invalid input state tuple provided.* hibához, a felhasználó által a szövegdobozban megadott kimenet formátuma nem megfelelő.
- **Cases where all of the outputs are empty are not allowed.** Az egyetlen üres kimeneti lista, vagy az olyan kimeneti lista esete, amely csupa üres listát tartalmaz, szintén triviális problémák, ugyanis megfelelő számú *drop* függvény alkalmazásával minden bemenet üressé tehető.
- **Remove unnecessary parentheses from output.** Hasonlóan a bemeneti szövegdoboz esetéhez, a megadott kimeneten felesleges zárójelek találhatók.

A hiba keletkezésekor felugró dialógusablakok az alábbi egységes megjelenéssel rendelkeznek:



2.7. ábra. Az egyes hibák esetén felugró dialógusablakok egységes megjelenése.

3. fejezet

Fejlesztői dokumentáció

A bemutatott szoftver képes egy példaalapú programszintézis-rendszerben felhasznált neurális háló tanításához megfelelő minőségű példák generálására, a neurális háló ezeken történő betanítására, majd a teljes programszintézis-rendszer alkalmazására a felhasználó által manuálisan megadott, vagy a grafikus felületen generált problémákon. A rendszer által megoldott problémákra adott megoldásokat egy képmegjelenítő komponensben lehet megtekinteni.

3.1. Célkitűzés

A szoftver tervezésének kezdeti fázisában nagy hangsúlyt fektettem a backend komponensekre. Egy korábbi, Gyarmathy Bálinttal és Czapp Ádámmal közösen fejlesztett programszintézis-rendszernek hátránya volt, hogy nyelvtana, és ez által a szintetizálható függvénykompozíciók is szekvenciálisan épültek fel. Ennek oka az volt, hogy nem definiáltunk olyan függvényt, amely egynél több listát várt volna paraméteréül. Célkitűzésem, és témaválasztásom alapja ennek a limitációnak a feloldása volt, melyet a témabejelentőmben is részletesen taglaltam.

3.2. Ütemterv

A célok meghatározása után először egy ütemtervet dolgoztam ki témavezetőmmel, Dr. Pintér Balázzsal közösen. Ez a jelentősebb fejlesztési állomásokat, illetve az

ezekhez kapcsolódó határidőket tartalmazta. Ennek egy leegyszerűsített verziója az alábbi lista:

1. GitHub repository létrehozása, licenszek megadása, láthatósági szint korlátozása
2. A progrmaszintézis-rendszer környezetfüggetlen nyelvtanának kidolgozása
3. A példagenerálás megalkotása, felparaméterezése
4. A generált függvénykompozíciók optimalizálása a neurális háló taníthatóságának segítése érdekében
5. A neurális háló implementálása, egyes hiperparaméterekkel történő kisérletezés
6. A már elkészült nyalábkereső algoritmus integrálása a rendszerbe
7. Egy, a háló teljesítményét kiértékelő szkript megírása
8. A grafikus felület megalkotása, a programszintézis-rendszer összekötése a felülettel

A megírt forráskód tesztelése nem került a listára, ugyanis azt a program tervezésének minden lépésben elvégeztem automatikus egységtesztek írásával és azoknak a continuous integration (CI) pipeline-ba történő integrálásával.

Az egyes mérföldkövek a megadott sorrendben lettek elérve. Nem volt lehetőség a sorrend módosítására, ugyanis az egyes listabeli pontok előfeltételei az előttük levő állomások. Az egyes pontokban azonban számos olyan, addig nem észrevett javítási lehetőség merült fel, melyek segítségével egy sokkal egzaktabb tanítási feladatot tudtam létrehozni a neurális háló számára.

3.3. Követelmények

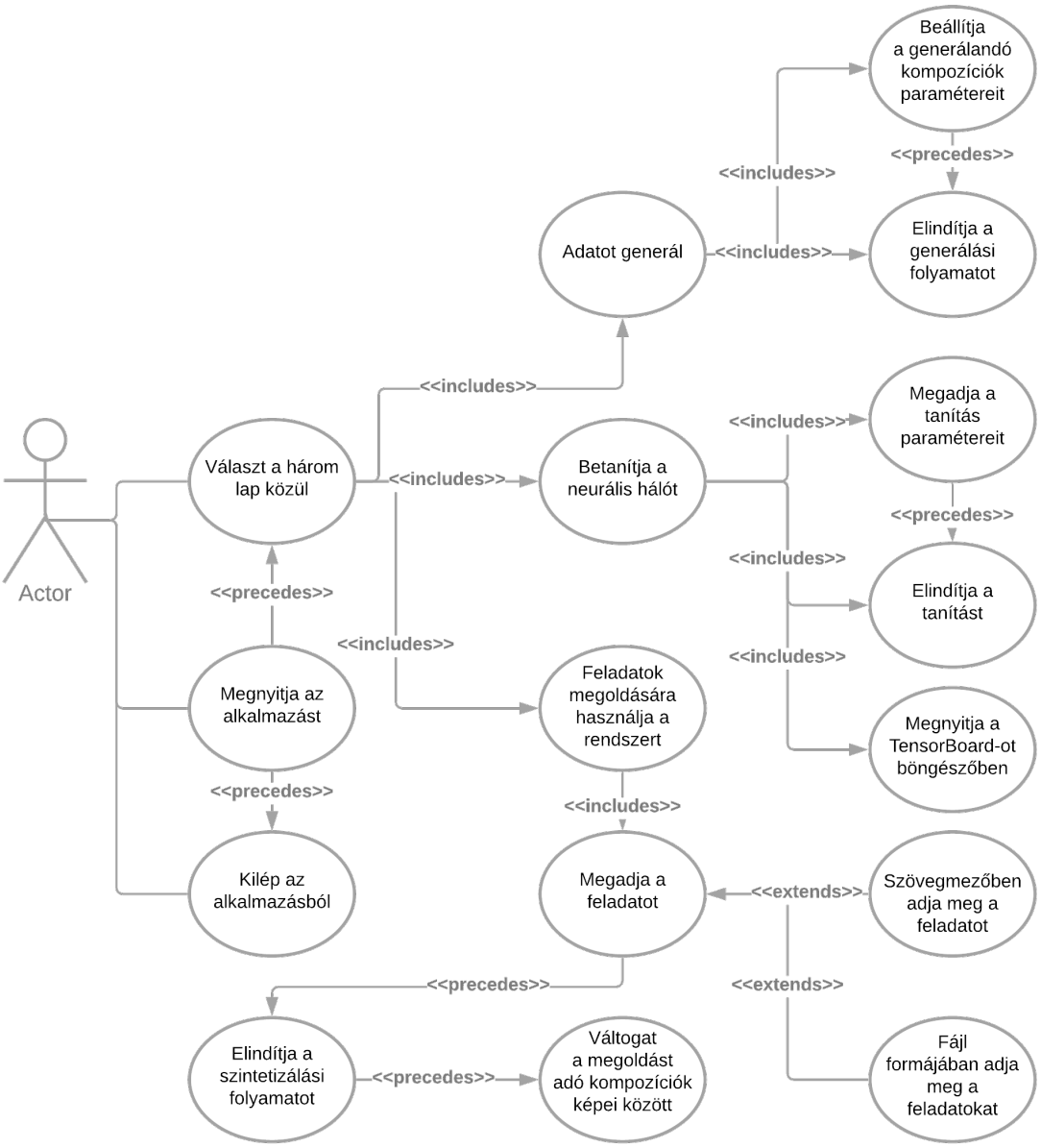
A szoftverrel szemben elvárás volt, hogy úgy terjessze ki a Gyarmathy et al. [11] cikkben bemutatott rendszert, hogy annak kifejezőerejét jelentősen növelje, azonban teljesítménye ne romoljon az eredeti példákon sem. Fontos részét képezte továbbá a projektnek egy olyan grafikus felület létrehozása, amely jelentősen megkönnyíti az

adatgenerálási, tanítási és szintetizálási folyamatok futtatását, de nem igényel jelentős mértékű erőforrást, azaz nem lassítja ezen folyamatokat. Mivel a szoftver jelentős számú hiperparaméterrel rendelkezik, fontos volt az ezek egyszerű megadhatósága is, ugyanis egy konzolos alkalmazást használva többsoros parancsokat kell megadnia a felhasználónak, melyekben nagyon egyszerű hibát véteni is. Ezektől körültekintő validációval véd a grafikus felület, az egyszerű adatbevitelt pedig nyomógombokkal, szövegmezőkkel és csúszkákal teszi lehetővé.

Hasonlóan hangsúlyos követelmény volt az egyes folyamatok nyomon követése is, ugyanis például a neurális háló betanítása akár több órát is igénybe vehet, mialatt kiemelten fontos a háló pontosságának, illetve loss értékeinek vizsgálata.

3.4. Használati eset diagram

Az alábbi ábrán a szoftver használati eset diagramja látható:



3.1. ábra. A szoftver használati eset diagramja.

3.5. Megoldási terv

A szoftver a *Python* [12] programozási nyelven került implementálásra. A *Natural Language Toolkit* [13] programcsomag használatával implementáltam a függvénykompozíciók megadásához használt környezetfüggetlen nyelvtant. A *Graphviz* [14] szoftver, és az ennek illesztőprogramjául szolgáló *PyDot* csomag segítségével vizualizáltam a létrehozott kompozíciókat. A *PyTorch* [15] és *PyTorch Lightning* [16] csomagokat használtam a neurális háló és az adat pipeline létrehozásához, majd a

háló tanításához is. Azért választottam ezeket a csomagokat, mert az automatikus gradiensszámítással és egyéb kényelmi funkcióikkal jelentősen megkönnyítik egy neurális háló implementálását, illetve egy praktikusan használható API-t nyújtanak. A *Sphinx* programmal készítettem automatikus dokumentációt az osztályok és függvények docstring-jeiből, amely automatikusan frissül a forráskód módosításával. A *Hypothesis* és *Pytest* csomagok tették lehetővé a fejlesztés alatt lévő szoftver átfogó tesztelését. A *Mypy* csomag a típusannotációk vizsgálatáért volt felelős, ami segített feltérképezni az előforduló hibákat. A *PyQt5* csomagot használtam teljes mértékben a grafikus felület megalkotásához, ugyanis véleményem szerint ez a grafikus keretrendszer adja a legtöbb személyreszabási lehetőséget.

A szoftvert a *PyCharm Community Edition* és a *Doom Emacs* fejlesztői környezetekben írtam meg, a programot konzolban futtattam.

3.6. Felhasznált technológiák

3.6.1. PyTorch

A PyTorch [15] egy flexibilis, nyílt forráskódú keretrendszer deep learning algoritmusok implementásához. A szoftvercsomag javarészt a 2007-ben Ronan Collobert és társai által nyilvánosságra hozott Torch7 projektre épül [17], amely a Lua programozási nyelvhez készült és javarészt C-ben íródott. A csomag könnyű hibakeresést biztosít, ugyanis csak a számításigényes kódrészletek futnak a hatékony és gyors C++ backend-en, minden mást a felhasználó adhat meg Python-ban. A PyTorch a NumPy [18] csomaghoz hasonlóan biztosít egy többdimenziós tömb adatszerkezetet, melynek neve torch.Tensor. Erre értelmezve vannak a gyakran használt, lineáris algebrából, statisztikából és matematikai analízisből ismert függvények, csakúgy, mint NumPy-ban.

Azonban a PyTorch automatikus gradiensszámítást is támogat a legtöbb gépi tanulásban használatos műveletre, ami jelentősen megkönnyíti a neurális hálók implementálását. Ennek miértjének tisztázásához szükséges a gépi tanulási módszerek rövid ismertetése.

Felügyelt tanulás

A felügyelt tanulás a gépi tanulás egyik ága. Ebben az esetben adott egy minta, amely (bemenet, címke) = (x, y) rendezett párokból áll. (Például a bemenet lehet egy kép egy emberről, a címke pedig az adott ember életkora.) A feladat egy $h(x)$ hipotézisfüggvény tanulása, amely elég jól közelíti adott mintabeli elemek inputjára az elemek címkéjét: $h(x) = \hat{y} \approx y$. A $h(x)$ függvény θ paramétereinek megtanulása után felhasználhajuk azt korábban nem látott, címkézetlen adatok címkéinek becslésére is.

Jelölés:

Tanítóminta (training set): $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(M)}, y^{(M)})\}$, ahol $x \in X$, $y \in Y$.

- **M** a mintaelemek száma,
- **X** a bemenetek halmaza,
- **Y** pedig a címkehalmaz.

A jelölés esetlegesen megtévesztő lehet: x és y nem ismeretlenek, hiszen ezek előre ismert konstansok: (bemenet, címke) párok.

Az ilyen felépítésű adathalmazokat általában három részre szokták bontani: tanító, validációs és tesztelési adathalmazra.

A hipotézisfüggvény paraméterei kizárólag a tanító adathalmazon kerülnek optimalizálásra, hogy minél jobban illeszkedjenek arra.

A validációs halmazon vizsgálhatjuk, hogy az így megtanult hipotézisfüggvény mennyire jól általánosít olyan adatpontokra, melyeken nem lett közvetlenül optimalizálva. A sikeres tanításhoz kifejezetten fontos, hogy a három adathalmaz eloszlása minimálisan térjen csak el, ugyanis ellenkező esetben könnyen lehet, hogy egy lényegesen eltérő problémán teszteljük, illetve validáljuk a hipotézisfüggvényünk teljesítményét, mint amin azt optimalizáljuk, azaz tanítjuk. A validációs adathalmaz felhasználható továbbá a hipotézisfüggvény egyes hiperparamétereivel történő kísérletezéshez is, melyről később fogok bővebben írni.

A tesztelési adathalmaz egyedüli létjogosultsága, hogy a hipotézisfüggvényünk paramétereinek optimalizálása és hiperparamétereinek körültekintő megválasztása után egy addig még nem felhasznált (azaz az a paramétereket és hiperparamétere-

ket semmilyen mértékben nem befolyásoló) adathalmazon győződjünk meg a tanult függvény predikcióinak minőségéről.

A felügyelt tanulásnak két típusa van: regresszió és klasszifikáció. A dolgozat keretében bemutatott rendszerben csak a klasszifikáció feladata jelenik meg, így ez a fogalom kerül csak bemutatásra.

Klasszifikáció

A klasszifikáció (osztályozás) a felügyelt tanulás egyik típusa. Ebben az esetben a címkehalmaz diszkrét, azaz $|Y| < \infty$. Példa erre egy olyan (bemenet, címke) rendezett pár, melyben a bemenet egy kép, a címke pedig: $Y = \{kutya, macska\}$. A klasszifikáció feladata ez esetben az, hogy a hipotézisfüggvény helyesen eldöntse, hogy egy kutya, vagy pedig egy macska látható a képen. Ha a címkehalmaz kételemű, a klasszifikációt *bináris klasszifikációnak* nevezzük.

Költségfüggvény, optimalizáció

Elérkeztünk ahhoz az alfejezethez, melyben magyarázatot kaphatunk az automatikus gradienssszámítás fontosságára. Ahhoz, hogy az előző fejezetekben bemutatott hipotézisfüggvényt megfelelően ráillesszük az adathalmazunkra, szükséges annak kezdetben kis abszolútértékű random θ paramétereinek megfelelő irányú és mértékű módosítása. Ehhez természetesen szintén szükséges egy érték (legtöbbször skalár), amely megadja, hogy a függvényünk egy adott állapotában mennyire jól illeszkedik a megadott tanító vagy validációs adathalmazra. Ennek a szerepét töltik be a gépi tanulási módszerekben a *költségfüggvények*, melyek a jobb modellekre alacsonyabb értéket adnak. Ezt az értéket röviden *költségnek* nevezzük. Az előbb említett tanítást leggyakrabban a *gradient descent* optimalizációs algoritmussal, vagy annak egy továbbfejlesztett variánsával végzik el iteratív módon. A *gradient descent* algoritmus és az egyes változatai a *backpropagation* algoritmusra épülnek, amely a hipotézisfüggvény egyes paramétereinek parciális deriváltját adja meg a költség függvényében. A *gradient descent* kivonja ezeknek az értékeknek egy skalárszorosát az egyes paraméterekből, ugyanis könnyen belátható, hogy ezzel a költségfüggvény (sokszor multidimenzionális) felületén lefele haladunk. A módszer a konvergencia egy általunk definiált mértékéig iteratívan végrehajtható. A *tanulási ráta* (pozitív

skalár) hiperparaméterrel adhatjuk meg, hogy a gradiensek hányszorosát vonjuk ki a hozzájuk tartozó paraméterekből. Ezzel a tanulás gyorsaságát és stabilitását szabályozhatjuk.

Az algoritmus további variánsai nem változtatnak a módszer fundamentális működésén, csupán finomítják az egyes paraméterek értékeinek frissítését, hogy minél gyorsabban konvergáljunk.

Természetesen az előbb említett parciális deriváltak kiszámítása nagyon nehéz feladatnak bizonyulhat, különösen egy meglehetősen bonyolult hipotézisfüggvény (vagy modell) esetén. Látható tehát, hogy nagy segítséget nyújt a PyTorch csomag a műveletek nyomon követésével, és a parciális deriváltak automatikus számításával.

Rétegek, aktivációs függvények

A neurális hálók legkisebb számítási egységei a neuronok. Egy neuronhoz súlyok, illetve egy bias érték is tartozik: ezek olyan skalárok, melyeket a tanítás során az optimalizációs algoritmus oly módon módosít, hogy a háló a megadott feladatot minél nagyobb pontossággal meg tudja oldani.

Az előbb említett neuronokat szokás rétegekbe szervezni: ezek lehetnek bemeneti, rejtett, illetve kimeneti rétegek. A bemeneti rétegek közvetlenül a teljes neurális háló inputját kapják bemenetként, majd ezen elvégzik a súlyaik által meghatározott transzformációt. A rejtett rétegek vagy a bemeneti réteg, vagy egy másik rejtett réteg kimenetét kapják meg, majd ezt transzformálják. A kimeneti rétegek abban különböznek a rejtett rétegektől, hogy kimeneteik képzik a teljes neurális háló kimenetét. Az egyes rétegek végére minden esetben szokás valamilyen aktivációs függvényt alkalmazni. Ezek olyan nemlinearitások, melyek növelik a háló kifejezőerejét, illetve megszorítást adhatnak egy neuron kimenetének intervallumára is. Köztudott, hogy két lineáris transzformáció kompozíciója egy új lineáris transzformációt jelent, így amennyiben az egyes lineáris transzformációk után nem teszünk aktivációs függvényt, úgy a rétegek halmozásával nem növeljük a háló komplexitását, ugyanis az így nyert architektúra ekvivalens egy egyetlen rétegből álló neurális hálóval.

A PyTorch számos gyakran használt réteget biztosít a felhasználóinak, mellyel jelentősen gyorsítja a szoftverfejlesztést, ugyanis nem kell minden modell implementációját az egyes neuronok működésének definiálásával kezdenünk.

3.6.2. PyTorch Lightning

A PyTorch Lightning [16] egy magasszintű interfész a PyTorch keretrendszerhez, amely egy modell betanításának számos komponensét absztrahálja. Az egyik absztrakciós eszköze a *pytorch_lightning.LightningModule*, melynek segítségével egyetlen Python osztályban összefoglalhatjuk a modellünk architektúráját, a felhasználandó adathalmazt, az adathalmaz betöltési algoritmusát, az optimalizáló algoritmust, illetve a tanítási ciklust. A szoftvercsomag nagyon fiatal: 2019-ben került kiadásra William Falcon által, aki a NYU PhD kutatója. A csomag PyTorch-hoz való illeszkedése hasonló a Keras és a TensorFlow között vonható párhuzamhoz.

A PyTorch Lightning több olyan kényelmi funkcióval is rendelkezik, ami jelentősen megkönnyítheti a tanítást. Ilyen például az EarlyStopping callback, amely a tanítás során végig vizsgálja a validációs halmazon számolt költséget, és megállítja a tanulást, ha az nőni kezd. Jogosan vetülhet fel a kérdés, hogy ez hogy fordulhat elő, ha a tanító adathalmazon éppen arra optimalizálunk, hogy a költség minél kisebb legyen. Két esetet különböztethetünk meg.

Az első esetben mind a tanítási költség, mind a validációs költség nő a tanítás során. Ez leggyakrabban akkor fordul elő, ha túl nagy tanulási rátát határoztunk meg, így bár jó irányba indulunk el a gradienssel, túlmegyünk a függvényfelület csökkenő szakaszán és egy magasabb költségű ponthoz érkezünk meg. Hasonlóan divergenciát tapasztalhatunk, ha a tanítóminták és a hozzájuk tartozó címkék nem konzisztensek a tanítás során.

A második esetben csak a validációs költség nő, a tanítási költség megfelelően csökken. Ez leggyakrabban a túltanulás jele: a modell túlságosan rásimult a tanító adathalmazra, olyan sajátosságokat is megtanulva, amelyek esetlegesen kizárólag a tanító adathalmazra jellemzőek és nem általánosíthatóak. Előfordulhat azonban az is, hogy a tanítási halmaz és a validációs halmaz eloszlása lényegesen eltér, azaz más adathalmazra optimalizálunk, mint amin figyeljük a költség értékét.

3.6.3. PyQt5

A PyQt5 grafikus keretrendszer a Riverbank Computing által fejlesztett csomagolómodul a jól ismert, C++ nyelven írt Qt keretrendszer körül. [19] A PyQt5 a Qt

szinte összes funkcióját elérhetővé teszi Python környezetben, amely egy említésre méltóan flexibilis és hatékony csomaggá teszi azt.

3.6.4. Graphviz

A Graphviz [14] egy gráfok vizualizálására szolgáló program. Ahhoz, hogy érthetővé váljon, miként lehet ezt a szoftvert alkalmazni a szakdolgozat témájaként szolgáló programszintézis-rendszer által megalkotott függvénykompozíciók vizualizálására, szükséges pár gráfelméleti fogalom tisztázása.

1. Definíció. *Gráfnak [20] egy (V, E, ϕ) rendezett hármast nevezünk, ahol*

- ***V** a csúcsok halmaza ($V \neq \emptyset$),*
- ***E** az élek halmaza ($V \cap E = \emptyset$),*
- *ϕ pedig az úgynevezett illeszkedési függvény, amely az élek E halmazáról V-beli rendezetlen párok halmazára képez.*

2. Definíció. *A $G = (V, E, \phi)$ gráfot fának [20] nevezzük, ha az összefüggő és körmentes.*

3. Definíció. *Irányított gráfnak [20] egy (V, E, ψ) rendezett hármast nevezünk, ahol*

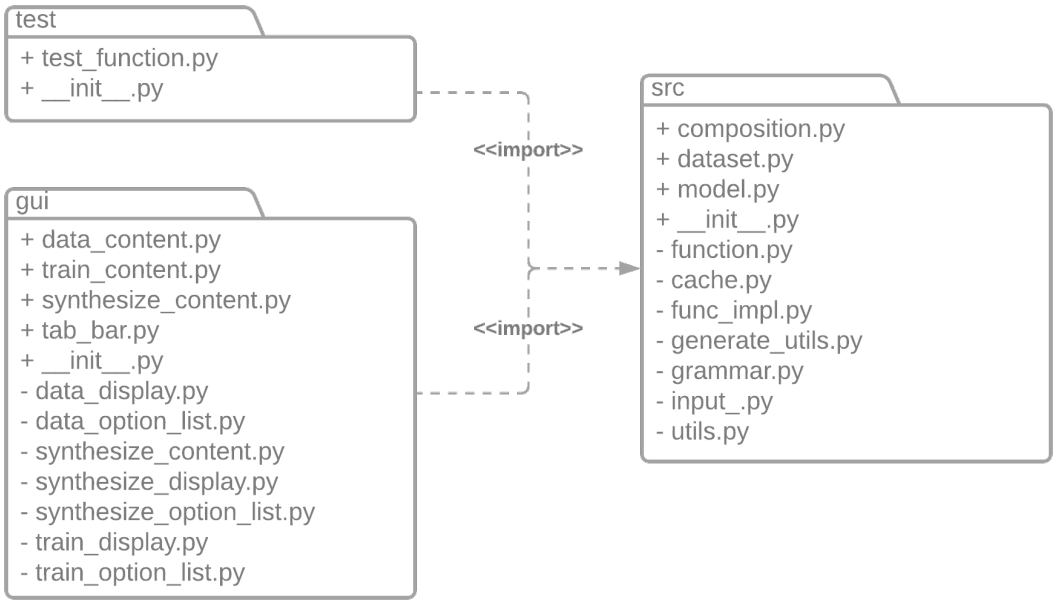
- ***V** a csúcsok halmaza ($V \neq \emptyset$),*
- ***E** az élek halmaza ($V \cap E = \emptyset$),*
- *$\psi : E \to V \times V$ pedig az illeszkedési függvény.*

4. Definíció. *A $G = (V, E, \psi)$ irányított gráfot irányított fának nevezzük, ha összefüggő, nem tartalmaz irányított kört és pontosan egy olyan csúccsal rendelkezik, melynek befoka 0. Ezt a csúcsot a fa gyökércsúcsának nevezzük.*

A dolgozatban bemutatott szoftver által szintetizált függvénykompozíciók irányított faként vannak reprezentálva, így speciális gráfokként felhasználható a Graphviz csomag a kompozíciók vizualizálására is.

3.7. Csomagdiagram

A fejlesztett applikáció csomagdiagramja az alábbi ábrán látható:

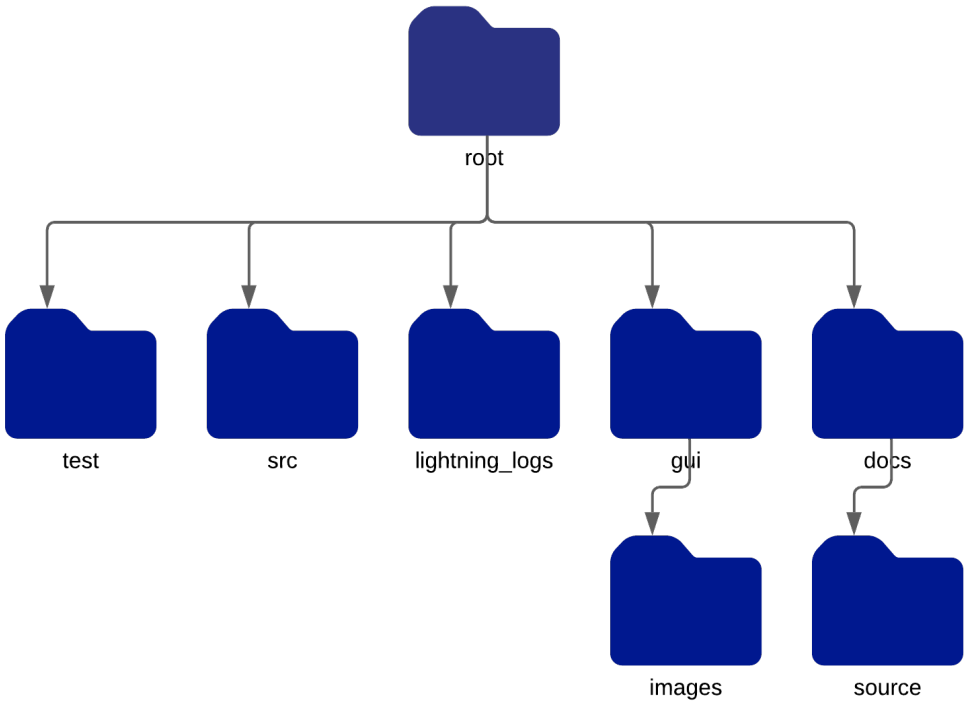


3.2. ábra. A szoftver csomagdiagramja, amely az egyes csomagok összes modulját tartalmazza. Az *src* csomag az applikáció backend-szintű forrásfájljait foglalja magában. A *test* csomag tartalmazza a megírt egységteszteket. A *gui* csomagban a GUI (grafikus felület) moduljait találhatjuk meg.

Az *src* csomagban találhatóak az applikáció backend-jének forrásfájljai. A *test* csomag kizárólag a szoftver fejlesztése során írt egységteszteket tartalmazza, melyekkel megbizonyosodtam a program egyes komponenseinek megfelelő működéséről. A *gui* csomag tartalmazza a grafikus felülethez kapcsolódó összes megjelenítőt.

3.8. Mappaszerkezet

Az alábbi ábra bemutatja a szoftver mappaszerkezetét:



3.3. ábra. A szoftver mappaszerkezete. A *root*, azaz gyökérmappában találhatóak azok a modulok, melyek a szoftver fő funkcióiért felelősek, így közvetlenül is futtathatóak parancssorból. A test mappa tartalmazza a program egységtesztjeit, az src mappa pedig a backend forrásfájljait. A lightning_logs mappába kerülnek elmentésre a neurális háló tanítása során mért tanítási, validációs és tesztelési költség és pontossági értékek, melyeket az alkalmazás felhasznál a TensorBoard szerver indításakor. A gui mappa tartalmazza a grafikus felület betöltéséhez szükséges képeket az image mappában, illetve a GUI forrásfájljait. A docs mappa a Sphinx program által a szoftvert alkotó osztályok és függvények docstring-jeiből generált dokumentációt tartalmazza, melynek generálását inicializáló szkriptek a source almappában találhatók.

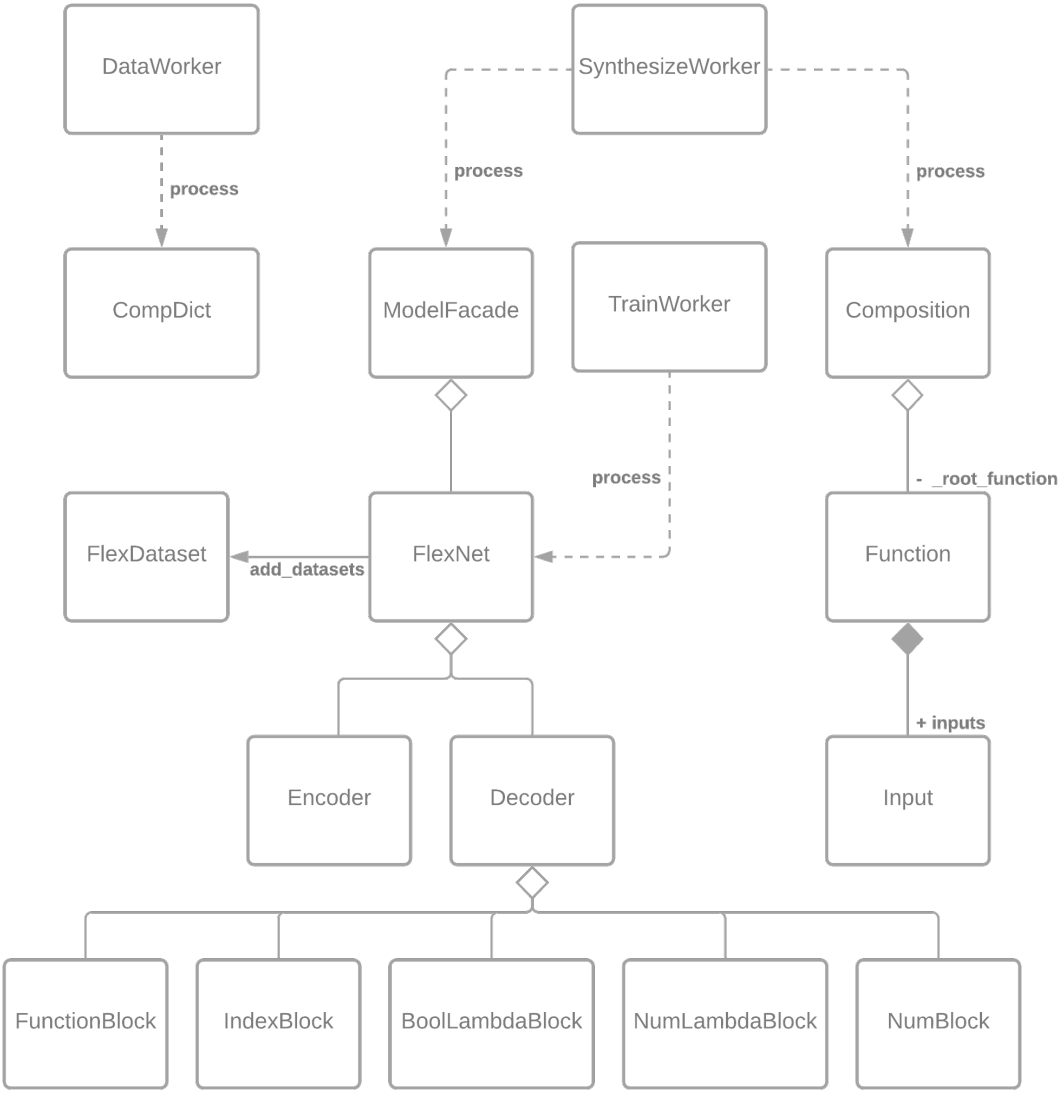
A *root* mappa foglalja magában azon modulokat, melyek parancssorból is közvetlenül futtathatóak, mivel központi funkcionalitást töltenek be a szoftverben. Ilyen például az adatgenerálás, a neurális háló tanítása, vagy a programszintézis-rendszer kiértékelő szkriptje. A test mappában egységteszteket találhatunk, az src mappában pedig a backend forrásfájljait. Ezek olyan fájlok, melyek nem képviselnek központi funkciókat, azonban meglétük elengedhetetlen a szoftver megfelelő működéséhez. A

lightning_logs mappa a neurális háló tanítása során mért tanítási, validációs és tesztelési költséget és pontossági értékeket tárolja. A gui mappa a GUI forrásfájljainak ad otthont, illetve a grafikus felület betöltéséhez szükséges képeket tartalmazza az image mappában. A docs mappa az automatikus dokumentáció Makefile-ját tartalmazza, illetve további forrásfájljait a source almappában.

3.9. Backend komponensek

A fejlesztett szoftver backend-je három fő komponensre bontható. A példagenerálás célja függvénykompozíciók generálása, majd azok felbontása olyan formában, hogy az így kapott adathalmazon az alkalmazott neurális háló sikeresen betanítható legyen. A tanításért felelős komponens célja a neurális háló architektúrájának és további hiperparamétereinek megválasztása oly módon, hogy az egy hatékony részét tudja képezni a szintetizálási folyamatban felhasznált keresési algoritmusnak. A harmadik komponens a szintetizálási folyamat, azonban mivel a nyalábkeresési algoritmust a szoftver fejlesztése során nem én implementáltam, azt kihagyom a bővebb ismertetésből.

A szoftver backend-jének osztálydiagramját az alábbi ábra mutatja be:



3.4. ábra. A szoftver backend-jének osztálydiagramja. A grafikus felület három fő funkciójának háttérben való futtatásáért a *DataWorker*, a *TrainWorker* és a *SynthesizeWorker* osztályok felelősek.

3.9.1. Adatgenerálás

Környezetfüggetlen nyelvtan

Az adatgenerálás előkészítése egy környezetfüggetlen nyelvtan (CFG) [10] meghatározásával indul, mellyel egyben meghatározzuk a programszintézis-rendszer főbb karakterisztikáit és kifejezőerejét is. A szoftverben használt CFG az alábbi ábrán látható:

$$
\begin{aligned}
S &\rightarrow ARRAY\_FUNCTION \\
S &\rightarrow NUMERIC\_FUNCTION \\
ARRAY\_FUNCTION &\rightarrow sort(array) \\
ARRAY\_FUNCTION &\rightarrow take(POS,\ array) \\
ARRAY\_FUNCTION &\rightarrow drop(POS,\ array) \\
ARRAY\_FUNCTION &\rightarrow reverse(array) \\
ARRAY\_FUNCTION &\rightarrow map(NUM\_LAMBDA\_UNARY,\ array) \\
ARRAY\_FUNCTION &\rightarrow filter(BOOL\_LAMBDA\_UNARY,\ array) \\
ARRAY\_FUNCTION &\rightarrow zip\_with(NUM\_LAMBDA\_BINARY,\ array,\ array) \\
NUMERIC\_FUNCTION &\rightarrow max(array) \mid min(array) \\
NUMERIC\_FUNCTION &\rightarrow sum(array) \mid length(array) \\
NUM &\rightarrow NEG \mid 0 \mid POS \\
LESS\_THAN\_MINUS\_ONE &\rightarrow -8 \mid -7 \mid \ldots \mid -2 \\
NEG &\rightarrow LESS\_THAN\_MINUS\_ONE \mid -1 \\
POS &\rightarrow 1 \mid GREATER\_THAN\_ONE \\
GREATER\_THAN\_ONE &\rightarrow 2 \mid 3 \mid \ldots \mid 8 \\
BOOL\_LAMBDA\_UNARY &\rightarrow BOOL\_UNARY\_OPERATOR\ NUM \\
BOOL\_LAMBDA\_UNARY &\rightarrow MOD \\
BOOL\_UNARY\_OPERATOR &\rightarrow == \mid < \mid > \\
MOD &\rightarrow \%\ GREATER\_THAN\_ONE == 0 \\
NUM\_LAMBDA\_UNARY &\rightarrow *\ MUL\_NUM \\
NUM\_LAMBDA\_UNARY &\rightarrow /\ DIV\_NUM \\
NUM\_LAMBDA\_UNARY &\rightarrow +\ POS \\
NUM\_LAMBDA\_UNARY &\rightarrow -\ POS \mid \%\ GREATER\_THAN\_ONE \\
NUM\_LAMBDA\_BINARY &\rightarrow * \mid + \mid - \mid max \mid min \\
MUL\_NUM &\rightarrow NEG \mid 0 \mid GREATER\_THAN\_ONE \\
DIV\_NUM &\rightarrow LESS\_THAN\_MINUS\_ONE \mid GREATER\_THAN\_ONE
\end{aligned}
$$

3.5. ábra. A nyelvtan által támogatott függvények megadása egy környezetfüggetlen nyelvtannal. A *sort* növekvő sorrendbe rendezi a bemenetéül kapott lista elemeit. A *take* függvény megtartja, a *drop* függvény eldobja az első *POS* elemét a bemeneti listának. A *reverse* függvény megfordítja a bemenetéül megadott listát. A *map* és *filter* függvények olyan magasabbrendű függvények, melyek egyetlen listát várnak paraméterként. A *map* függvény végrehajtja a *NUM_LAMBDA_UNARY* függvényt az inputlista minden elemén. A *filter* függvény csak azokat az elemeit tartja meg a bemeneti listájának, melyekre a *BOOL_LAMBDA_UNARY* függvény igazat ad vissza. A *min* és a *max* függvények a legkisebb és legnagyobb elemeit adják vissza a bemenetüknek, ebben a sorrendben. A *sum* függvény a lista elemeinek összegét, a *length* függvény ezek számát adja meg. A *zip_with* olyan magasabbrendű függvény, mely a kimenete i-edik indexére a bemeneti listái i-edik indexén található elemein végrehajtott operátorának eredményét teszi.

A függvények nem változtatják meg a bemenetükként kapott listát, így azok mellékhatásmentes, tiszta függvények. Minden esetben egy új lista kerül visszaadásra. Az egyes függvények hatását egy táblázat mutatja be:

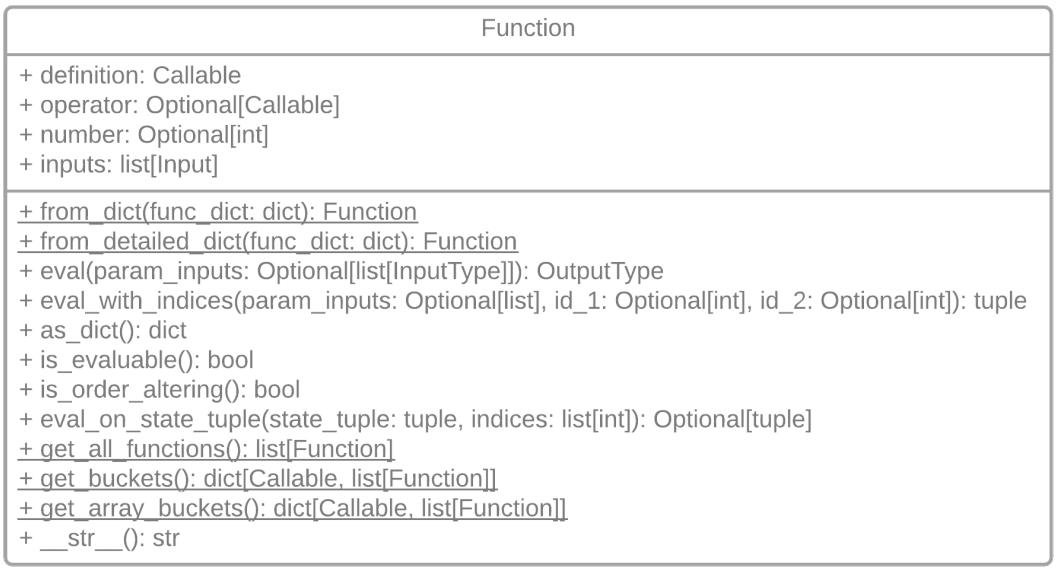
Függvénytípus	**Hatás**
sort	Növekvő sorrendbe rendezi a bemeneti lista elemeit.
take	Megtartja a bemeneti lista első *POS* elemét.
drop	Eldobja a bemeneti lista első *POS* elemét.
reverse	Megfordítja a bemeneti listát.
map	Végrehajtja a *NUM_LAMBDA_UNARY* függvényt az inputlista minden elemén.
filter	Csak azokat az elemeit tartja meg a bemeneti listának, melyekre a *BOOL_LAMBDA_UNARY* függvény igazat ad vissza.
min	Visszaadja a bemeneti lista legkisebb elemét.
max	Visszaadja a bemeneti lista legnagyobb elemét.
sum	Visszaadja a bemeneti lista elemeinek összegét.
length	Visszaadja a bemeneti lista hosszát.
zip_with	Kimenete i-edik indexére a bemeneti listái i-edik indexén található elemein végrehajtott operátorának eredményét teszi.

3.1. táblázat. A környezetfüggetlen nyelvtan által meghatározott függvények hatásai táblázat formájában.

A környezetfüggetlen nyelvtan oly módon kerül felhasználásra, hogy kigenerálom vele az összes lehetséges felparaméterezett függvényt, majd ezen string-ekből a *Natural Language Toolkit* parser-évl token-ek asszociatív tömbjét nyerem ki, ezekből pedig *Function* objektumokat hozok létre.

Function osztály

A *Function* osztály osztálydiagramja az alábbi ábrán látható:



3.6. ábra. A *Function* osztály osztálydiagramja. Lehetőség van a @dataclass dekorátor által generált alapértelmezett konstruktort felhasználni egy függvényobjektum létrehozására, azonban a *from_ dict* gyártófüggvény szintén felhasználható objektum konstruálására egy asszociatív tömbből, amely jól illeszkedik a *Natural Language Toolkit* parser-éhez. A *from_ detailed_ dict* gyártófüggvény kizárólag a szintetizálási folyamatban kerül felhasználásra, és kényelmi funkciót tölt be.

Egy *Function* objektum létrehozható a @dataclass dekorátor által generált alapértelmezett konstruktorral, illetve a *from_ dict* gyártófüggvénnyel is, amely jól illeszkedik a *Natural Language Toolkit* által nyújtott parser-hez, így további átalakítás nem szükséges a tokenekből való konstruáláshoz. Az *eval* metódus egy függvényobjektum kiértékelését teszi lehetővé, amennyiben annak *inputs* adattagja nem üres, vagy meg van adva egy függvényinput a metódus paraméterében. Az *inputs* adattag *Input* objektumok listája. Az *Input* osztály egy egyszerű csomagolóosztály egy bemeneti lista körül, mely rendelkezik egy *id* adattaggal, amely az egy kompozíción belüli "copy-zott" (lemásolt) bemenetek esetén megegyezik, így számon tartható a *copy* függvény hatása, mely a példagenerálás során kerül és a szintetizálási folyamatban egyaránt felhasználásra kerül. Erre a 1.3 ábrán láthatunk példát: itt a take(2) függvény bemeneténél (az ábrán None) egy verziókódot olvashatunk: v0.1, egy másik ágon pedig v0.2. Ez azt jelenti, hogy mindkét bemenet id-ja 0, azaz lemásolt bemenetek, és a bal oldali az első előfordulás, a jobb oldali pedig a második.

Ezzel el is érkeztünk a dolgozat keretén belül bemutatott rendszer egyik legfontosabb kontribúciójához. A neurális háló szempontjából kifejezetten fontos tudni azt, hogy egy ág másolata-e egy másik ágnak egy kompozíción belül, vagy sem. Ugyanis a programszintézis-rendszer alulról felfelé épít fel egy függvénykompozíciót a szintetizálási folyamat során, és rendelkezik egy *copy* függvénnyel, amellyel egyszerűen le tud másolni egy ágat, és újra fel tudja használni azt később. A *zip_with* függvény ilyen szempontból tekinthető a *copy* függvény inverzének: ez éppen hogy összevon két ágat egyetlen ágba.

A programszintézis-rendszert irányító neurális hálót azonban meg kell tanítani arra, hogy ezt a *copy* függvényt megfelelő konfidenciával tudja prediktálni, ehhez pedig számos olyan példa szükséges, melyben alkalmazva van ez a függvény.

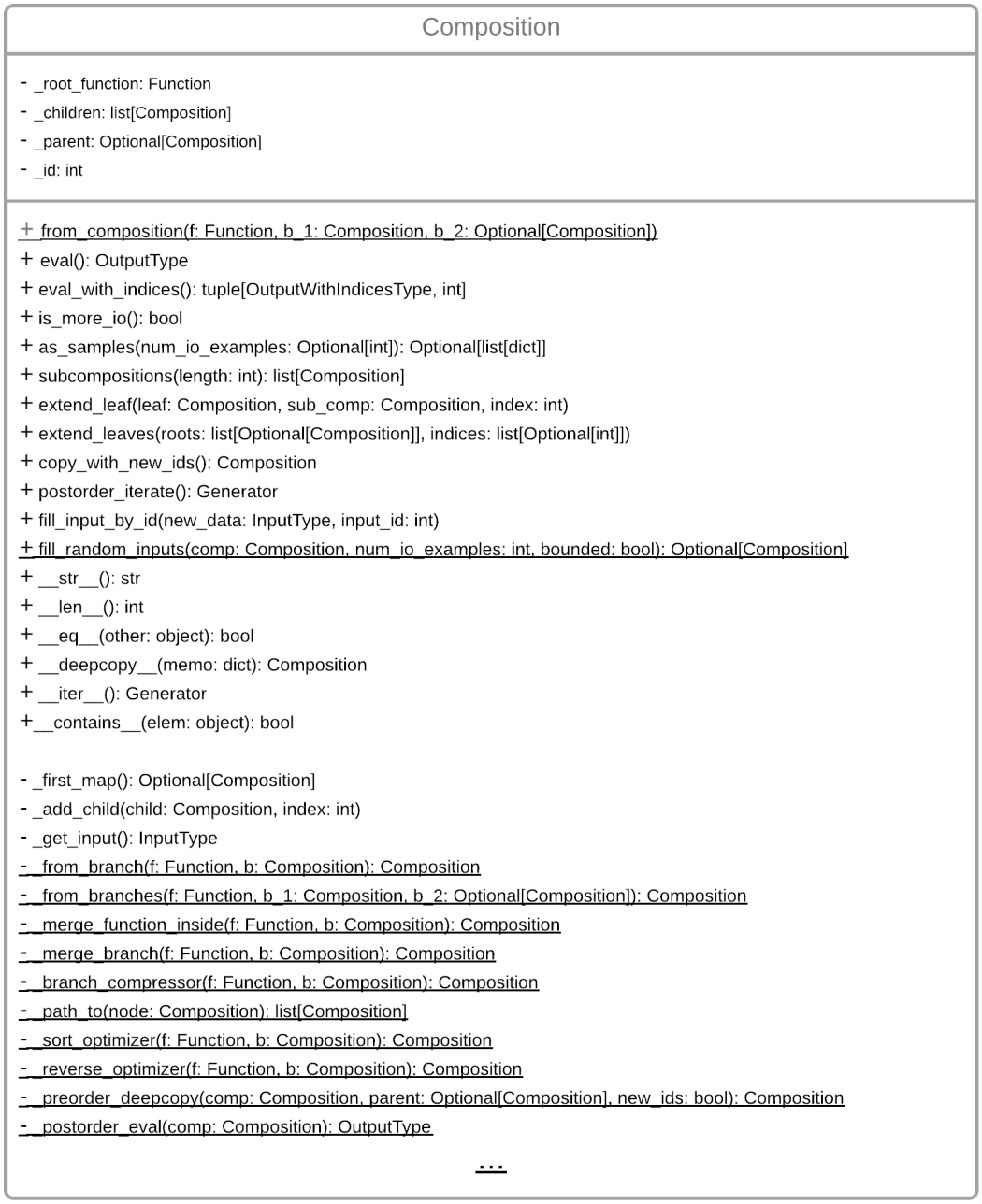
Amennyiben egy függvénykompozíció két, egyébként teljesen identikus ága eltérő id-val rendelkezik, akkor ezek nem egymás másolatai. Fontos kiemelni, hogy a függvénykompozíció használata során teljesen indifferens, hogy az egyes ágai egymás másolatai-e, vagy sem. Azonban amikor példát készítünk az adott kompozícióból a neurális háló számára, akkor nagy jelentőséget kap, hogy egy részkompozíciót lemásoljon-e, hogy azon később külön számításokat végezzen azon, vagy nincs az adott ág másolatára szükség a szintetizálási folyamatban.

Az *eval_with_indices* metódust kizárólag a *Composition* osztály használja fel optimalizációs célokra. Az *as_dict* metódus egy függvényobjektum asszociatív tömbként vett reprezentációját adja meg, melyet a tanítópéldák generálása során használok fel, ugyanis az asszociatív tömb már kiírható egy *.json* kiterjesztésű fájlba. A többi metódus kényelmi funkciókat biztosít a generáláshoz, így külön szót nem érdemelnek.

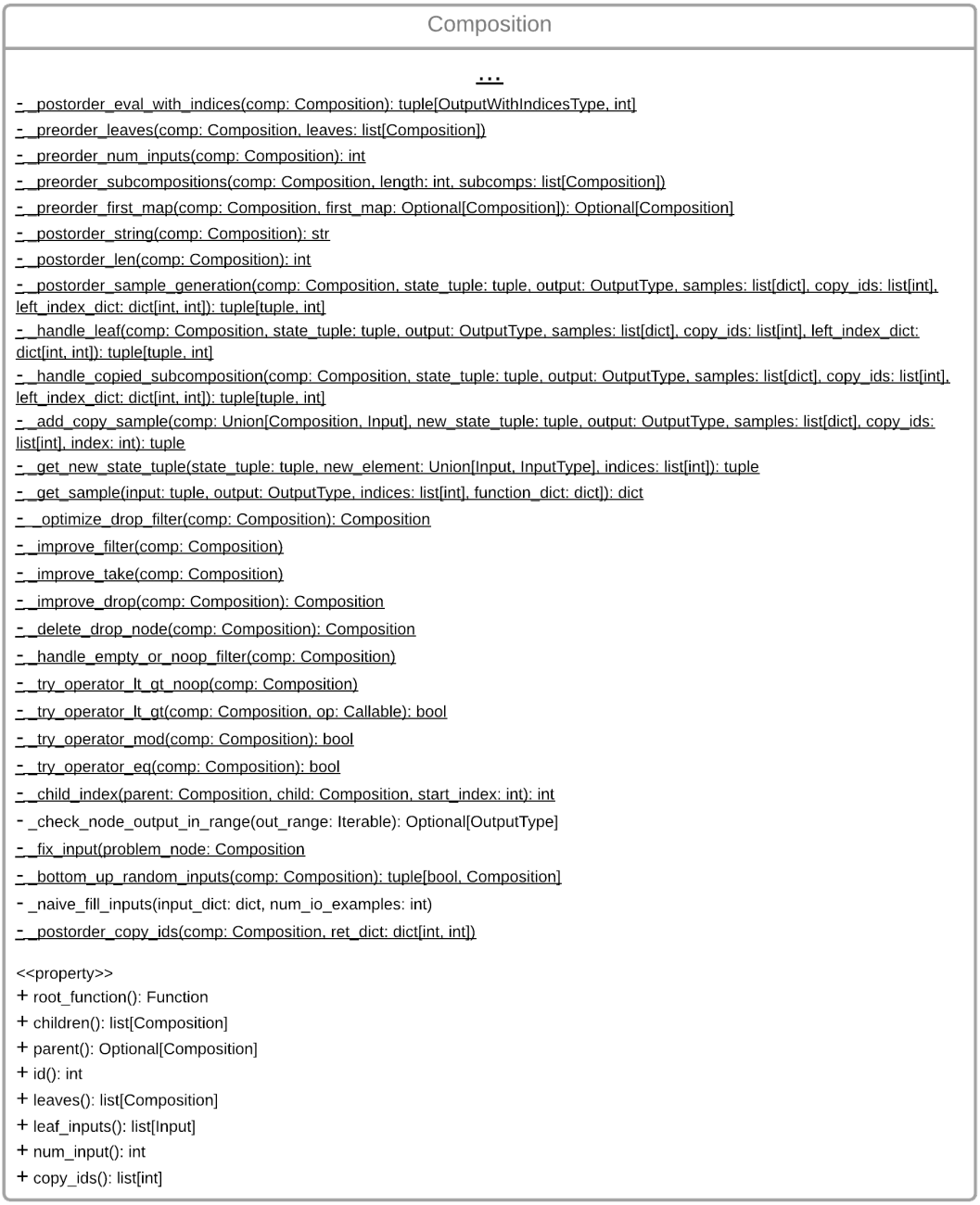
A *Function* objektumokat ezután *Composition* objektumok építőköveiként használom fel.

Composition osztály

A *Composition* osztály osztálydiagramja az alábbi két ábrán látható:



3.7. ábra. A *Composition* osztály osztálydiagramjának első fele. Az osztály lehetőséget biztosít függvénykompozíciók felépítésére, kiértékelésére, string-reprezentációjuk megadására, és többek között a kompozíció tanítópéldává történő áttranszformálására is.



3.8. ábra. A *Composition* osztály osztálydiagramjának második fele. Az osztály lehetőséget biztosít az eddigiekben említetteken kívül a függvénykompozíciók optimalizálására is a felépítésük során, melyek garantáltan ekvivalens, ámde a tanítási folyamathoz egyértelműbb tanítópéldákat adnak.

A *Composition* osztály központi szerepet játszik a szoftver működésében. Segítségével tetszőleges, a nyelvtan szabályainak eleget tevő függvénykompozíciókat építhetünk fel, melyhez számos metódust biztosít az osztály. Először ezeket szeret-

ném sorra venni.

A *Composition* objektumokat lehetséges a @dataclass dekorátor által generált alapértelmezett konstruktorral létrehozni, ekkor csak a *_root_function* attribútumot kell és lehet megadnunk, azaz ezzel a konstruktorral csak egy egyetlen függvényt tartalmazó kompozíció készíthető.

Minden más kompozíció létrehozásához használható a *from_composition* gyártófüggvény. Ez minden esetben vár egy függvényt, melyet a második (és opcionálisan a harmadik) paraméterben megadott kompozíció(k) tetejére teszünk, ezzel képezve egy új kompozíciót. A *zip_with* függvény két listát is vár paraméterül, így a felépített kompozíció alakja egy bináris fa lesz. Erre a 1.3 ábra nyújt példát.

Az *extend_leaf* és *extend_leaves* metódusok segítségével az adott kompozíció levelét vagy adott leveleit tudjuk kiterjeszteni a paraméterként megadott egy, vagy több kompozícióval. Ezzel tehát fentről lefelé tudunk kompozíciókat építeni.

Az adatgeneráláshoz a szoftver dinamikus programozást használ. Amennyiben egy n hosszú kompozíciót szeretnénk generálni, akkor a szoftver először kigenerálja az összes egy hosszúságú kompozíciót, majd a két hosszúságú kompozíciókat a már kigenerált egy hosszúságú kompozíciókból és egy ezekhez hozzávett új függvényből adja meg. Ez így folytatódik tovább, és mire az n hosszúságú kompozíciókhoz érünk, már rendelkezésünkre állnak az $(n-1)$ hosszúságú kompozíciók, melyekhez megintcsak egyetlen új függvényt kell hozzávennünk. Egy függvénykompozíció hossza alatt minden esetben az azt alkotó függvények számát értem.

Említésre méltó még a *fill_random_inputs* metódus is. Ezzel random egész számokból álló bemeneti listákkal tudjuk feltölteni az adott kompozíció leveleit. A metódus alapértelmezetten figyeli azt, hogy úgy határozza meg a bemeneteket, hogy az így kiértékelt kompozíció kimenete beleessen a szoftver konfigurációs fájljában megadott kimeneti intervallumba. Ennek elérése érdekében egy iterációs algoritmus használ a metódus, melyben leköveti azt, hogy a kimenet megadott intervallumból kilógó elemeit melyik input melyik elemei befolyásoltak, és azokat növeli vagy csökkenti attól függően, hogy hol helyezkednek el a kimenethez képest a kompozícióban, és a köztük levő úton milyen függvények találhatóak. Az elemek lekövetéséhez a metódus az *eval_with_indices* metódust használja fel.

A függvényekhez hasonlóan a kompozíciókat is lehetséges kiértékelni az *eval* me-

tódussal. Fontos különbség, hogy egy *Composition* objektumnak már mindenképp fel kell lennie töltve az összes bemenetével a levélszintjén, különben nem kiértékelhető a kompozíció. A kiértékeléshez egy postorder fabejárási algoritmusra van szükségünk, ugyanis ahhoz, hogy a kompozíció egy nódusát kiértékeljük, szükséges annak gyerekei értékeinek ismerete, amely egy rekurzív algoritmushoz vezet. Általánosan is elmondható, hogy a *Composition* osztály algoritmusai

A postorder bejárás pszeudokódja az alábbi:

Algoritmus 1 Egy általános postorder bejárás

Funct postorder(*tree*)

```
if tree != nil then
    postorder(tree.left)
    postorder(tree.right)
    process(tree)
end if
```

A generálásnál kifejezetten fontos a kompozíciók optimalizálása annak érdekében, hogy a neurális hálónak egy jól tanítható feladatot adjunk meg. Először azonban szükséges annak a meghatározása, hogy a neurális háló milyen szerepet tölt be a programszintézisben. A programszintézis a nyalábkeresés gráfkereső algoritmus végrehajtásával történik. Ez az algoritmus felfedezi a problémateret, azaz az összes lehetséges függvénykompozíció megszámlálhatóan végtelen számosságú terét. Természetesen ez például egy szélességi gráfkereséssel pár lépés után lehetetlen feladatnak bizonyulna: a tárolandó és kiterjesztendő nódusok száma exponenciálisan nő. Nem nyújtana jó megoldást a mélységi gráfkeresés sem, ugyanis a megoldás eltévesztése esetén ez a végtelenségig megy előre a végtelen csúccsal rendelkező állapotgráfunkban. Jó kompromisszum tehát a nyalábkeresés (beam search): ennek tár -és időigénye nem száll el a lépések számával, azonban megfelelő számú (nyalábméretnyi) lehetséges megoldás-jelöltet számon tart, így nem csak egyetlen jelöltre támaszkodik általánosan, mint a mélységi gráfkeresés.

A beam search algoritmust jól elkülöníthető lépésekre, vagy más szóval szintekre lehet bontani. A neurális háló feladata így már egyszerűen definiálható: **az *n*-edik szinten meg kell adnia, hogy az addig szintetizált kompozíciók (azaz a je-**

löltek) esetén mi az az egy-egy felparaméterezett függvény, amely azokat egy lépéssel közelebb vinné a megoldáshoz. Ez természetesen egy meglehetősen nehéz feladat, és a tapasztalatok azt mutatják, hogy nehéz olyan hálóarchitektúrát megalkotni, amely könnyen tanítható marad, mégis megfelelően ráilleszkedhet a problémára a komplexitását tekintve. Emiatt láthatjuk a nyalábkeresés szükségességét is: a háló pontatlanságát ellensúlyozza az, hogy egyszerre több jelöltet is számon tart az algoritmus, melyek esetlegesen a megoldást jelenthetik.

A neurális háló feladatának tisztázása után az optimalizáció létjogosultsága már nyilvánvaló. A teljesség igénye nélkül az alábbiakban felsorolok pár esetet, ahol optimalizációra van szükség.

- **filter(>5, [-1, -2, -3])**: Ebben az esetben az egyetlen függvényből álló kompozíció egy üres listát eredményez. Ez nem minősül jó példának a neurális háló betanításához, ugyanis üres listát bármilyen bemeneti listából tudunk képzeni a *drop* függvény többszöri alkalmazásával. Ekkor tehát optimalizáljuk a *filter* függvény numerikus paraméterét: csökkentjük azt egészen addig, amíg az -2 nem lesz. Ekkor ugyanis a függvényt kiértékelve már a **[-1]** egyelemű listát kapnánk eredményül, amely így már egy megfelelő tanítópélda.

- **filter(<5, [-1, -2, -3])**: Az előző példa ellenpéldája. Ebben az esetben a filter függvény nem változtat a bemenetén, így nem jelent érdekes tanítópéldát. Kifejezetten káros lenne az ilyen példák meghagyása, hiszen ezzel arra optimalizálnánk a hálót, hogy ha lát egy listát, válasszon egy olyan függvényt, amely nem változtat azon a listán. Ez ezen példa esetén valóban egy lépéssel közelebb vinné a rendszert a megoldáshoz, azonban könnyen látható, hogy valójában nem történt tényleges lépés.

- **drop(4, [1, 2, 3])**: A megintcsak egy függvényből álló kompozíciót kiértékelve üres listát kapunk. Javíthatunk a példán, ha a numerikus paramétert elkezdjük csökkenteni. A 3 esetén még üres listát kapnánk, a 2 esetén azonban már nem, így a példa már nem triviális, érdemes azt a neurális háló optimalizálásához felhasználnunk.

Mivel a neurális háló feladata egy alulról felfelé épített, befejezetlen (azaz felülről lezáratlan) függvénykompozíció esetén annak a függvénynek a megadása, amely a

függvénykompozíciót egy lépéssel közelebb viszi a megoldáshoz, az adathalmaznak is hasonló felépítéssel kell rendelkeznie, melyen a neurális hálót tanítjuk. Az adatgenerálási fázis végére előállt adathalmazt felfoghatjuk asszociatív tömbök listájaként. A lista hossza a generált tanítópéldák száma. Az egyes listaelemek a tanítópéldák. Egy tanítópéldához tartozó asszociatív tömb az alábbi kulcsokkal rendelkezik:

- **input**: Egy kiértékelt, ámde befejezetlen (azaz akár több, összekapcsolatlan részágból álló) függvénykompozíció.
- **output**: Az elvárt kimenet, azaz a szintetizálandó függvénykompozíció kiértékelése.
- **next_transformation**: A tanítóminta címkéje. Az a transzformáció, melyet a programszintézis-rendszernek végre kellene hajtania a befejezetlen függvénykompozíció megfelelő ágán ahhoz, hogy az *input* érték egy lépéssel közelebb kerüljön az *output* értékhez.

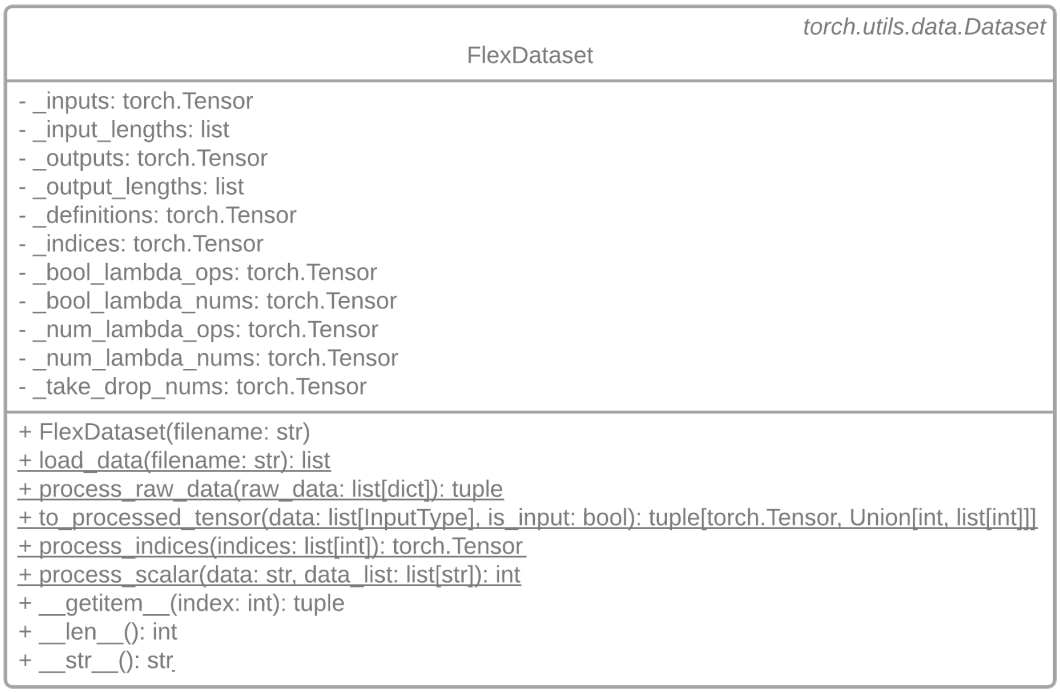
3.9.2. A neurális háló tanítása

FlexDataset osztály

Egy neurális háló, mint az előzőekben láthattuk, csupán numerikus bemeneti értékeken képes számításokat végezni. Ennek érdekében a generálási lépésben létrehozott adathalmazt tisztán numerikus értékekből álló formára kell hozni. Ehhez a függvények egyes komponenseihez, és azon belül azok lehetséges értékeihez hozzárendelek egy indexet, amely azt az értéket reprezentálja. Például a *zip_with* függvénynek 5 lehetséges operátora van. Ezeket felsorolhatjuk egy listában: [+, -, *, min, max]. Ekkor mindegyik operátorhoz egyértelműen hozzárendeltünk egy numerikus értéket, azaz egy indexet, ami azt reprezentálja. Ez a + operátor esetén a 0, a - operátor esetén az 1, és így tovább.

Szükséges továbbá az is, hogy a gyors és stabil tanítás érdekében kötegekben (batch-ekben) adjuk át a neurális hálónak a tanítópéldákat, ne pedig egyenként. Ehhez elvárt az, hogy a változó hosszúságú bemeneti és kimeneti listákat egységes hosszúságúra hozzuk (pad-eljük).

Az imént említett transzformációkat a *FlexDataset* osztály végzi el, amely így a neurális háló által elfogadott formátumban szolgáltatja az adatokat a hálónak. Ennek az osztálynak az osztálydiagramja az alábbi ábrán látható:



3.9. ábra. A *FlexDataset* osztály osztálydiagramja. Az osztály célja, hogy olyan formátumban prezentálja az adatot a neurális hálónak, melyet az fel tud dolgozni, és fel tud használni a tanulási folyamatban. Ehhez az adott *FlexDataset* objektum az osztályszintű *process_raw_data* metódust hívja segítségül, amely elvégzi a formai konverziókat.

A *FlexDataset* osztály *process_raw_data* metódusa felelős azért, hogy elvégezze a formai konverziót az adatgenerálás eredményéül kapott adathalmaz és a neurális háló által elvárt formátum között.

Dolgozatomban említettem, hogy egy neurális háló neuronjait szokás rétegekbe szervezni, azonban az eddigi téma nem kívánta meg, hogy az egyes rétegfajtákat bővebben taglaljam. A rendszerben használt neurális háló architektúrájának ismertetéséhez azonban már szükségessé válik két speciális rétegfajta ismerése.

Sűrűn összekötött réteg

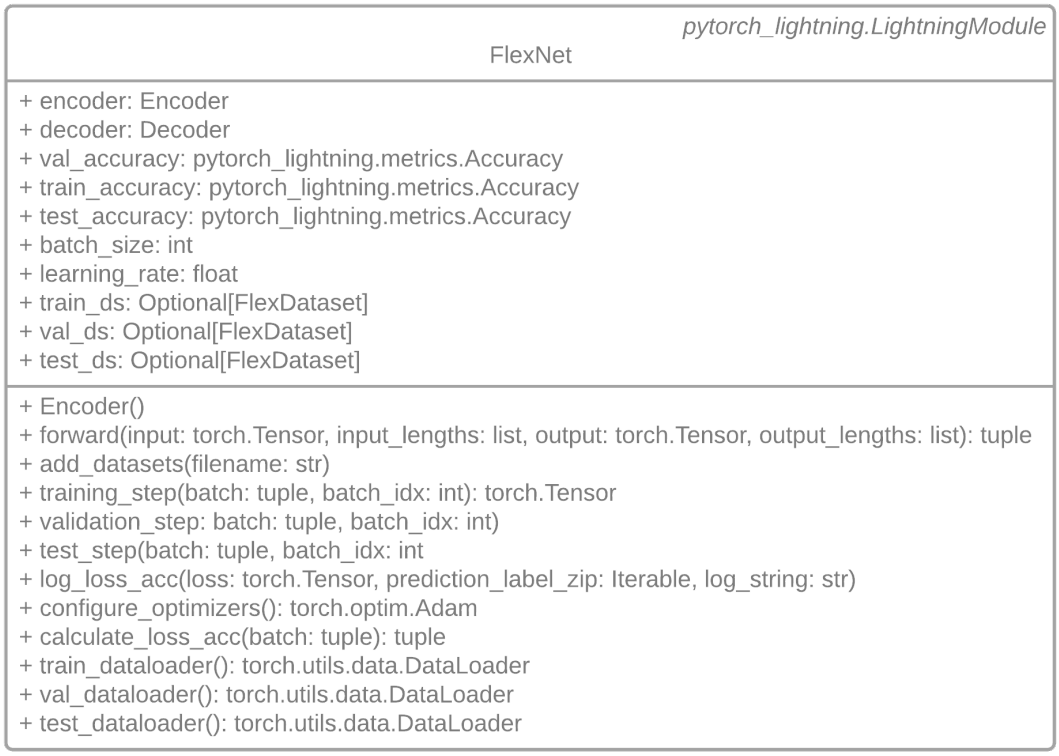
A sűrűn összekötött (densely connected, fully connected, linear) réteg talán a legegyszerűbb rétegtípus. Ebben az i-edig réteg összes neuronja az (i-1)-edik réteg összes neuronjához kapcsolódik. Ez a rétegtípus gyakran használt jól strukturált adatok esetén, melyeknél a bemenet egy adott indexű elemének szemantikus jelentése nem változik nagy mértékben az egyes minták esetén.

Rekurrens réteg

A sűrűn összekötött rétegek limitációja, hogy a bemenet dimenzionalitását előre ismernünk kell. Ez nehezen megoldható, amennyiben egy változó hosszúságú szekvenciát kapunk bemenetként. (Ekkor természetesen ha tudunk mondani egy felső korlátot a bemenet hosszára, akkor alkalmazhatnánk elméletben sűrűn összekötött rétegeket a feladat megoldásához, azonban ezek általában gyengén szerepelnek.) Ilyenkor jönnek jól a rekurrens rétegek, melyek tetszőlegesen hosszú szekvencia feldolgozására képesek. Ezek minden esetben egy rejtett reprezentációt tartanak számon, amely egy n-dimenziós vektor. Ez tehát azt jelenti, hogy egy rekurrens rétegnek egy nagyon hosszú szekvencia minden eleméből kinyert tudást egyetlen vektorban kell számon tartania. Ez természetesen egy kifejezetten nehéz feladat, ezért nagyon hosszú szekvenciák esetén általában a rekurrens rétegek sem teljesítenek kiemelkedően, azonban a legtöbb felhasználásra több, mint jó eredményt nyújtanak.

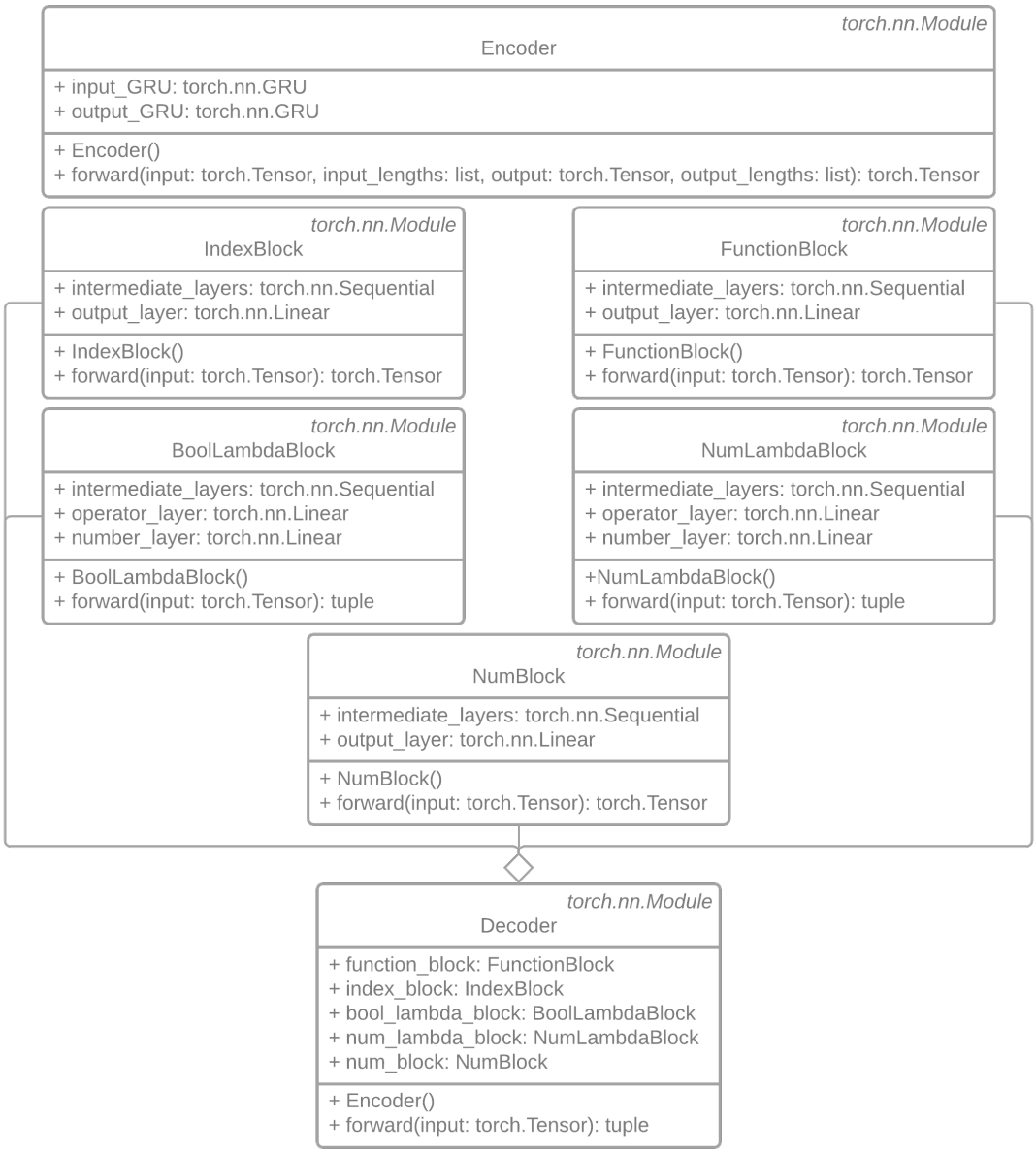
FlexNet osztály

A fejlesztett szoftver *FlexNet* osztálya implementálja a létrehozott programszintézis-rendszer szerves részét képző neurális háló architektúráját, illetve end-to-end módon magában foglalja az annak betanításához szükséges adatokat és optimalizációs algoritmusokat is. Az osztály osztálydiagramja az alábbi ábrán látható:



3.10. ábra. A *FlexNet* osztály osztálydiagramja. Az osztály end-to-end módon foglalja magában a definiált hálóarchitektúra mellett a felhasznált adathalmazokat és az azok pipeline-jául szolgáló *DataLoader* objektumokat, illetve az optimalizációs algoritmusokat is. A *forward* metódus a "lelke" az osztálynak, ugyanis ebben implementálhatjuk, hogy a neurális hálónk hogyan dolgozza fel a paraméterként kapott adatot, és hogyan állítsa elő belőle a kívánt kimenetet. A *training_step* metódus legalább ilyen fontos: Ez a *forward* metódust hívja segítségül a háló predikciójának megadásához, majd a predikciót összehasonlítja az elvárt eredménnyel a költségfüggvénnyel, és az így kapott költséget visszaadja. A *PyTorch Lightning Trainer* osztálya ezt a metódust hívja iteratívan a konvergencia eléréséig a tanítási folyamat során.

Az általános ismertetés után érdemesnek tartom a modell egyes komponenseinek tárgyalását. Ezeket az alábbi ábra mutatja be:



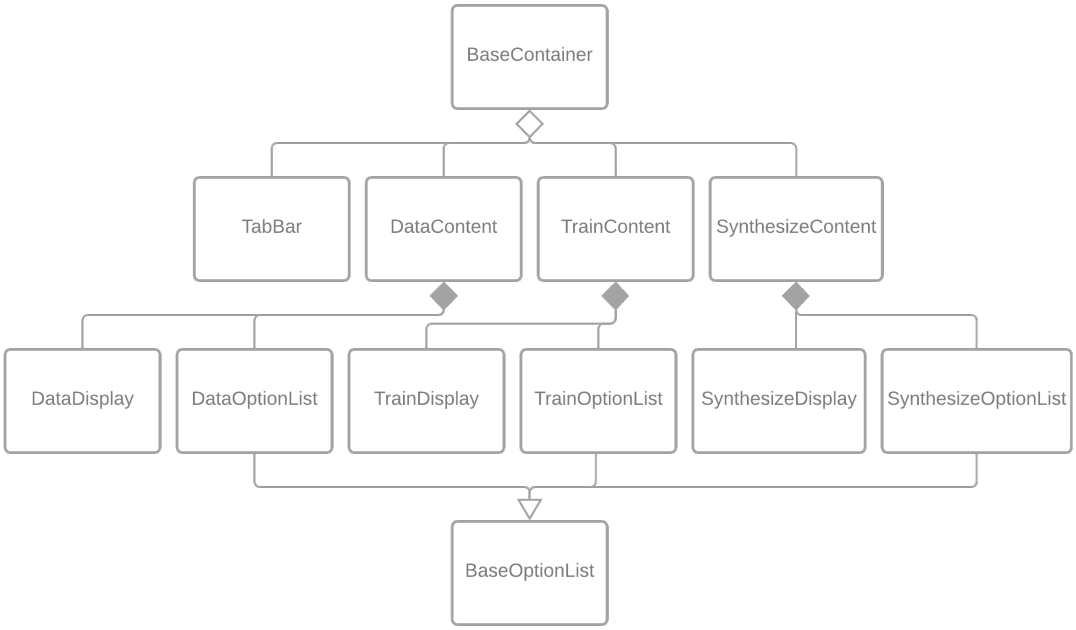
3.11. ábra. A *FlexNet* osztály egyes komponenseinek osztálydiagramjai. Az *Encoder* rész felelős a programszintézis-probléma bemenetének és kimenetének egy belső reprezentációba való elkódolásáért. A neurális hálónak 7 outputfeje van, melyeket az 5 db *Block*-kal végződő részkomponensek adják meg. A *Decoder* modul ezen 5 komponenst foglalja magában, így a rejtett reprezentáció dekódereként szolgál.

Az *Encoder* rész felelős a megadott probléma bemenetének és kimenetének egy belső reprezentációba való elkódolásáért. A további blokkok ezt a rejtett reprezentációt használják fel a predikcióik megadásához. Az *IndexBlock* komponens felelős azért, hogy a félkész kompozíció részágai közül kiválassza azt, amelyikre a prediktált

függvényt alkalmazni kell. A *FunctionBlock* felelős az alkalmazandó függvény típusának prediktálásáért. A *BoolLambdaBlock* felelős a *filter* függvény operátorának és operandusának prediktálásáért. Amennyiben a *FunctionBlock* által prediktált függvénytípus nem *filter*, akkor nem kerül felhasználásra a blokk által adott kimenet. A *NumLambdaBlock* felelős *map* függvény esetén az operátor és az operandus meghatározásáért, míg *zip_with* függvény esetén csak az operátor van figyelembe véve, ugyanis a *zip_with* függvény nem rendelkezik numerikus paraméterrel. A *Decoder* ezen outputfejeket foglalja magában, így a teljes hálónak összesen 7 outputfeje van.

3.10. Frontend komponensek

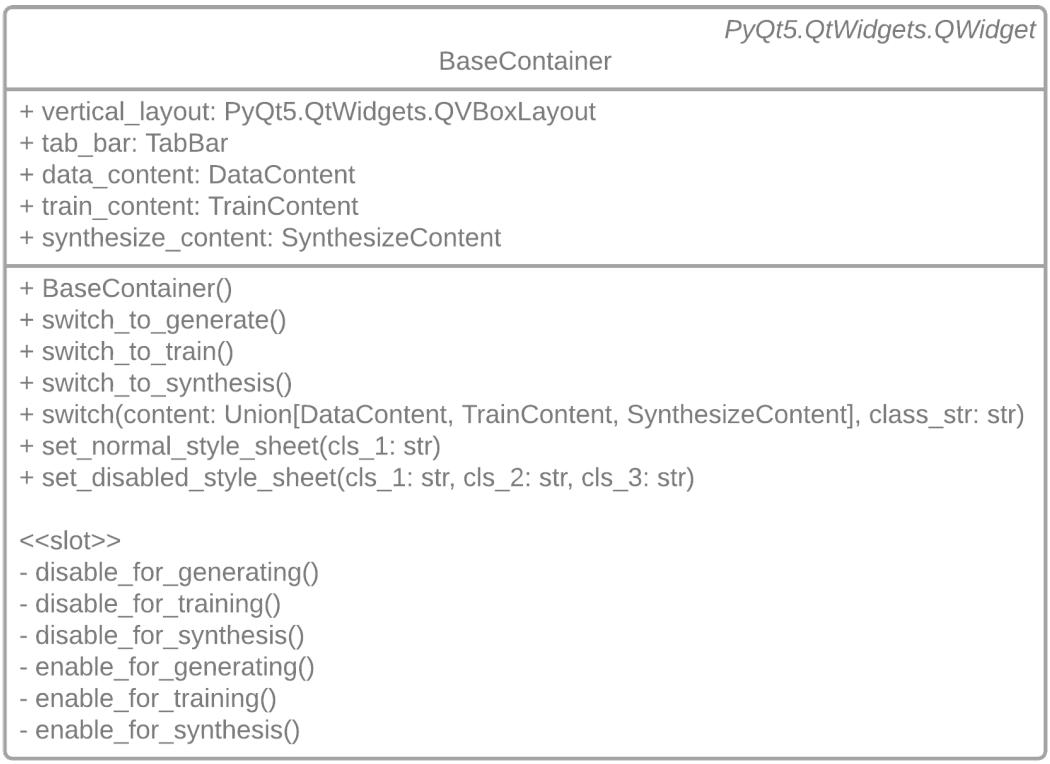
A fejlesztett szoftver frontend-je szintén három komponensre bontható, ugyanis a grafikus felületnek három lapja van: *Data Generation*, azaz adatgenerálás, *Training*, azaz tanítás, és *Synthesizing*, azaz szintetizálás. Látható, hogy a Frontend felosztása leköveti a backend funkcionalitását. A szoftver frontend-jének osztálydiagramját az alábbi ábra mutatja be:



3.12. ábra. A szoftver frontend-jének osztálydiagramja. A grafikus felület három komponensre bontható, ugyanis három lapja van: *Data Generation*, *Training* és *Synthesizing*, melyek a szoftver fő funkcionalitásának megjelenítéséért felelősek.

3.10.1. BaseContainer osztály

A *BaseContainer* a grafikus felület osztályhierarchiájában legmagasabban levő osztály, amely a teljes GUI-ját magában foglalja. Ennek osztálydiagramja a következő ábrán látható:



3.13. ábra. A *BaseContainer* osztály osztálydiagramja. Az osztály magában foglalja a *TabBar* vezérlőt, amely a lapok közti váltogatásért felelős, illetve a *DataContent*, a *TrainContent* és a *SynthesizeContent* vezérlőket, melyek rendre az adatgenerálást, a neurális háló tanítását és a szintetizálási folyamatot jelenítik meg.

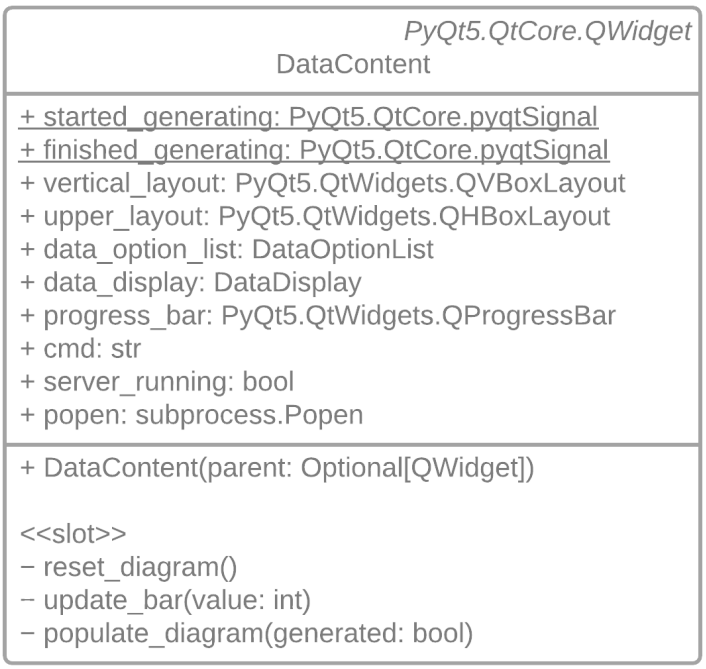
A *BaseContainer* vezérlő tartalmazza a *TabBar* vezérlőt, mellyel a grafikus felület egyes lapjai között váltogathatunk, illetve a *DataContent*, *TrainContent* és *SynthesizeContent* vezérlőket, melyek az egyes lapok tartalmát jelenítik meg.

Az egyes lapok között a *switch_to_ ** metódusokkal váltogathatunk. A *disable_for_ ** metódusok a lapok közti váltogatást hivatottak letiltani, amennyiben egy erőforrásigényes folyamat futtatásába kezdett a felhasználó annak érdekében, hogy ne lehessen egyszerre több, a rendszert lényegesen megterhelő processzt futtatni és a grafikus felület használata ne jelentsen jelentős overhead-et a konzolos használattal

szemben. Az *enable_for_** metódusok felelősek az egyes folyamatok befejeztével a lapok közti váltogatás újbóli engedélyezéséért.

3.10.2. DataContent osztály

A *DataContent* vezérlő az adatgenerálási folyamatot jeleníti meg, illetve statisztikát készít az egyes függvények eloszlásáról a generált adathalmazban. A felület bal oldalán számos, a generálandó adathalmazt befolyásoló paramétert beállíthatunk. Az osztály osztálydiagramját az alábbi ábra mutatja be:



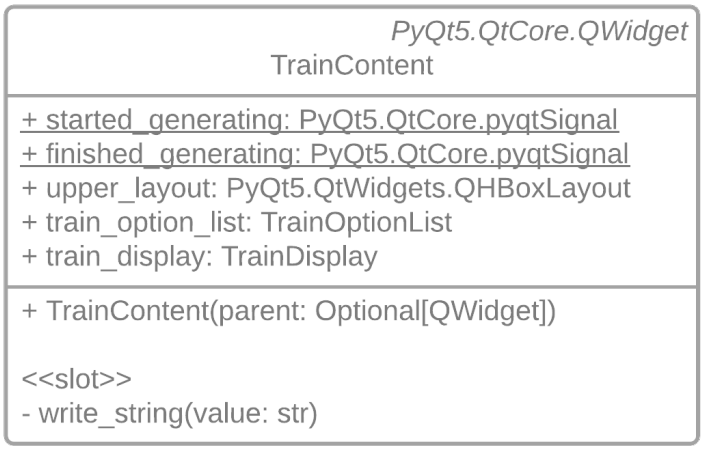
3.14. ábra. A *DataContent* osztály osztálydiagramja. A vezérlő eseményekkel kommunikál az őt magába foglaló *BaseContainer* vezérlővel. A vezérlő aszinkron módon indítja el az adatgenerálás folyamatát, és folyamatosan frissíti az ablak alján látható töltőcsíkot a generálás előrehaladtával. Végül a felület lekéri a backend által kiszámolt statisztikákat és egy oszlopdiagramban megjeleníti azokat.

A vezérlő a *started_generating*, illetve a *finished_generating* eseményekkel kommunikál a szülő widget-évl, amely egy *BaseContainer* objektum. A vezérlő aszinkron módon indítja el a generálás folyamatát, és az *update_bar* eseménykezelő metódusával folyamatosan frissíti az ablak alján látható töltőcsíkot a generálás során. A

folyamat végén a felület egy oszlopdigramban jeleníti meg a backend által kiszámolt statisztikákat.

3.10.3. TrainContent osztály

A *TrainContent* vezérlő felelős a tanítási folyamat megjelenítéséért, illetve az egyes tanítási hiperparaméterek beállításáért. A tanítás során a *TensorBoard* alkalmazáson keresztül is nyomon tudjuk követni a pontosság és loss értékek alakulását. Az alábbi ábra tartalmazza az osztály osztálydiagramját:



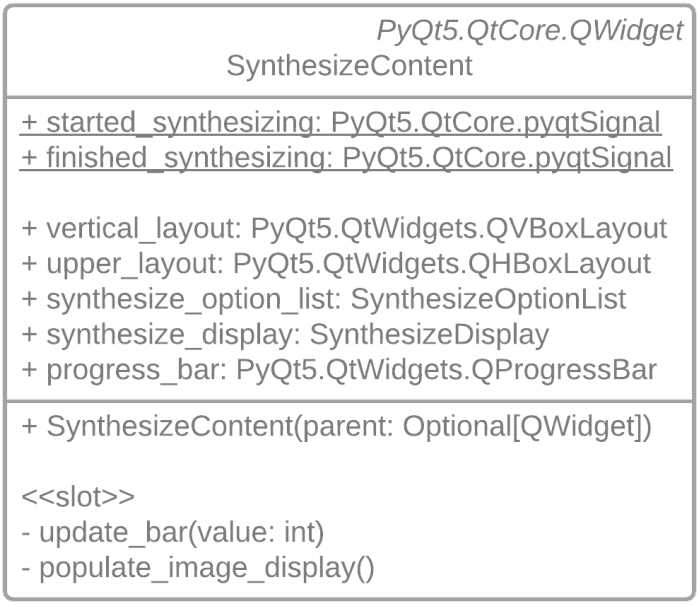
3.15. ábra. A *TrainContent* osztály osztálydiagramja. A vezérlő a *DataContent* osztályhoz hasonlóan eseményekkel kommunikál az őt magába foglaló *BaseContainer* vezérlővel. A vezérlő *TrainOptionList* adattagja a tanítási folyamat elindításakor egy külön alprocesszt indít el, amivel PIPE adatszerkezettel kommunikál. Az alprocessz log-jait a vezérlő esemény formájában adja tovább a *TrainContent* alvezérlőjének, amely a *TrainDisplay* adattagjának megjelenítőjére írja azokat. A tanítás végén az alprocessz a modell tanítása sikerességéről egy teszt adathalmazon való kiértékeléssel bizonyosodik meg, melyről a megjelenítő is tájékozódik.

A vezérlő a *DataContent* osztályhoz hasonlóan a *started_generating*, illetve a *finished_generating* eseményekkel kommunikál a szülő widget-évl, amely egy *BaseContainer* objektum. A vezérlő *train_option_list* nevű, *TrainOptionList* típusú adattagja a tanítási folyamat elindultakor egy alprocesszt indít el a *subprocess.Popen* osztály használatával, amivel PIPE adatszerkezet segítségével kommuni-

kál. Az alprocessz log-jait a *TrainOptionList* vezérlő esemény formájában továbbadja a *TrainContent* vezérlőnek, amely a *TrainDisplay* típusú adattagjának megjelenítőjére ír.

3.10.4. SynthesizeContent osztály

Végül a *SynthesizeContent* vezérlő teszi lehetővé a szintetizálási folyamat elindítását, állásának nyomon követését az állapotsáv segítségével, majd a szintetizált kompozíciók megtekintését a képmegjelenítő panelben. Alább ezen osztály osztálydiagramja látható:



3.16. ábra. A *SynthesizeContent* osztály osztálydiagramja. A vezérlő a *TrainContent* osztályhoz hasonlóan eseményekkel kommunikál az őt magába foglaló *BaseContainer* vezérlővel. A *DataContent* vezérlőhöz hasonlóan ez is rendelkezik egy töltőcsíkkal, amely a szintetizálási folyamat előrehaladását jelzi. A szintetizálási folyamat aszinkron módon folyik, az azzal való kommunikáció eseményekkel történik. A szintetizálás végeztével a *populate_image_display* metódus felelős a képmegjelenítő feltöltéséért a szintetizált programok vizualizációival.

A vezérlő a *TrainContent* osztályhoz hasonlóan a *started_generating*, illetve a *finished_generating* eseményekkel kommunikál a szülő widget-évl, amely egy

BaseContainer objektum. A szintetizálási folyamat a megfelelő paraméterek beállítása után aszinkron módon veszi kezdetét, azaz egy külön *QThread* objektum felelős a szintetizálás futtatásáért. Az ezzel való kommunikáció események formájában történik meg. A *populate_image_display* metódus végzi a szintetizálás végén a képmegjelenítő feltöltését a szintetizált függvénykompozíciók képeivel.

A függvények, metódusok, modulok és osztályok docstring-jeiből generált Sphinx dokumentáció angol nyelven további, részletes ismertetőként szolgál a forráskódról.

3.11. Tesztelési terv

Az alkalmazás alaposan tesztelve lett mind fekete doboz, mind fehér doboz tesztelési módszerekkel. A fehér doboz tesztelés a programkód írása során végig megvalósult, ugyanis az egységteszteket a forráskóddal párhuzamosan írtam. Így elértem, hogy az egyes függvények, metódusok corner case-eit átfogóan teszteljem.

3.11.1. Fehér doboz tesztelés

A fehér doboz tesztelést a fejlesztett szoftver backend komponenseire alkalmaztam. Mivel a dolgozatomban ismertetett *Function* és *Composition* osztályok képzik a backend jelentős részét és kivétel nélkül felhasználják a további backend modulokat és osztályokat, elég volt ezek átfogó tesztelését megterveznem és megvalósítanom. Fontos megemlíteni, hogy kivételt képez ezen modulok alól a beam search algoritmus, ugyanis azt a szoftver fejlesztése során nem én implementáltam, így dolgozatomban is annyit mutattam be belőle, amennyi az általam írt komponensek megértéséhez szükséges volt.

A fehér doboz tesztelésben felhasznált egységteszteket a *Pytest* és *Hypothesis* csomagok segítségével írtam meg, így röviden ismertetem ezeket.

A *Pytest* egy tesztelési keretrendszer a Python programozási nyelvhez, amely parancssorból futtatható. Segítségével a könnyen használható *assert* szintaxissal írhatunk egységteszteket anélkül, hogy ez gátolná a szoftver kifejezőkészségét. A szoftver nagy népszerűségnek örvend a Python fejlesztők körében: 2021. áprilisában 26,883,347-en töltötték le a csomagot. [21]

A *Hypothesis* a Haskell programozási nyelv felhasználóinak körében jók ismert Quickcheck csomaghoz hasonló Python könyvtár egységtesztek írásához. A *Hypothesis* ereje abban rejlik, hogy a programozó definiálhatja, hogy egy függvénynek vagy metódusnak milyen formális specifikációnak megfelelő inputra kell helyesen működnie, majd a könyvtár olyan random inputokat generál a teszteléshez, melyek eleget tesznek a specifikációnak, azonban igyekeznek a lehető legtöbb corner case-t lefedni (gyakran olyanokat is, amire a programozó a program megírásakor nem is gondolt). A *Hypothesis* nagy mértékben megkönnyíti a hibakeresést is, ugyanis ha egy hibát talál, akkor igyekszik az adott input bizonyos tulajdonságai mentén egy olyan nagyon egyszerű inputpéldát megtalálni, melyre a függvény még mindig rossz eredményt ad. Ezzel sokszor triviálissá teszik annak felismerését, hogy az inputok milyen *családjára* nem működik megfelelően a program.

Az egységteszteket az src.test_function modul tartalmazza. Ebben a *Function* osztály teszteléséhez 5 függvény található, melyek mindegyikéhez a *Hypothesis* könyvtár segítségével adtam meg a formális specifikációt, a maximális példaszámot, amennyin a *Hypothesis* kipróbálja az egységtesztet pedig 20-nak választottam meg. Az egyes tesztesetek:

1. *from_dict* osztályszintű metódus tesztelése az összes lehetséges felparaméterezett függvényre

2. *zip_with* függvénycsalád kiértékelésének tesztelése 1 bemeneti-kimeneti példával, a kapott eredmény összehasonlítása a Python beépített *eval* függvényének használatával

3. A *zip_with* függvénytől eltérő függvények kiértékelésének tesztelése 1 bemeneti-kimeneti példával, a kapott eredmény összehasonlítása a Python beépített *eval* függvényének használatával

4. *zip_with* függvénycsalád kiértékelésének tesztelése több bemeneti-kimeneti példával

5. A *zip_with* függvénytől eltérő összes függvény kiértékelésének tesztelése több bemeneti-kimeneti példával

A *Composition* osztály jóval nagyobb, mint a *Function*, így ennek egységtesztéséhez körültekintőbb tervezés volt szükséges. A tesztelés során több olyan hibát is felfedeztem, melyet a *Hypothesis* nélkül vagy nagyon nehezen, vagy nem találtam volna meg. A kérdéses kompozíció minden tesztesetben úgy lett megalkotva, hogy minél több corner case-t tartalmazzon. Az egyes tesztesetek:

1. Kompozíció levelei lekérdezésének ellenőrzése
2. Kompozíció string-gé alakító függvényének validálása
3. Kompozíció kiértékelésének tesztelése 1 input-output példával
4. Kompozíció kiértékelésének tesztelése több input-output példával
5. Kompozíció optimalizálásának ellenőrzése
6. Kompozíció alulról történő kiterjesztésének ellenőrzése
7. Kompozíció új id-kkel történő lemásolásának validálása
8. Kompozíció input-okkal való feltöltésének tesztelése

3.11.2. Fekete doboz tesztelés

A fekete doboz tesztelés azt tűzi ki célként, hogy a felhasználó a programkód ismerete nélkül, működés közben próbálja ki a szoftvert, figyelve az esetleges felmerülő problémákra. A fekete doboz tesztelés alatt kipróbált tesztjeimet egy táblázatba foglaltam:

Teszteset	*Hatás*
Generálás gomb megnyomása közvetlenül a program indítása után	Hibaüzenet megjelenése.
Helytelen érték beírása a kompozíciók száma mezőbe	Hibaüzenet megjelenése
Megfelelő érték beírása a kompozíciók száma mezőbe	Az érték eltárolásra kerül

Teszteset	*Hatás*
Mentési útvonal megadásának kihagyása generálásnál	Hibaüzenet megjelenése
Mentési útvonal megadásának megkezdése majd megszakítása és generálás gomb megnyomása	Hibaüzenet megjelenése
Mentési útvonal megfelelő megadása	Az útvonal eltárolásra kerül
Paraméterek helyes kitöltése generálásnál majd a generálás gomb megnyomása	A generálás elkezdődik
Generálás megszakítása	A generálás probléma nélkül véget ér
Adathalmaz útvonala megadásának kihagyása tanításnál	Hibaüzenet megjelenése
Adathalmaz útvonala megadásának megkezdése majd megszakítása és tanítás gomb megnyomása	Hibaüzenet megjelenése
Adathalmaz megfelelő megadása	Az útvonal eltárolásra kerül
Mentési útvonal megadásának kihagyása tanításnál	Hibaüzenet megjelenése
Mentési útvonal megadásának megkezdése majd megszakítása és tanítás gomb megnyomása	Hibaüzenet megjelenése
Mentési útvonal megfelelő megadása	Az útvonal eltárolásra kerül
TensorBoard szerver elindítása bármikor	A szerver megfelelően elindul
A tanítási paraméterek megfelelő kitöltése után a tanítás gomb megnyomása	A tanítás elindul, a konzolban log-ok jelennek meg

Teszteset	*Hatás*
Útvonalak kiválasztásának kihagyása szintetizálásnál	Hibaüzenet megjelenése
Útvonalak kiválasztásának megfelelő elvégzése szintetizálásnál	Az összes útvonal eltárolásra kerül
Helytelen input érték beírása szintetizálásnál	Hibaüzenet megjelenése
Helytelen output érték beírása szintetizálásnál	Hibaüzenet megjelenése
Megfelelő input és output értékek megadása szintetizálásnál	Az értékek validálás után eltárolásra kerülnek
Szintetizálás elkezdése az útvonalak megadása nélkül	Hibaüzenet megjelenése
Szintetizálás elkezdése az útvonalak megfelelő megadásával a teszt adathalmaz kiválasztása nélkül, de egzakt probléma megadásával	A szintetizálási folyamat kezdetét veszi
Szintetizálás elkezdése az útvonalak megfelelő megadásával a teszt adathalmaz kiválasztásával együtt, de egzakt probléma kihagyásával	A szintetizálási folyamat kezdetét veszi
Szintetizálás elkezdése az útvonalak megfelelő megadásával a teszt adathalmaz kiválasztásával együtt és egzakt probléma megadásával	A szintetizálási folyamat kezdetét veszi az egzakt problémán

3.2. táblázat. A fekete doboz tesztelés teszteseteinek táblázatos összefoglalása.

4. fejezet

Összegzés

Nagy örömmel tölt el, hogy a szakdolgozatom témájának választott programszintézis-rendszer implementálása a követelményeknek és célkitűzéseimnek megfelelően sikerült. A felvetett feladat egy olyan példaalapú programszintézis-rendszer megalkotása volt, mely a Gyarmathy Bálinttal és Czapp Ádámmal közösen fejlesztett rendszerünket kiterjeszti egy fastruktúrájú nyelvtanon történő alkalmazásra. A program szerves részét képző grafikus felület használata intuitív, így az MLforSE kutatócsoportban végzett munkámat is megkönnyítheti a továbbiakban.

A backend átfogó benchmarkolása során arra az eredményre jutottam társaimmal, hogy rendszerünk kifejezőereje jelentősen megnőtt a lineáris nyelvtanra korlátozott kezdeti verzióhoz képest, és a rendszer sok esetben meglepően jobb eredményeket ér el a teszthalmazokon, mint Zohar and Wolf [5] módszere. Teszthalmazokat saját rendszerünkkel és Zohar and Wolf rendszerével egyaránt generáltunk, hogy igazságos összehasonlítást végezzünk.

4.1. Fejlesztési lehetőségek

A modell kifejezetten jó általánosítóképességet mutatott a szakdolgozatban közölt nemlineáris nyelvtanon. Ez arra enged következtetést, hogy további, a *zip_with* függvényhez hasonló művelet bevezetése, illetve a *copy* függvény továbbfejlesztése jó irányt mutathat, ígéretes eredményekkel. A példagenerálási procedúrába egy cacheelési módszer (amely a nyalábkeresési algoritmusban már implementálásra került)

szintén sokat tudna faragni az adathalmazok generálási idején, amely megkönnyíthetné a csoport gyors kisérletezését.

4.2. Köszönetnyilvánítás

Elsősorban szeretném megköszönni családom minden tagjának végtelen támogatását, amely nemcsak a szakdolgozat megírása alatt, hanem teljes egyetemi tanulmányaim alatt jelen volt a mindennapjaimban. Szeretném továbbá megköszönni Dr. Pintér Balázsnak a mérhetetlen segítséget, amivel ellát amióta az MLforSE kutatócsoport tagja vagyok.

Irodalomjegyzék

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

[2] Pengcheng Yin et al. *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". In: *Automated Reasoning*. Ed. by Nicola Olivetti and Ashish Tiwari. Cham: Springer International Publishing, 2016, pp. 9–14. ISBN: 978-3-319-40229-1.

[4] Matej Balog et al. "Deepcoder: Learning to write programs". In: *arXiv preprint arXiv:1611.01989* (2016).

[5] Amit Zohar and Lior Wolf. "Automatic program synthesis of long programs with a learned garbage collector". In: *Advances in Neural Information Processing Systems*. 2018, pp. 2094–2103.

[6] Yu Feng et al. "Program Synthesis Using Conflict-Driven Learning". In: *SIGPLAN Not.* 53.4 (June 2018), 420–435. ISSN: 0362-1340. DOI: 10.1145/3296979.3192382. URL: https://doi.org/10.1145/3296979.3192382.

[7] Emilio Parisotto et al. *Neuro-Symbolic Program Synthesis*. 2016. arXiv: 1611.01855 [cs.AI].

[8] Woosuk Lee et al. "Accelerating search-based program synthesis using learned probabilistic models". In: *ACM SIGPLAN Notices* 53.4 (2018), pp. 436–449.

[9] Ashwin Kalyan et al. “Neural-guided deductive search for real-time program synthesis from examples”. In: *arXiv preprint arXiv:1804.01186* (2018).

[10] Noam Chomsky and Marcel P Schützenberger. “The algebraic theory of context-free languages”. In: *Studies in Logic and the Foundations of Mathematics*. Vol. 26. Elsevier, 1959, pp. 118–161.

[11] Bálint Gyarmathy et al. “Flexcoder: Practical Program Synthesis with Flexible Input Lengths and Expressive Lambda Functions”. In: (2021).

[12] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

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

[14] Wikipedia contributors. *Graphviz — Wikipedia, The Free Encyclopedia*. [Online; Hozzáférés dátuma: 2021.05.01.] 2020. URL: `https://en.wikipedia.org/wiki/Graphviz`.

[15] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[16] William Falcon et al. “PyTorch Lightning”. In: *GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning* 3 (2019).

[17] Thomas Viehmann Eli Stevens Luca Antiga. *Deep Learning with PyTorch*. Shelter Island, NY: Manning Publications Co., 2020. ISBN: 9781617295263.

[18] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: `10.1038/s41586-020-2649-2`. URL: `https://doi.org/10.1038/s41586-020-2649-2`.

[19] Wikipedia contributors. *PyQt — Wikipedia, The Free Encyclopedia*. [Online; Hozzáférés dátuma: 2021.05.01.] 2021. URL: `https://en.wikipedia.org/wiki/Graphviz`.

[20] Burcsi Péter. *ELTE IK, Diszkrét matematika I. előadás*. 2019.

[21] *PyPI Stats*. [Online; Hozzáférés dátuma: 2021.05.02.] 2021. URL: `https://pypistats.org/packages/pytest`.

Ábrák jegyzéke

Táblázatok jegyzéke