ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

NEUMANN JÁNOS
INFORMATIKAI KAR

SZAKDOLGOZAT

OE-NIK	Hallgató neve:	**Bognár Tamás**
2023	Hallgató törzskönyvi száma:	**T-006708/ FI12904/N**

Óbudai Egyetem
Neumann János Informatikai Kar
Kiberfizikai Rendszerek Intézet

SZAKDOLGOZAT FELADATLAP

Hallgató neve:	**Bognár Tamás**
Törzskönyvi száma:	T/006708/FI12904/N
Neptun kódja:	[redacted]

A dolgozat címe:

Gráf neurális hálózatok tanítása elosztott környezetben
Training of graph neural network in a distributed environment

Intézményi konzulens:	Farkas Attila
Külső konzulens:	
Beadási határidő:	2022. december 15.
A záróvizsga tárgyai:	Számítógép architektúrák Felhőszolgáltatási technológiák és IT biztonság specializáció

A feladat

A gépi tanulási és a mély tanulási módszerek jelentős előrelépést hoztak a gépi fordítás, beszédfelismerés és képfeldolgozás területén. Azonban az elmúlt években megnövekedett a gráfként reprezentált és tárolt adatok mennyisége, többek között a közösségi média elterjedésével. Ezen adatokban való minták felderítésére és feldolgozására jöttek létre a gráf neurális hálózatok és a fejlesztésükhöz szükséges keretrendszerek. Az ilyen típusú hálózatok tanítása is egy erőforrás igényes feladat, a rendelkezésre álló adatok mennyisége és a hálózatok komplexitása miatt. A hallgató feladata egy olyan referencia architektúra kifejlesztése, amely lehetőséget biztosít gráf neurális hálózatok elosztott környezetben való tanítására. A kidolgozandó referencia architektúrának, több nem funkcionális követelménynek is meg kell felelnie, mint például a hordozhatóság, automatikus kiépítés és skálázhatóság.

A dolgozatnak tartalmaznia kell:

- a referencia architektúra koncepció bemutatását,
- a gráf neurális hálózatok ismertetését a szakirodalom alapján,
- az elosztott mély tanulást támogató keretrendszerek összehasonlítását,
- a kiválasztott mély tanulási keretrendszer alapján a referencia architektúra tervezését,
- a referencia architektúra megvalósítását,
- az elkészült rendszer tesztelési dokumentációját,
- a továbbfejlesztési lehetőségek bemutatását.

ÓBUDAI EGYETEM * Neumann János Informatikai Kar * 1. Budapest

Ph.

.....
Dr. Fleiner Rita
intézetigazgató

A szakdolgozat elévülésének határideje: **2024. december 15.**
(OE TVSz 55.§ szerint)

A dolgozatot beadásra alkalmasnak tartom:

.....
külső konzulens

Tarkus Attila
.....
intézményi konzulens

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar

HALLGATÓI NYILATKOZAT

Alulírott hallgató kijelentem, hogy a szakdolgozat / diplomamunka saját munkám eredménye, a felhasznált szakirodalmat és eszközöket azonosíthatóan közöltem. Az elkészült szakdolgozatomban / diplomamunkámban található eredményeket az egyetem és a feladatot kiíró intézmény saját céljára térítés nélkül felhasználhatja.

Budapest, 2022. 12. 13.

Bozsik János

.....

hallgató aláírása

KONZULTÁCIÓS NAPLÓ

Hallgató neve: Neptun kód: Tagozat:

Bognár Tamás [redacted] Nappali

Telefon: Levelezési cím (pl: lakcím):

[redacted]

Szakdolgozat / Diplomamunka[1] címe magyarul:

Gráf neurális hálózatok tanítása elosztott környezetben

Szakdolgozat / Diplomamunka[2] címe angolul:

Training of graph neural network in a distributed environment

Intézményi konzulens: Külső konzulens:

Farkas Attila

Kérjük, hogy az adatokat nyomtatott nagybetűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	2022.03.02.	Szakdolgozat felépítése, tartalmi követelmények	Farkas Attila
2.	2022.04.13.	Elosztott gráf tanítási keretrendszerek használata	Farkas Attila
3.	2022.04.29.	Szakdolgozat formai követelményei	Farkas Attila
4.	2022.05.03.	Szakirodalomra való hivatkozások formája	Farkas Attila

A Konzultációs naplót összesen 4 alkalommal, az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a Szakdolgozat I. / Szakdolgozat II. (BSc) vagy Diplomamunka 1 / Diplomamunka 2 / Diplomamunka 3 / Diplomamunka 4[3] tantárgy követelményét teljesítette, beszámolóra / védésre[4] bocsátható.

Budapest, 2022.05.11.

Farkas Attila
.....
intézményi konzulens

[1] Megfelelő aláhúzandó!
[2] Megfelelő aláhúzandó!
[3] Megfelelő aláhúzandó!
[4] Megfelelő aláhúzandó!

KONZULTÁCIÓS NAPLÓ

Hallgató neve: Neptun kód: Tagozat:

Bognár Tamás [redacted] Nappali

Telefon: Levelezési cím (pl: lakcím):

[redacted]

Szakdolgozat / Diplomamunka[1] címe magyarul:

Gráf neurális hálózatok tanítása elosztott környezetben

Szakdolgozat / Diplomamunka[2] címe angolul:

Training of graph neural network in a distributed environment

Intézményi konzulens: Külső konzulens:

Farkas Attila

Kérjük, hogy az adatokat nyomtatott nagybetűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	2022.09.08.	Elosztott környezetet biztosító felhő elérése	Farkas Attila
2.	2022.10.07.	Tanító szkript optimalizálásának lehetőségei	Farkas Attila
3.	2022.11.09.	Implementáció és tesztelés felépítése	Farkas Attila
4.	2022.11.25.	PPT diasor formai követelményei	Farkas Attila

A Konzultációs naplót összesen 4 alkalommal, az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a Szakdolgozat I. / Szakdolgozat II. (BSc) vagy Diplomamunka 1 / Diplomamunka 2 / Diplomamunka 3 / Diplomamunka 4[3] tantárgy követelményét teljesítette, beszámolóra / védésre[4] bocsátható.

Budapest, 2022.11.25.

Farkas Attila
.....
intézményi konzulens

[1] Megfelelő aláhúzandó!
[2] Megfelelő aláhúzandó!
[3] Megfelelő aláhúzandó!
[4] Megfelelő aláhúzandó!

TARTALOMJEGYZÉK

1. BEVEZETÉS

Jelentős előrehaladást és nagy sikereket értek el a gépi tanulási és a mély tanulási módszerek olyan számítástechnikai területeken, mint például a képfeldolgozás, gépi látás vagy akár a beszédfelismerés. Az utóbbi években azonban egyre inkább megnövekedett a gráfként reprezentált, illetve tárolt adatok mennyisége részben a közösségi média elterjedésének eredményeképp. Ebből kifolyólag volt szükség olyan gráf neurális hálózatokra és a fejlesztésükhöz szükséges keretrendszerekre, amelyek az ilyen struktúrájú adatokban található minták felderítésére, feldolgozására és elemzésére szolgálnak. A gráf neurális hálózatoknak köszönhetően a kutatók olyan területeken is kiemelkedő eredményeket értek el, mint például a természetes nyelv feldolgozás vagy a bioinformatika. Azonban az ilyen típusú hálózatok tanítása egy erőforrás igényes feladat, az adathalmazok mérete és a hálózatok komplexitása miatt. Ahhoz, hogy egy ilyen tanítási folyamat optimálisabb legyen például a végrehajtáshoz szükséges idő szempontjából valamilyen elosztott stratégiát követő keretrendszert célszerű alkalmazni. Az én dolgozatomnak az a célja, hogy egy ilyen keretrendszer segítségével egy olyan referencia architektúrák biztosítsak, amely segítségével valamilyen felhő platformon a megfelelő erőforrások felhasználásával egy ilyen gráf neurális hálózat tanítása elosztott módon kivitelezhető legyen.

2. REFERENCIA ARCHITEKTÚRA KONCEPCIÓ

Egy referencia architektúra bizonyos szempontból egy PaaS szolgáltatásnak felel meg, továbbá követi az Infrastructure-as-Code (IaC) megvalósítást, ami leíró fájlok segítségével komplex infrastruktúrák kiépítést teszik lehetővé valamilyen felhő platformon. Ehhez használhatunk különböző konfigurációs menedzsment és orkesztrációs eszközöket egyaránt, amelyek képesek felépíteni magukat a virtuális gépeket, majd ezeken a gépeken telepíteni tudják a szükséges szolgáltatásokat. Ezeken felül a referencia architektúra kitér még további, nem funkcionális követelmények teljesítésére is, mint például a skálázhatóság, biztonsági követelmények és magas rendelkezésre állás biztosítása, illetve beilleszthető valamilyen CI/CD pipeline-ba, így a folyamatos teszteléssel is biztosítható a megfelelő működés. [1][2]

2.1. Orkesztrátorok

Az orkesztráció gyakorlatilag az egyes számítógépes rendszerek és szoftverek automatizált konfigurációját, menedzsmentjét és koordinációját foglalja magában. Az egyik elterjedtebb ilyen konténer orkesztrátor eszköz a Kubernetes, mely különböző konténerizált alkalmazások automatizált indítását, skálázását és menedzsmentjét végzi. [3] Egy ilyen másik eszköz a Portainer, ami lehetőséget ad a felhasználóknak, hogy egy futó szolgáltatást újra lehessen konfigurálni például a környezeti változók módosításával. [4] A felhő orkesztrátorok egy adott felhőben biztosítják az egyes erőforrások automatizált létrehozását. Ilyen eszköz például a Terraform, amely olyan szolgáltatókat is támogat, mint a Microsoft Azure, Amazon AWS vagy akár az OpenStack. [51]

2.2 Konfigurációs menedzsment eszközök

A virtuális gépeken lévő szoftverekben történő változások eszközölésére és nyomon követesére úgynevezett konfigurációs menedzsment eszközöket használunk. Egyik ilyen az Ansible, amely egy nyílt forráskódú, deklaratív nyelvet használó IaC eszköz. Ez egy SSH kapcsolaton keresztül végzi el az egyes gépeken a szükséges műveleteket. A másik példa, amiről érdemes szót ejteni a Puppet, ami eltérően az Ansible-től egy master-slave architektúrát használ, így az egyes gépeknek kell letölteni a masterről a konfigurációkat. [5]

2.3. CI/CD megoldások

Mivel az agilis fejlesztési technikák egyre fontosabbá válnak, ezért szükség van egy olyan rendszerre, ami automatizált módon tudja tesztelni az adott alkalmazást, így gyorsítva a fejlesztését. A Jenkins egy nyílt forráskódú automatizálást nyújtó szerver, amin a tényleges alkalmazás építése (build) és tesztelése folyik. Egy másik hasonló megoldást kínál a CircleCI, ami a Jenkinsszel ellentétben már nem ingyenesen használható. [6]

3. A GÉPI TANULÁS

A gépi tanulás modern kori értelemben vett eredete Frank Rosenblatt nevéhez köthető, aki a Cornell Egyetemen szerzett doktori címet.[1] Ő volt az, aki azon elképzelés mentén, hogy hogyan működhet az emberi idegrendszer, megalapított egy csoportot, mely tagjai építettek egy olyan gépet, ami képes volt felismerni az ABC betűit. Ezt a gépet az alkotó „perceptron”-nak nevezte el. Mind diszkrét, mind analóg jelekkel is dolgozott, illetve volt egy olyan része is, mely egy adott küszöbértékhez viszonyítva analóg jeleket tudott diszkrét jelekké alakítani. [7][8]

3.1. A gépi tanulás korai évei

Az 1960-as évek elejétől egészen az 1990-es évekig a gépi tanulás mint kutatási terület számos felfutást, illetve letörést megélt mire eljutott a 21. század első feléhez. Ez az időszak ugyanis egy fordulópontnak mutatkozott a gépi tanulás történelmében. Ez nagyrészt három, egymással feltűnően összhangban lévő új trendnek volt köszönhető. Az egyik ilyen a Big Data megjelenése volt. Olyan nagy mennyiségű adat keletkezett már akkoriban is, hogy szükség volt új megközelítések felé mozdulni annak érdekében, hogy továbbra is képesek legyenek ezen adatok feldolgozására. Így a gépi tanulás már nem csak a tudományos kutatások miatt, hanem praktikus okok miatt is újra szerepet kaphatott.[8]

A második új irányként egy olyan igény vált népszerűvé, hogy a páhuzamosan végzett számítások és az ezekhez szükséges memória költségeit csökkentsék. Ez a trend igazán 2004-ben csúcsosodott ki, amikor a Google bemutatta az új MapReduce névre hallgató technológiáját. Nemsokkal ezt követően - 2006-ban [9] - megjelent ennek a technológiának a nyílt forráskódú megfelelője, a Hadoop. E két modell tette lehetővé hatalmas mennyiségű adat feldolgozásának elosztását egyszerűbb feldolgozó egységek között. Ezekkel az újításokkal egyidőben az Nvidia is nagy áttörést ért el a videokártyák piacán, ugyanis azon kártyáknak köszönhetően, melyek gépi tanulási célokra is alkalmazhatók, monopolhelyzetbe került a cég. Ezekben az időkben a memóriák fogyasztói ára is nagyot csökkent. Ez lehetőséget teremtett arra, hogy olyan, nagy mennyiségű adattal dolgozzanak, amit a memóriában tudnak tárolni. Ennek eredményeképp számos új adatbázis is megjelent, mint például a NoSQL. Végül 2014-ben elérhetővé vált az Apache által fejlesztett Spark nevű keretrendszer, minek segítségével elosztottan lehet végezni a strukturáltalan vagy gyengén strukturált adatok feldolgozását, ami elősegítette a különböző gépi tanuláson alapuló algoritmusok megjelenését. [8]

Ezeket a korábban említett trendeket szorosan követte új algoritmusok kifejlesztése, melyek a mély tanulás számára jelentettek nagy előrelépést. Ezen új fejlesztések során a Frank Roseblatt nevéhez fűződő perceptron ötletét vették alapul, illetve fejlesztették magasabb szintekre. Sok év kutatás után a többrétegű neurális hálózatok terén végre

megfogalmazódott egy koncepció, melyet mély neurális hálózatoknak neveztek el a kutatók. [8]

Már a 2010-es évek közepére a mély neurális hálózatok fogalma és használati lehetőségi jelentősen beépültek a köztudatba, így elkerülhetetlen volt, hogy valaki még mélyebb területeit is feltárja ennek az új technológiai fejlesztésnek. E téren Geoffrey Hinton, egy brit származású kutató volt az, aki felvette a vezető szerepét. 2006 óta számos publikációt tárt a világ elé kollégáival együttműködve a mély neurális hálózatok tudományos területén. [8]

3.2. Gépi tanulás napjainkban

Napjainkban a gépi tanulás által kínált lehetőségek, már számos egyéb tudományos területre is beszivárogtak. Az egyik ilyen terület az orvosi diagnosztika. Az egyik legfőbb alkalmazása a gépi tanulásnak az olyan, különböző kórok és betegségek azonosítása, illetve diagnózisa, amiket eddig csak nagyon nehezen lehetett felismerni, illetve megfelelő kezelést nyújtani ellenük. Ezen betegségek között lehet szó rákról, amit kimondottan nehéz azonosítani a korai fázisaiban vagy más, örökléstani eredetre visszavezethető betegségekről is. Egy másik fontos felhasználása a gépi tanulásnak az orvostudományban az újabb gyógyszerek kifejlesztése és gyártása során mutatkozik meg. Itt ugyanis kimondottan nagy szerepet kap ez az új technológia a gyógyszerfejlesztés korai fázisaiban. Ez magában foglalja az olyan kutatási területeket is, mint például az újgenerációs szekvenálás.[10][11]

Az előzőtől egy teljesen eltérő ágazatban, a közúti forgalom irányításában, elemzésében is nagy szerepe van a gépi tanulásnak, aminek segítségével előrejelzéseket lehet készíteni a forgalom sűrűségének alakulásáról az utakon. Az emberek egyre nagyobb számban használnak manapság GPS-t. Ezek az eszközök számos adatot továbbítanak egy adott szolgáltatónak menet közben. Ilyen például az aktuális sebesség vagy a pontos hely. Ezen adatok segítségével egy térképet tudnak felállítani a jelenlegi forgalmi helyzetről. Az igazi probléma azonban abból fakad, hogy bár sokan használnak GPS-t még mindig nem áll rendelkezésre elég adat ahhoz, hogy pusztán ezekből a valós idejű információkból dolgozzanak. Ebből kifolyólag a gépi tanulás igazi ereje abban nyilvánul meg, hogy egy ilyen rendszer képes ebből a viszonylag kisméretű adathalmazból úgy extrapolálni, hogy aránylag nagy pontossággal tudja előrevetíteni a torlódási pontokat egy régióban. [12]

3.3 Mély tanulás

A mély tanulás (deep learning) a már korábban ismertetett gépi tanulásnak a részhalmaza. A gépi tanulásnak ez az új formája elméleti szinten már az 1980-as években megfogalmazódott, azonban több ok miatt is csak az elmúlt években bizonyult hasznosnak. Ennek az egyik legfőbb oka, hogy nagy mennyiségű címkézett adatra van szükség ahhoz, hogy egy ilyen rendszert képesek legyenek megfelelő módon betanítani. Másrészről pedig jóval nagyobb számítási kapacitásra van szükség, amit a korszerű, párhuzamos architektúrákra épülő grafikus feldolgozó egységek (GPU) fejlesztése hozott

el. Továbbá ezen erőforrások összefűzése egy fürtbe (cluster), még inkább lerövidítheti a tanítási időt akár pár órára is.

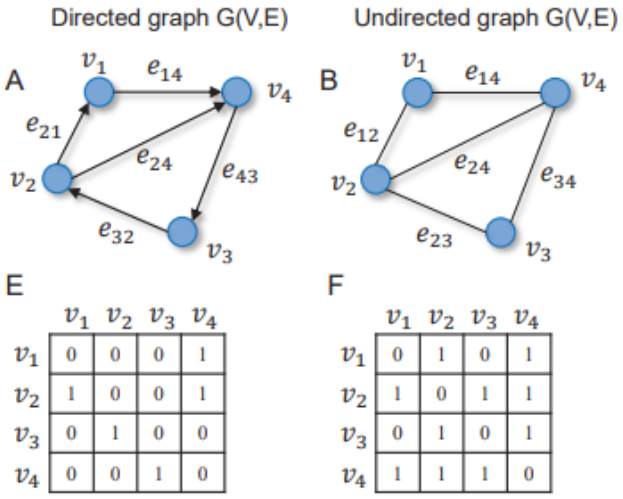
A legtöbb mély tanulási eljárás valamilyen neurális hálózat architektúrát használ, ebből kifolyólag ezeket a modelleket gyakran nevezik mély neurális hálózatoknak. Legfőképpen a rejtett rétegek száma különbözteti meg a hagyományos neurális hálózatoktól a mély neurális hálózatokat. Míg az előbbi általában 2-3 rétegből áll, addig a másik akár százas nagyságrendű rétegszámmal is dolgozhat. Egy másik szintén fontos tényező, amivel a mély tanulás kibővíti a gépi tanulást az, hogy itt nem kell manuálisan kinyerni az adathalmaz releváns részeit, mert a különböző mély tanulási megvalósítások ezt automatikusan végzik. További előnye a korábbi megközelítésekkel szemben, hogy a mély neurális hálózatok pontossága gyakran egyenesen arányos módon növekszik a bemeneti adathalmaz nagyságával. A mély tanulási rendszereknek egy új ága a gráf neurális hálózatok. [13][14][15]

4. GRÁF NEURÁLIS HÁLÓZATOK

4.1. Gráf adatszerkezet

Számos alkalmazási területen az adat akár gráfként is ábrázolható. Ilyen terület például a szoftverfejlesztés, vagy a képek elemzése. A gráfok legegyszerűbb típusai csak egyszerű csomópontokat és éleket tartalmaznak. A legtöbb alkalmazási esetben azonban az információ általában sokkal összetettebb gráf struktúrában jelenik meg. Ezek az összetettebb struktúrák lehetnek akár fák, ciklikus gráfok vagy aciklikus gráfok. A számítástudományban a gráfok olyan adatszerkezetek, melyek a matematikában azon belül is a gráfelmélet területén megismert irányított, illetve irányítatlan gráfok koncepcióját alkalmazzák. Egy gráf struktúra egy végessok csúcspontot tartalmazó halmazból és egy másik halmazból áll, ami vagy rendezett formában ábrázolja ezeket a csúcspont párokat vagy rendezetlen módon. Ha rendezetten vannak ábrázolva a párok, akkor irányított gráfról, ellenkező esetben pedig irányítatlan gráfról beszélhetünk. Ezeket a csúcspont párokat éleknek nevezzük. Nem ritka, hogy ezeken az éleken valamilyen egyéb információt is elhelyeznek, például egy numerikus attribútumot. Gyakran ábrázolják a gráfokat más adatszerkezetek segítségével, például mátrixokkal. Ilyen ábrázolási móddal három lehetőség áll rendelkezésünkre: a szomszédsági mátrix, az illeszkedési mátrix, illetve a szomszédsági lista vagy más néven éllista. [16]

A szomszédsági mátrix, ahogy 4.1-es ábrán is látható egy kétdimenziós mátrix, melyben a sorok reprezentálják az egyes csúcspont párok forrás csúcsát, míg az oszlopok ábrázolják e párok cél csúcsait. Fontos megjegyezni, hogy az éleken és csúcspontokon elhelyezett egyéb adatokat valamilyen külső helyen kell tárolni, habár lehetőségünk van arra, hogy egy adott indexen egy csúcspont pár között húzódó útvonal költségét feltüntessük. Ha egy irányítatlan gráfot ábrázolunk egy ilyen mátrix struktúrában, akkor egy szimmetrikus mátrixot kapunk. Ez azért lényeges, mert így elegendő csak egy felsőháromszög mátrixot tárolni, ami az eredeti mátrixnak csak a felét fogja foglalni



E	v_1	v_2	v_3	v_4
v_1	0	0	0	1
v_2	1	0	0	1
v_3	0	1	0	0
v_4	0	0	1	0

F	v_1	v_2	v_3	v_4
v_1	0	1	0	1
v_2	1	0	1	1
v_3	0	1	0	1
v_4	1	1	1	0

4.1. ábra - Szomszédsági mátrixok [19]

tárhely szempontjából, tehát ez egy jóval optimálisabb megoldás. Az illeszkedési mátrix szintén egy kétdimenziós mátrix. A sorok ebben az esetben a csúcspontokat jelentik, míg az oszlopok az egyes éleket. A mátrixban található adatok magát a csúcspontok és élek közötti relációt ábrázolják. [17][18]

Egy szomszédsági lista esetében a csúcspontokat objektumokként tároljuk. Ezek az objektumok pedig egy-egy listát tartalmaznak, melyekben a vele szomszédos csúcspontok vagy élek vannak definiálva. Az előzőkkel ellentétben ez a módszer lehetőséget biztosít arra, hogy az egyes csúcspontokon is egyéb, további adatokat helyezzünk el. A leggyakoribb alkalmazási területe egy ilyen listának az, hogy egy tetszőlegesen kiválasztott csúcspontnak megkapjuk az össze vele szomszédos csúcspontokat vagy éleket. Erre a célra ez a struktúra különösen hatékony megoldást nyújt, hiszen egy ilyen művelet elvégzéséhez szükséges idő a lekérdezni kívánt csúcspontok számával egyenesen arányos. Az előző struktúrákhoz hasonlóan itt is lehetséges azt vizsgálni, hogy két tetszőlegesen kiválasztott csúcspont között vezet-e él. Ennek a műveletnek az időkomplexitása függ attól, hogy rendezett módon vannak-e tárolva a csúcspontok vagy sem. Ha rendezett módon vannak, akkor lehetőségünk van logaritmikus keresést használni, de még így sem olyan hatékony ez a struktúra, mint a mátrixok. [16]

4.2. Gráfok átalakítása

A gráf neurális hálózatok (GNN) olyan, gépi tanuláshoz köthető algoritmusok, melyek segítségével fontos információkat tudunk kinyerni egy gráfból, melyek segítségével következtetéseket lehet készíteni. Ahogyan azt korábban leírtam, a gráfok csúcspontokból és élekből állnak. Ennek okán például egy közösségi hálót felírhatunk úgy, hogy az egyes csúcspontok reprezentálják a felhasználókat és a hozzájuk tartozó egyéb tulajdonságaikat, az élek pedig a felhasználók közötti viszonyt. A gráf struktúra azonban korántsem annyira alkalmas arra, hogy gépi tanulási célokra használjuk. A neurális hálózatok ugyanis az adatot egy egységes formában várják bemenetükként. A gráfokat definiálhatjuk különböző struktúrákban, illetve különböző méretekkel rendelkezhetnek, amik nem feltétlenül illeszkednek ahhoz a jól definiált bemenethez, amit a neurális hálók elvárnak. Ezen felül a gráfoknak vannak további különleges tulajdonságaik, amik az elvárt bemenettől eltérnek. Ilyen tulajdonság például az, hogy ha a csomópontok sorrendjét és pozícióját megváltoztatjuk, nem változik a struktúra egészen addig, míg a közöttük lévő reláció megmarad. Annak érdekében, hogy a gráfok alkalmasak legyenek arra, hogy mély tanulási algoritmusokat valósítsunk meg velük, transzformálni kell őket egy olyan formátumba, amit már fel tud dolgozni egy neurális háló. A formátum típusa annak függvényében változhat, hogy milyen a kiinduló gráf. Nagy általánosságban azonban elmondható, hogy az egy jó megoldás lehet, ha a gráfokat mátrixokként ábrázoljuk. Ha visszatérünk az előbb említett példára, akkor felhasználók tulajdonságait tartalmazó csúcspontokat egy táblázatként is fel lehet írni. Egy ilyen tábla ahol sorokba rendezett módon tároljuk az egy felhasználóhoz tartozó adatokat tipikusan

egy hagyományos neurális háló bemeneti forrása lehetne. Ezen túlmenően azonban a gráf neurális hálók egyéb információkból is tanulhatnak. Az éleket is ábrázolhatjuk hasonló táblázatos módon, ahol az egyes sorokba a különböző felhasználók közötti viszony egyéb paramétereit írhatjuk le. Így el is jutottunk az előző fejezetben említett szomszédsági mátrixszal való ábrázoláshoz. [20]

4.3. Gráfok beágyazása

Gráf neurális hálózatokat ugyan olyan módon létrehozhatunk, mint egy bármilyen másik neurális hálót. Használhatunk teljesen kapcsolt rétegeket vagy akár konvolúciós rétegeket is. A használt típus és a rétegek száma lényegében csak a gráf típusától, és a komplexitásától függ. Miután a GNN megkapta a gráfból készült formázott adatokat, egy numerikus értékeket tartalmazó vektort állít elő, ami a csomópontokból és a köztük lévő élekből kinyert releváns információkból áll. Ezt a folyamatot gráf beágyazásnak (graph embedding) is nevezik. Ez a mechanizmus úgy működik, hogy mikor a gráf adatokat megkapta a GNN, akkor az egyes csomópontok tulajdonságait a velük szomszédos csomópontokéval összefésüli. Ezt üzenettovábbítási (message passing) eljárásnak is szokták nevezni. Ha a GNN több rétegből épül fel, akkor minden rétegben megismétlődik ez a folyamat annyi különbséggel, hogy az aktuális rétegben kapott eredmények az előző réteg által kiszámolt értékekhez hozzáfűződnek. [19]

Ismét a közösségi hálós példát említve, egy több rétegű gráf neurális háló első rétege az aktuális felhasználó adatait kombinálná a vele közös ismerőseiével. Ezt követően a következő réteg a felhasználó barátai barátainak az adatait aggregálná az előzőekben kapott értékkel és így tovább. Végezetül a kimeneti réteg létrehozza a végleges vektor reprezentációját az egyes csomópontoknak és azok szomszédjainak.

4.4 Gráf neurális hálózatok alkalmazásai

A gráf neurális hálózatok alkalmazási területei évről évre bővülnek. Ezek közül ebben az alfejezetben, néhány fontos és elterjedt felhasználási módot fogok bemutatni.

4.4.1. Természetes nyelv feldolgozás

Gyakori felhasználása a gráf neurális hálózatoknak a természetes nyelv feldolgozás (NLP) területén a különböző szövegek osztályozása. A GNN-ek az egyes szavak vagy dokumentumok közötti relációk felhasználásával következtetnek a különböző címkékre, amikkel ezeket a szavakat vagy dokumentumokat kategorizálja. Az NLP-nek számos hasznosítási módja van, ezek közül a fontosabbak a spam levelek detektálása, fordítógépek pontosságának növelése, illetve nagy mennyiségű szövegből rövidebb összefoglalók előállítása. [21]

4.4.2. Gépi látás

Az elmúlt években nagyon sok konvolúciós neurális hálózatokat használó megoldás született arra, hogy képeken különböző objektumokat legyenek képesek felismerni a

gépek. Azonban eddig nem tudták azt megállapítani, hogy e tárgyak között milyen reláció van. Ilyen problémák esetén mutatkozik meg igazán a GNN-ek sokoldalúsága, ugyanis miután valamilyen módon felismertük a tárgyakat egy képen, ezeket az adatokat átadhatjuk egy GNN-nek, hogy az egyes objektumok közötti kapcsolatokra egy predikciót készítsen. A kimeneti eredmény egy olyan generált gráf, mely lemodellezi a relációkat. Egy másik érdekes alkalmazás a gépi látáson belül, különböző képek generálása egy gráf alapú leírás alapján. Ez tulajdonképpen a korábban említett folyamat megfordításán alapszik. A hagyományos, elterjedtebb módja a képgenerálásnak eddig szöveg alapú volt akár egy GAN vagy autoencoder rendszer segítségével. A GNN alapú megközelítéssel azonban sokkal több információ áll rendelkezésre a kép struktúráját illetően. [22]

4.4.3. Bioinformatika

Különösen fontos szerepe van a gráf neurális hálózatoknak a különböző fehérjék funkcióira és struktúráira irányuló kutatások során, ugyanis a legtöbb betegség valamilyen protein diszfunkcióval szoros összefüggésben álló eredetre vezethető vissza. Jelentős előrehaladás történt az elmúlt időkben a különböző proteinek struktúrájának predikcióját illetően. A legpontosabb előrejelzést adó modellek képesek molekuláris szinten elemezni a biológiai mechanizmusait egy fehérjének. Ezen eredmények felhasználása nagyban hozzájárul az újabb gyógyszerek fejlesztéséhez, és a biokémiai kutatási területekre is nagy hatással van. [10]

5. ELOSZTOTT MÉLY TANULÁST TÁMOGATÓ KERETRENDSZEREK

Számos elosztott mély tanulást támogató keretrendszert fejlesztettek az elmúlt években. Ezek közül a legelterjedtebb megoldásokról, illetve ezek összehasonlításáról lesz szó ebben a fejezetben.

5.1. TensorFlow distributed

A TensorFlow egy nyílt forráskódú csomag, melyet a Google fejlesztett elsősorban mély tanulási célokra. Eredetileg nagy számokkal történő számításokra tervezték anélkül, hogy a mély tanulás lehetőségeit figyelembe vették volna. Azonban később erre is alkalmasnak nyilvánult, így a Google elérhetővé tette mindenki számára. A TensorFlow úgynevezett adatfolyam gráfok alapján működik. Ebből kifolyólag, mivel a végrehajtó mechanizmus gráf alapú, sokkal könnyebb egy TensorFlow kódot egy elosztott módon több számítógép GPU-ját felhasználva végrehajtani. Ez a keretrendszer a Kerast használja magas szintű API-ként a Pythonhoz, de akár C++ nyelven is lehetőségünk van használni. A mély tanuláson alapuló alkalmazások nagyon komplikáltak részben azért, mert a tanulási folyamat általában nagy számítási teljesítményt igényel. Éppen ezért gyakran sok ideig is tarthat a különböző iteratív mátrix műveletek miatt, ha ezeket a feladatokat a CPU-n végezzük. A grafikus kártyákat eredetileg a számítógépes játékok miatt fejlesztették ki, ahol minden egyes képkockának nagy felbontásban kell megjelennie. Azonban az elmúlt években egyre nagyobb teret nyert a felhasználásuk a mély tanulás területén is, mert ezekkel a feldolgozó egységekkel, jóval rövidebb ideig tart egy tanulási folyamat. Ennek a trendnek köszönhetően a TensorFlow nem csak CPU-kat hanem GPU-kat is támogat. 2016 májusában a Google bejelentette a Tensor feldolgozó egységét (TPU), ami egy ASIC vagyis egy adott felhasználási módra optimalizált integrált áramkör. Ez kimondottan gépi és mély tanulási célokra lett kifejlesztve, ezen felül pedig konkrétan a TensorFlow-hoz. A Google állításai szerint nagyságrendekkel jobban optimalizált a teljesítmény és a felhasznált energia aránya. [23] A TensorFlow továbbá képes arra, hogy egy elosztott környezetben hajtsa végre a tanítási folyamatokat, így tovább növelve a hatékonyságot. A konvolúciós neurális hálózatok mellett támogat még RNN alapú hálózatokat is, mely kifejezetten szekvenciális adatok modellezésére használatosak. Azonban egyéb kiegészítő csomagok segítségével gráf neurális hálózatokkal is lehet dolgozni a TensorFlow platformon is. [24][25][26]

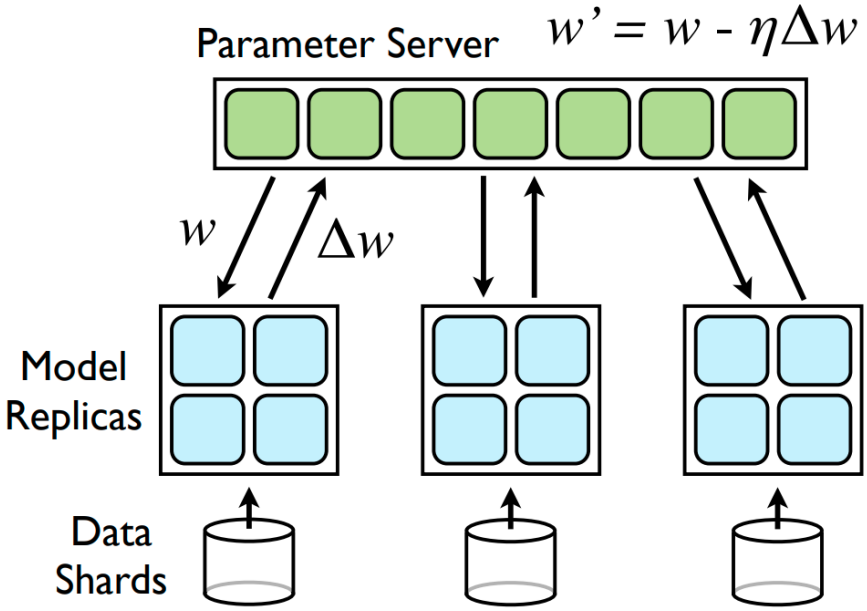
Ahhoz, hogy elosztott módon tudjuk végezni a tanítást a TensorFlow egy olyan API-t kínál, melynek segítségével több GPU-ra, számítógépre vagy több TPU-ra is kiterjeszthető az elvégzendő feladat. A TensorFlow által nyújtott számos elosztott stratégiát követő mintákat két fő részre lehet bontani, aszerint, hogy szinkron vagy aszinkron módon képesek az adott modellekkel elosztott tanítást megvalósítani. A szinkronizált tanítás esetén, minden dolgozó gép egymással szinkronban a bemeneti adat egy részével foglalkozik csak, majd a számított értékeket minden lépés végén egymás

között aggregálják. Ezzel szemben az aszinkron módon működő modelleknél minden dolgozó gép egymástól függetlenül dolgozza fel a bemeneti adatot, majd az egyes folyamatok végén frissítik a változókat. Általában egy szinkron tanítást támogató minta az all-reduce, aszinkron tanítást támogató rendszer pedig a paraméter szerver architektúrára épül. Az egyes elosztott stratégiákra jellemző tényező még, hogy milyen fizikai eszközökön tudják biztosítani a modellek skálázhatóságát. Tehát például ad-e megfelelő skálázási lehetőséget egy adott stratégia olyan infrastruktúrák esetében, ahol több GPU van ugyan, de ezek egyetlen számítógépben, vagy esetleg olyan helyzetben, amikor sok számítógép áll rendelkezésünkre, de ezek nem feltétlen tartalmaznak GPU-kat. [27]

A MirroredStrategy a TensorFlow által nyújtott egyik stratégia, mely a szinkronizált elosztott tanítási mintát követi. Ez a módszer olyan egygépes környezetben működik ahol több GPU található. Ez tulajdonképpen annyi másolatot készít a modellről ahány GPU van a rendszerben, ezáltal minden változó a modellben úgymond tükrözve van a grafikus kártyákon. A változókat úgy tartja szinkronban, hogy egyforma frissítéseket alkalmaz rajtuk. Azt, hogy az egyes GPU-k megkapják ezeket a frissítéseket, valamilyen all-reduce algoritmus segítségével hajtja végre. Ez az algoritmus az összes tensort úgy aggregálja, hogy összeadja, majd minden eszköz számára elérhetővé teszi őket. Nagyon hatékonyan képes működni egy all-reduce algoritmus hiszen, a szinkronizációs folyamat által generált egyéb adatok mennyiségét is csökkenti. Ez a stratégia alapértelmezetten az NCCL-t használja az all-reduce implementációjaként, de lehetőségünk van akár egy sajátot is megadni. Egy részben másik stratégia a TPUStrategy, ami annyiban tér el a korábban bemutatottól, hogy a TPU-k a saját all-reduce algoritmusukat és egyéb mechanizmusukat használják ahhoz, hogy több TPU között elosztott környezetet hozzanak létre. [28]

A MultiWorkerMirroredStrategy egy szintén szinkronizált elosztott tanítási stratégiát követ. Hasonlóan működik, mint a MirroredStrategy annyi különbséggel, hogy itt lehetőségünk van olyan infrastruktúrát használni, amiben több gép is található melyekben grafikus kártya is lehet. Két lehetőséget kínál az eszközök közötti kommunikáció megvalósítására. Az egyik egy RPC (Remote Process Call) alapú, mely egyaránt támogat CPU-kat és GPU-kat is, míg a másik a már korábban említett NCCL-t használja, ami csak grafikus kártyát támogat. Az egyik legfőbb különbség az egygépes rendszerrel szemben, hogy itt be kell állítani, hogy mely gépek képzik az infrastruktúrát. Ezt környezeti változók segítségével lehet megadni. [29]

Olyan stratégiát is támogat a TensorFlow, melynek segítségével hasonlóan az előző megközelítéshez több számítógépen lehet a tanítást végezni. Egy ilyen környezet az 5.1-es ábrán is látható dolgozó gépekből és paraméter szerverekből épül fel. Az egyes változók a modellek másolataiban (model replicas) ezeken a szervereken jönnek létre és a többi gép ezekről olvassa ki az adatokat, majd frissítik az új adatokkal az egyes lépések után. Alapértelmezetten a dolgozó gépek egymástól függetlenül végzik a feladatukat, ezért az ilyen stratégiát követő rendszereket aszinkron rendszernek is nevezik. [30][31]



5.1. ábra - Paraméter szerver stratégia [32]

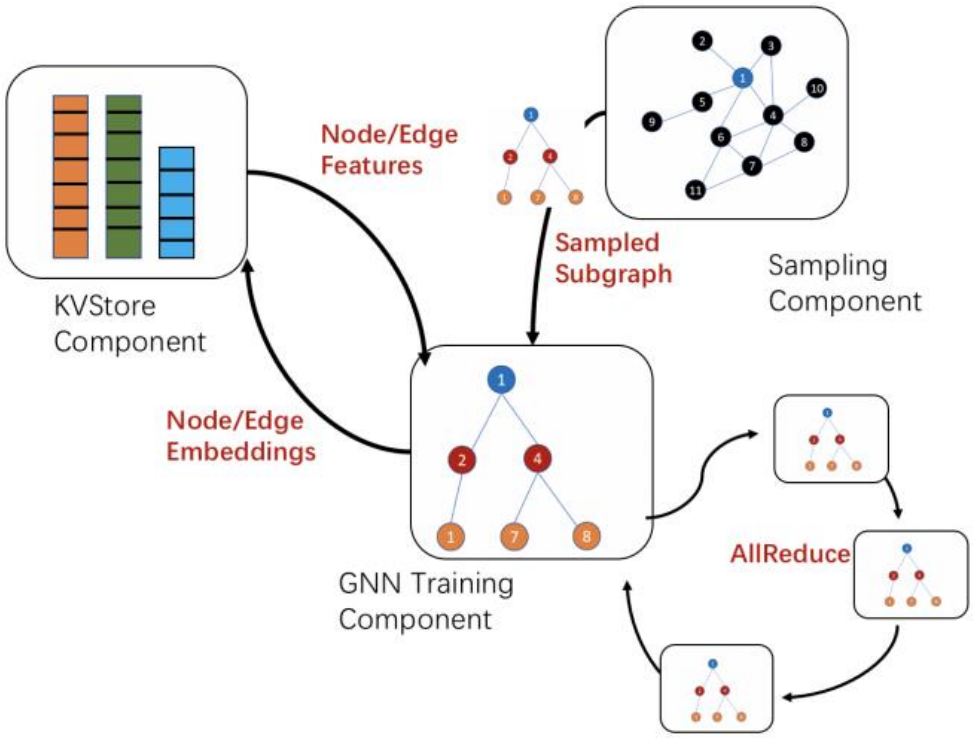
Egy jelenleg kísérleti fázisban lévő stratégián is dolgoznak, ami egygépes, több GPU-s környezetben működik szinkron módon. Lényeges eltérése a hasonló rendszerektől, hogy ebben az esetben a változók nincsenek tükrözve az összes GPU-ra, hanem a CPU-n kapnak helyet, és a csak a műveletvégzés hárul a GPU-ra. Abban az esetben, ha csak egy grafikus kártya van a gépben, akkor a modell változói is arra kerülnek. [33]

5.2. DistDGL

A Deep Graph Library (DGL) egy nyílt forráskódú Python csomag, mely különböző gráf neurális hálózat modellek készítését és tanítását teszi lehetővé. Ez olyan, már meglévő mély tanulást támogató keretrendszerekre épül, mint például a Tensorflow vagy PyTorch. A DGL-t egy mély tanulásra épülő csoport hozta létre, amely a Distributed Deep Machine Learning Community névre hallgat. Rendkívül jól átlátható, letisztult, tömör API-t használ. Eddig több mint kétszázan járultak hozzá a fejlesztéséhez valamilyen formában. A DGL a Gilmer [34] által leírt neurális üzenetküldés paradigmájára épül. Ez egy sokoldalú keretrendszert nyújt, mely lefedi a különböző gráf rétegek típusait. Számos egyéb területre specializálódott alkalmazás épül erre a csomagra, sőt már az sem ritka, hogy egyéb kiegészítő csomagokat építenek a DGL-re. Ilyen például a DGL-LifeSci ami kimondottan olyan gráf alapú mély tanulási célokra született, melyek legfőképp a bioinformatika területén érhetnek el hatalmas előrelépést, illetve a DGL-KE amely úgynevezett tudás gráfok (knowledge graph) beágyazott adatainak feldolgozására szolgál. A DGL bevezetett egy magasabb szintű absztrakciót, ami lehetővé teszi az automatikus kötegelést. Ennek a mechanizmusnak köszönhetően sokkal jobban tudja a DGL optimalizálni a sebességet. [35][36][37]

A DistDGL egy olyan keretrendszer, mely részgráfok (mini-batch) előállításával képes tanítani gráf neurális hálózatokat elosztott környezetben. Ez a rendszer a szinkronizált

sztochasztikus gradiens csökkentés (stochastic gradient descent, SDG) tanítási elvet követi. A DistDGL architektúráját az 5.2-es ábrán is látható módon négy logikai komponensre lehet bontani, melyek közül a mintavételező komponens (Sampling Component) a bemeneti gráfból az egyes részgráfok összeállításáért felelős, amit RPC kommunikációval valósít meg. A KVStore komponens elosztott módon képes tárolni az összes csúcspont, illetve él adatait, továbbá két interfészt is biztosít ezen adatok fel- és letöltéséhez. A tanító folyamatok arra hivatottak, hogy minden iterációban a kapott részgráfhoz tartozó csúcspontokhoz illetve élekhez tartozó adatokat lekérjék a KVStore-ból, majd a számításaik eredményét közöljék egy kommunikációs rendszerrel (dense model update) ami szinkronban tartja a frissítéseket. A DistDGL kommunikációért felelős komponense felhasználja az alatta lévő keretrendszerek által nyújtott stratégiákat, hogy szinkronizált SDG-t hajtson végre. Így például PyTorch felett egy all-reduce eljárást használ, míg TensorFlow esetében egy paraméter szerver megoldást használ. [38]



5.2. ábra - DistDGL komponensek [38]

Amikor ezeket a logikai komponenseket tényleges gépeken alkalmazzuk, akkor a legfontosabb cél a hálózati forgalom csökkentése. Ezért a DistDGL egy úgynevezett „owner-compute” [38] szabályt alkalmaz. Ennek az eljárásnak a lényege, hogy minden gép azokon az adatokon dolgozik, melyekhez lokálisan fér hozzá. Ennek a tervezésnek, köszönhetően a hálózati adatforgalom nagy mértékben csökkenthető. [38]

5.3. PyTorch distributed

A PyTorch szintén egy nyílt forráskódú, a Facebook AI kutató csoportja által fejlesztett mély tanulást támogató keretrendszer, mely a Torch nevű eszközre épülve jött létre. A PyTorch támogatja a Python, illetve C++ nyelvet is, azonban a Python API jóval kiforrottabb. Nagyon elterjedt keretrendszernek számít manapság a kutatók körében, ennek ellenére nem sokan alkalmazzák éles rendszerekben, ezért az ilyen környezetekben még mindig a TensorFlow uralja a piacot. Egyik legfőbb adottsága a PyTorch-nak a tensorok felhasználásával végzett számítások. Ezek a tensorok tulajdonképpen olyan tárolók, melyek képesek több dimenzióban is tárolni az adatot. Ezek az adatok lehetnek akár egyszerű számok, vektorok, mátrixok vagy olyan több dimenziós listák, mint amiket már a NumPy-nál is megszokhattunk. A különböző tensorokkal végzett számítások egyaránt működnek CPU-n és GPU-n is. Éppen ezért a PyTorch a CPU mellett még a GPU-kat is támogatja, mint végrehajtó egység. Azon felül, hogy e két feldolgozó egységet tudjuk hasznosítani a PyTorch segítségével, még arra is módunk van, hogy az előző fejezetben említett Google által fejlesztett TPU-kat is használjuk. Erre egy olyan kiegészítő csomaggal van lehetőségünk, mely lehetővé teszi, hogy XLA eszközöket, mint amilyenek a TPU-k, is megadhassunk típusként egy modellnek. A PyTorch úgynevezett dinamikus számítási gráfokat alkalmaz egy kód futtatása során. Ezt define-by-run megközelítésnek is nevezik. Ez tulajdonképpen annyit jelent, hogy az egyes számítások minden kódsor után történnek. Ennek köszönhetően sokkal nagyobb flexibilitást kínál, ha egy komplexebb architektúrát kell felépíteni. A PyTorch támogat különböző Feed Forward típusú hálózatokat is. Ezek tulajdonképpen a már korábban említett RNN hálózatok ellenkezője, itt ugyanis a csomópontok közötti kapcsolatok nem alkotnak egy ciklust. Továbbá lehetőség van egyéb neurális hálózat modellek implementálására is a PyTorch platformon, egy kifejezetten erre fejlesztett csomaggal. [39][40][41][42]

A PyTorch Geometric (PyG), egy szintén nyílt forráskódú eszköz, mely segítségével gráf neurális hálózatokat tudunk írni és tanítani. Ez kimondottan a már korábban bemutatott PyTorch keretrendszerre épül, illetve bővíti ki azt úgy, hogy lehetőségünk legyen mély tanulási feladatokat is elvégezni nemeuklideszi adatokon. Tensor centrikus API-t használ, annak érdekében, hogy minél közelebb maradhasson a felépítése a PyTorch-hoz. Továbbá olyan, úgynevezett mini-batch kötegelést implementál, mely lehetővé teszi, hogy több kisebb, vagy akár egyetlen nagy gráfon lehessen dolgozni. Ezen felül még olyan rendszereket is támogat melyekben több GPU található, illetve a Quiver csomag segítségével elosztott módon is képes a gráf neurális hálózatok tanítására, sőt olyan transzformációkat elősegítő csomaggal is rendelkezik, melyek képesek hagyományos, általános gráfokkal is működni, de alkalmasak komplexebb, 3D hálókon vagy pontfelhőkön végzett számításokra is. [43][44]

A PyTorch-nak a distributed csomagja lehetőséget biztosít arra, hogy egy adott modell tanításához szükséges számításokat elosztott környezetben is lehessen végezni. Ez magában foglalja a különböző üzenettovábbító sémákat, melyek segítségével az egyes

folyamatok adatot tudnak közölni bármely másik folyamattal. A distributed csomag két beépített kommunikációs rendszert tartalmaz, az NCCL-t és a Gloo-t. A legfőbb eltérés a kettő között, hogy míg a Gloo csak CPU-kat támogat, addig az NCCL a CPU-kon felül a GPU-kat is. E csomag által nyújtott komponensek 3 fő csoportba rendezhetők. [45][46]

Egyik ilyen az elosztott adat-párhuzamos tanítás (Distributed Data-Parallel Training, DDP), ami egy széles körben alkalmazott paradigma. A DDP esetében a modell másolata rákerül minden végrehajtó egységre, majd minden egyes gép a bemeneti adathalmaz egy részével dolgozik. Ezen kívül a gépek közötti kommunikációról is gondoskodik, valamilyen all-reduce megvalósítással, így szinkron módon végzi az elosztott tanítást. Képes több gépes környezetben vagy egyetlen GPU-val is működni, de lehetőséget biztosít a tanításra olyan helyzetben is amikor, egy gépben több grafikus kártya található. Továbbá kínál egy olyan megoldást, ami egy bizonyos szintű hibatűrést tesz lehetővé. A DDP használatakor gyakori eset lehet, hogy out-of-memory hibába ütközik, amit nem lehet egy egyszerű try-except blokkba tenni. Ez azért van, mert a DDP minden végrehajtójának szigorúan szinkronban kell lennie, illetve minden all-reduce kommunikációnak egyeznie kell az egyes gépeken. Ha azonban az elosztott környezet egyik gépe valamilyen kivételt dob, akkor nagy valószínűséggel deszinkronizációt, okozna. Azonban az elastic csomag lehetőséget biztosít, hogy a különböző erőforrások dinamikusan lépjenek ki illetve be az elosztott környezetbe. [45][47]

Egy másik komponense a distributed csomagnak az RPC alapú elosztott tanítás (RPC-Based Distributed Training, RPC), amely olyan általános tanítási struktúrákat is támogat, melyek nem illeszkednek a DDP megközelítésbe, mint például a már korábban említett paraméter szerver architektúra. [45][48]

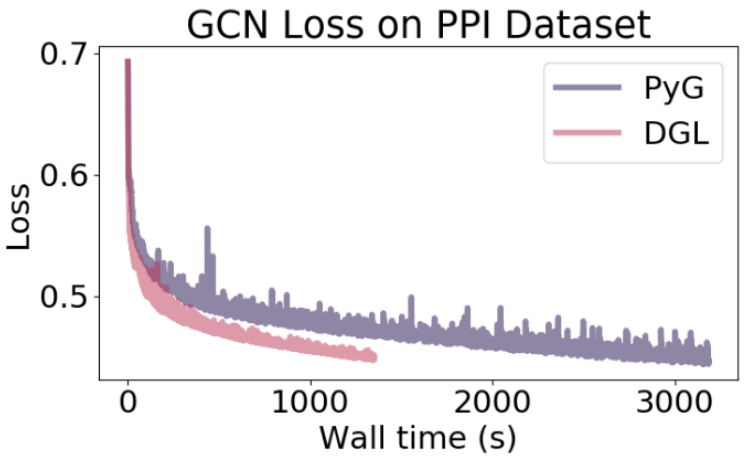
Végül a harmadik összetevője a Pytorch distributed csomagnak a kollektív kommunikációt (Collective Communication, c10d) támogató csomag, amely tensorok továbbítását teszi lehetővé az egyes folyamatok között. Támogatja a kollektív kommunikációra épülő API-kat, illetve a peer-to-peer kommunikációra épülőket is. A már korábban ismertetett DDP és RPC is a c10d-re épül. Általában a fejlesztők nem használják ezt a nyers kommunikációs csomagot, hiszen a DDP vagy RPC segítségével a legtöbb tanítási eljárás kivitelezhető. Azonban vannak olyan esetek, ahol ez az API kimondottan hasznos. Ilyen eset lehet például, amikor a modell paramétereinek az átlagértékét szeretnénk kiszámolni ahelyett, hogy a DDP beépített mechanizmusára hagyatkoznánk. Ezzel sokkal jobban befolyásolhatóvá válik, hogy hogyan működjön a kommunikáció. Így azonban elveszítjük azt a teljesítmény optimalizációt, amit például a DDP nyújt. [45][49]

5.4. Elosztott mély tanulást támogató keretrendszerek összehasonlítása

Az egyik legfontosabb szempont, hogy hogyan lehet hozzáférni az egyes keretrendszerekhez. Ezalatt azt értem, hogy nyílt forráskódú-e az adott rendszer vagy sem. Ez azért lényeges számomra, mert az az elképzelésem, hogy minél szélesebb körben alkalmazható legyen a referencia architektúra. E szempontból mind a három keretrendszer nyílt forráskóddal rendelkezik. A támogatott programozási nyelveket tekintve elmondható, hogy a TensorFlow jóval sokoldalúbb, ugyanis azon felül, hogy a Pythont szintúgy támogatja, mint a PyTorch vagy a DistDGL, még C++ nyelvhez is tartalmaz API-t. A különböző támogatott számítási erőforrásokat figyelembe véve egyformán alkalmazhatók mind grafikus kártyákon, mind CPU-kon. Ezeken kívül a TensorFlow, illetve a PyTorch is támogatja a már korábban ismertetett TPU-kat is melyekkel, jóval optimálisabb tanítást lehet elérni, ha a jövőben ezekre az eszközökre is ki szeretném terjeszteni a keretrendszert. Hagyományos konvolúciós hálózatokat mindegyik rendszer támogat, azonban a feladat megvalósításához olyan keretrendszerre van szükség, ami gráf neurális hálózatokkal is működik. Ennek a feltételnek a korábban bemutatott rendszerek mindegyike eleget tud tenni, újabb csomagok segítségével. A PyTorch platformon a PyG vagy a DistDGL, míg a TensorFlow esetében például a DistDGL vagy a már korábban említett Keras nyújt támogatást a GNN-ek számára. A támogatott elosztott stratégiák szempontjából is mindegyik keretrendszer egyformán alkalmazható akár több grafikus kártyán vagy több gépből álló infrastruktúrában is, így nincs közöttük lényeges eltérés.

		PyTorch	**Tensorflow**	**DistDGL**
Hozzáférhetőség		Nyílt forráskódú	Nyílt forráskódú	Nyílt forráskódú
Támogatott programozási nyelv		Python	Python, C++	Python
Támogatott hardverek		GPU, CPU, TPU	GPU, CPU, TPU	GPU, CPU
Támogatott gráf neurális hálózat típusok		GCNConv, GraphConv, GatedGraphConv, ResGatedGraphConv, SAGEConv	GCNConv, GraphSageConv, GATConv, GINConv, GatedGraphConv	GraphConv, SAGEConv, GatedGraphConv, GATConv, TAGConv, RelGraphConv, EGATConv
Támogatott elosztott stratégiák	**Több GPU egy gépen**	X	X	keretrendszertől függ
	Több gép, melyekben van GPU is	X	X	keretrendszertől függ
	Paraméter szerver	X	X	keretrendszertől függ

5.1. táblázat - Elosztott keretrendszerek összehasonlítása

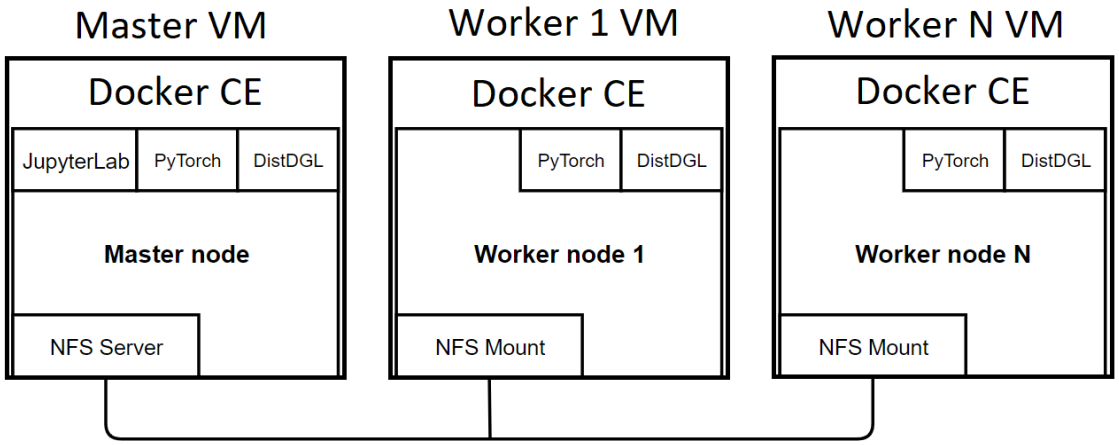


5.3. ábra - Modellek tanításához felhasznált idő, különböző keretrendszerek alatt [50]

Az 5.1-es összefoglaló táblázatból is jól látszik, hogy az egyes keretrendszerek legfőbb tulajdonságai nagyrészt megegyeznek. Azonban a PyTorch a TensorFlow-val ellentétben dinamikus számítási gráfokkal dolgozik. Ennek köszönhetően sokkal flexibilisebb környezetet biztosít egy komplexebb architektúra kiépítése során és a hibakeresés is jóval könnyebbé válik. Ezért a TensorFlow keretrendszer használatát elvetem. A DistDGL sokkal gyorsabban képes dolgozni a PyTorch alatt, mint a PyG, abból az okból kifolyólag, hogy nagyon jó memória optimalizálási mechanizmusokat valósít meg. Ez látható az 5.3-as ábrán is, ahol két egyforma modell tanítását végezték a két rendszer segítségével, és a PyG-nek majdnem háromszor annyi időbe tellett a folyamat, annak ellenére, hogy a pontossága alig növekedett. Ezen okok miatt a PyTorch keretrendszert választom a DistDGL rendszer alá.

6. TERVEZÉS

Az infrastruktúrát egy felhő platformon fogom kialakítani, amiben az egyes erőforrások a felhasználó által definiált mennyiségben virtuális gép alapon, Docker konténerekben futnak majd, ezzel biztosítva a környezet hordozhatóságát. A rendszer egy központi egységből (Master node) és több dolgozó egységből (Worker node) áll majd. Mindegyik eszközön rendelkezésre kell állnia a PyTorch és a DistDGL csomagoknak. Ezen felül kell lennie egy JupyterLab szolgáltatásnak a központi gépen, ami egy grafikus interfészként funkcionál a felhasználó és az elosztott környezet között. A felhőben lévő összes gépnek hozzáférést kell biztosítani az aktuális, legfrisseb tanító szkripthez, illetve a tanításhoz szükséges bemeneti adathalmazhoz. Erre egy NFS megosztást fogok alkalmazni. Az NFS szerver a master virtuális gépen lévő konténerben fog futni. A többi virtuális gépen, nem konténerizált formában fogom konfigurálni az NFS klienseket. Ebből kifolyólag a dolgozó gépeken futó konténerek, úgy lesznek képesek elérni a megosztott fájlrendszert, hogy a virtuális gépre felcsatolt megosztást, továbbítom a konténer felé az indítási folyamat során. Az alábbi 6.1-es ábrán a tervezett referencia architektúra infrastruktúrája látható, a szükséges szolgáltatásokkal együtt.



6.1. ábra - Referencia architektúra terv

Az infrastruktúra létrehozását a Terraform nevű felhő orkesztrátorral valósítom meg. Ez a rendszer képes arra, hogy különböző leíró fájlok segítségével automatizált módon hozza létre az előre definiált felhő struktúrát. Itt a felhasználónak lehetősége lesz különböző változók definiálása révén az infrastruktúra konfigurációját befolyásolni. Olyan paramétereket tervezek bevezetni az egyes leíró fájlok elkészítése során, melyek segítségével a felhasználó megadhatja például a virtuális gépek számát, erőforrásainak mennyiségét, háttértáainak méretét, illetve az NFS által megosztott tárhely méretét is. Továbbá tervezem olyan paraméterek használatát is, melyek az infrastruktúra egészére vannak hatással. Itt olyan tényezőkre gondolok, mint például a JupyterLab felületére

történő belépéshez szükséges jelszó definiálása vagy az, hogy a felhasználónak van-e igénye grafikus kártyák használatára vagy sem.

Az infrastruktúra létrehozásán kívül szükség van továbbá arra, hogy szintén egy automatizált megoldással lehessen konfigurálni az egyes gépeket. Erre a feladatra az Ansible konfigurációs menedzsment eszközt fogom alkalmazni. Ennek ugyanis a master-slave architektúrára épülő rendszerekkel ellentétben, csak egy SSH kapcsolatra van szüksége, hogy alkalmazza az előre definiált beállításokat a konfigurálandó gépeken, így ez egy jóval kezelhetőbb megoldást nyújt. Tervezem továbbá az úgynevezett SSH proxy használatát is a konfiguráció során. Ez azért szükséges, mert csak a master gép IP címe lesz publikusan elérhető, így ahhoz, hogy az összes többi gép is megkapja a helyes konfigurációs beállításokat, a master gép egyfajta proxyként továbbítja az Ansible-től érkező adatcsomagokat.

Két különböző konténert tervezek készíteni, ezek leginkábban abban fognak különbözni, hogy míg az egyik elő van készítve grafikus kártyák használatára is, addig a másik csak és kizárólag a CPU-n képes tanítási folyamatokat végezni. Ezen konténerekben lévő alkalmazásokat úgynevezett belépési pont (entrypoint) szkriptekkel fogom inicializálni. Ezekben fogom definiálni, az SSH szerverre vonatkozó beállításokat, illetve a felhasználó által megadott környezeti változók alapján indítom a JupyterLab szolgáltatást, ha szükséges.

7. IMPLEMETÁCIÓ

Az implementáció során Docker technológiát használtam, így először a konténereket alakítottam ki. Ezt követően létrehoztam a Terraformhoz szükséges fájlokat, majd végül megalkottam a konfiguráció menedzsmenthez szükséges leírókat és integráltam az orkesztrációs folyamatba.

7.1. Konténerek felépítése

Kétféle konténert hoztam létre, melyek közül a felhasználó igénytől függően kiválaszthatja, hogy melyiket szeretné használni. Az egyik csak CPU-n támogatja az elosztott tanítást, míg a másik NVIDIA grafikus kártyákkal is képes dolgozni, így a két konténer közti különbség csak a kiinduló képfájlból adódik.

A mellékletben található *docker/gpu/entrypoint.sh* fájl szolgál a konténer belépési pontjaként. Ez először készít egy SSH kulcsot, majd hozzáadja azt egy, a megosztott mappában található fájlhoz. Ennek az elérési útját egy környezeti változó segítségével lehet megadni. Ha üres ez a változó, akkor egy hibaüzenet tájékoztatja a felhasználót.

```
if [[ ! -z "${SSH_PORT}" ]]; then
    sed -i 's:#AuthorizedKeysFile:AuthorizedKeysFile:' /etc/ssh/sshd_config
    sed -i 's:.ssh/authorized_keys .ssh/authorized_keys2:.ssh/authorized_keys .ssh/authorized_keys2
/shared/nfsssh/authorized_keys:' /etc/ssh/sshd_config
    sed -i 's:#Port 22:Port '$SSH_PORT':' /etc/ssh/sshd_config
    service ssh restart
else
    echo "ERROR: SSH_PORT environment variable not set."
    exit 1
fi
```

7.1. ábra – Konténer belépési pont, SSH beállítások

Ahhoz, hogy a konténerek közötti SSH kapcsolat megfelelően működjön, módosítani kell az SSH szerver beállításait. Ez a folyamat látható a 7.1-es ábrán. A már korábban létrehozott kulcsok elérési útját hozzáadtam a megbízható kulcsok listájához, így nincs szükség jelszóra, két gép között. Ezen felül továbbá a szkript módosítja az alapértelmezett SSH portot is, amit szintén egy környezeti változóval adhat meg a felhasználó.

Az alábbi 7.2-es ábrán a JupyterLab konfigurációja látható. A felhasználó ismét egy környezeti változó segítségével adhatja meg, hogy az adott konténerben elinduljon-e egy Jupyter szerver. Itt készül egy hash a szöveges formában lévő jelszóból, amit a felhasználó adott meg. Erre azért van szükség, mert a Jupyter hash-elt formában menti a belépéshez a jelszót. Ha a felhasználó úgy dönt, hogy ne fusson a Jupyter, akkor a

konténerben lévő szolgáltatások csak várakozik, amíg el nem kezdődik a tanítási folyamat. Ezek után a konténerben elkezd futni az a parancs, mely a mellékletben található *docker/gpu/Dockerfile.gpu* fájl CMD blokkjában van definiálva. Ez elindítja magát a Jupyter szervert a megadott jelszóval, illetve beállítja a megosztott mappa elérési útját is.

```
if [[ "${JUPYTER_LAB}" = "true" || "${JUPYTER_LAB}" = "True" || "${JUPYTER_LAB}" = "TRUE" ]]; then
    CONFIG_DIR=/root/.jupyter
  if [[ ! -f "$CONFIG_DIR/jupyter_server_config.py" ]]; then
    mkdir -p $CONFIG_DIR
    export JUPYTER_PASSWORD_HASH=$(python3 -c "from jupyter_server.auth import passwd;
print(passwd('$JUPYTER_PASSWORD', 'sha1'))")
    echo "c.ServerApp.password = '$JUPYTER_PASSWORD_HASH'" >> $CONFIG_DIR/jupyter_server_config.py
    echo "c.ServerApp.terminado_settings = {'shell_command': ['/bin/bash']}" >>
$CONFIG_DIR/jupyter_server_config.py
  fi
  set -- jupyter lab "$@"
  exec "$@"
else
  sleep infinity
fi
```

7.2. ábra – Konténer belépési pont, JupyterLab indítás

7.2. Infrastruktúra létrehozás

Az infrastruktúra létrehozásához szükséges Terraform leírót, jól elkülöníthető szempontok alapján több fájlba szerveztem ki. Ezek a fájlok a mellékletben, a *terraform* mappában találhatók. A fájlok neveit az infrastruktúra létrehozása során betöltött szerepük szerint neveztem el. A *main.tf* fájlban az inicializációhoz szükséges adatokat tárolom, a *fip.tf* felel a publikus IP címek gépekhez való hozzárendeléséért, a *configs.tf* állományban az Ansible leírókat integráltam, illetve a *storage.tf* fájlban a megosztott tárolót hozom létre és csatolom fel a master gépre.

Ezeken felül létrehoztam a *userconfig.auto.tfvars* nevű állományt melyben kizárólag olyan adatok szerepelnek, amiket a felhasználó szabadon módosíthat. Itt olyan paramétereket adhat meg, mint például az autentikációhoz szükséges kulcsok, illetve a felhő elérési útja. Továbbá itt kell definiálni az erőforrások tulajdonságait, a megosztott tárhely méretét, illetve a JupyterLab-hoz tartozó jelszót és azt is, hogy legyen-e a gépekben dedikált GPU. Itt a tároló esetében a felhasználó megadhat egy létező kötetet is ID alapján, hogy azt csatolja fel a rendszer. Ezek után az erőforrásokat létrehozó

instances.tf nevű leíró paraméterként kapja meg a felhasználó által meghatározott tulajdonságokat.

A megfelelő működéshez elengedhetetlen a biztonsági csoportok használata. Ezeket a *security_groups.tf* nevű fájlban úgy hoztam létre, hogy ügyelnem kellett arra, hogy csak bizonyos szolgáltatások portjai legyenek elérhetőek egy külső IP címről. Ugyanakkor az is szempont, hogy a gépek tudjanak egymással kommunikálni a privát hálózaton. Ennek

```
resource "openstack_networking_secgroup_rule_v2" "jupyter-rule" {
  direction        = "ingress"
  ethertype        = "IPv4"
  protocol         = "tcp"
  port_range_min    = 8888
  port_range_max    = 8888
  remote_ip_prefix  = "0.0.0.0/0"
  security_group_id = openstack_networking_secgroup_v2.BognarT-master.id}

resource "openstack_networking_secgroup_rule_v2" "tcp-rule" {
  direction        = "ingress"
  ethertype        = "IPv4"
  protocol         = "tcp"
  port_range_min    = 0
  port_range_max    = 0
  remote_ip_prefix  = "192.168.0.0/24"
  security_group_id = openstack_networking_secgroup_v2.BognarT-all-node.id}
```

7.3. ábra – Tűzfal szabályok (részlet)

érdekében létrehoztam egy olyan biztonsági csoportot, mely beengedi az SSH kapcsolatot a hálózatba, illetve nyit egy portot a JupyterLab számára is. Ezeket a szabályokat csak a master node kapja meg. Egy másik biztonsági csoport felel azért, hogy engedje a kommunikációt a privát hálózaton. Ezt úgy értem el, hogy csak az adott hálózati címről érkező TCP, UDP csomagokat engedjék a szabályok. Tesztelés miatt az ICMP protokoll használata is engedélyezett a hálózaton.

A fenti 7.3-as ábrán látható a korábban említett tűzfal szabályok közül kettő, melyek sorrendben, a JupyterLab kívülről történő elérését, illetve a privát hálózat TCP forgalmát engedélyezik.

7.2.1. Létrehozott erőforrások IP címeinek összesítése

Az elosztott tanítás egyik alapvető feltétele, hogy a tanítási folyamatban résztvevő gépek tudjanak kommunikálni egymással. Ez a kapcsolat azért szükséges, mert az elosztott tanítási folyamatot elindító szkript így tudja az egyes gépeken futtatni magát a tanító szkriptet. Éppen ezért, hogy mindegyik gép kapcsolatot tudjon létrehozni egy másikkal, valamilyen módon meg kell szerezni és elérhetővé kell tenni a cél gépek IP címét. Ezt lokálisan futtatható szkriptek segítségével értem el. Először létrehoztam egy olyan resource-ot, mely készít egy fájlt amiben majd a privát IP címeket el tudom tárolni.

Miután létrejött a fájl a Terraformból kinyert master node címét bele is írja a szkript. Ezt követően egy másik szkript végigmegy a többi gépen és minden iterációban kiírja az adott gép privát IP címét ebbe az állományba. Végül egy harmadik resource egy Ansible leíró segítségével felmásolja a már elkészült IP listát tartalmazó fájl a megosztott tárhelyre, így minden gép rendelkezik a hálózaton lévő többi gép IP címével. Ez a folyamat úgy van beállítva, hogy ha valamelyik létfontosságú paraméter megváltozik az infrastruktúrában, például a master node, vagy a worker gépek száma, akkor újra lefutnak a szkriptek. Így nem fordulhat elő olyan, hogy bármikor is helytelen adatok szerepeljenek az IP-ket tartalmazó fájlban. A korábban említett eljárások az alábbi 7.4-es ábrán láthatók.

```
resource "null_resource" "create_inventory" {
  triggers = {
    master_id = openstack_compute_instance_v2.master.id
    worker_ids = join(",", openstack_compute_instance_v2.worker.*.id)
    shared_volume_id = null_resource.master_train_volume_attach.id}
  provisioner "local-exec" {
    working_dir = "../ansible/misc/"
    command = "rm -f inventory && echo $MASTERIP >> inventory"
    environment = {
      MASTERIP = openstack_compute_instance_v2.master.network.0.fixed_ip_v4}}
  depends_on = [openstack_compute_instance_v2.master, openstack_compute_instance_v2.worker,]}
resource "null_resource" "fill_inventory" {
  count = var.worker_node.count
  triggers = {
    master_id = openstack_compute_instance_v2.master.id
    worker_ids = join(",", openstack_compute_instance_v2.worker.*.id)
    shared_volume_id = null_resource.master_train_volume_attach.id}
  provisioner "local-exec" {
    working_dir = "../ansible/misc/"
    command = "echo $CURRENTIP >> inventory"
    environment = {
      CURRENTIP = openstack_compute_instance_v2.worker[count.index].network.0.fixed_ip_v4
    }}
  depends_on = [null_resource.create_inventory,]}
resource "null_resource" "copy_inventory" {
  triggers = {
    fill_inventory_ids = join(",", null_resource.fill_inventory.*.id)}
  provisioner "local-exec" {
    working_dir = "./"
    command = <<EOF
            ANSIBLE_HOST_KEY_CHECKING=False ANSIBLE_SSH_RETRIES=10 \
            ansible-playbook --private-key=~/.ssh/BognarT-key.pem -u ubuntu -i
'${openstack_compute_floatingip_associate_v2.fip_1.floating_ip},'
../ansible/misc/copy_inventory_to_cluster.yaml
    EOF}
  depends_on = [null_resource.create_inventory,]}
```

7.4. ábra – Létrehozott erőforrások IP címeinek kigyűjtése

7.3. Infrastruktúra konfigurálás

Az Ansible fájlok a mellékletben az *ansible* mappában találhatók. Itt további két mappa kapott helyet. A *misc* nevű mappa tartalmazza a *copy_inventory_to_cluster.yaml* leírót, illetve a *mount_train_volume.yaml* leírót is. Ezeken felül ide generálódik a Terraform által az előbb említett inventory fájl is. Az Ansible lehetővé teszi úgynevezett szerepek (role) használatát a konfigurációs fájlok létrehozásakor, így a master és worker gépekre vonatkozó beállításokat ennek a gyakorlatnak megfelelően hoztam létre. Ez nem csak a konfigurációs fájlokat egyszerűsíti le, hanem ezáltal az esetlegesen felmerülő hibák felderítését és javítását könnyíti meg, annak köszönhetően, hogy az egymástól független eljárások külön fájlokban vannak elhelyezve. Ezek a role-ok a korábban említett *ansible* mappában belül a *roles* mappában kaptak helyet. Itt az Ansible által javasolt elnevezési konvenciókat alkalmaztam. Ez azt jelenti, hogy minden egyes független szerep külön mappában kapott helyet úgy, hogy a tényleges leíró ezen mappán belül egy *tasks* nevű mappában van elhelyezve. Például a master gépre vonatkozó szerep a *roles/master/tasks/main.yaml* elérési úton van definiálva. A főkönyvtárban található még kettő, eddig nem említett fájl is. A *master_config.yaml*, illetve a *worker_config.yaml* tartalmazza az adott géptípushoz tartozó szerep hozzárendeléseket.

Mindkét géptípushoz, a masterhez, illetve a workerhez is hozzá van rendelve egy közös szerep. Ez megbizonyosodik arról, hogy régebbi Docker ne legyen telepítve a gépeken, majd telepíti a legfrissebb verziót, a szükséges függőségekkel együtt. A master, illetve worker gépekre vonatkozó egyedi szerepeinek konfigurációs fájljai annyiban hasonlítanak, hogy nagyrészt konténerizált alkalmazásokat indítanak el. Azonban ahhoz, hogy fájlokat tudjak megosztani a hálózaton keresztül, a master gépen a tervezésnek megfelelően, egy NFS szervert hozok létre, míg a worker gépek konfigurációs fájljában a korábban elindított szerverre csatlakozva egy mappán keresztül felcsatolom a megosztott kötetet. Ezek után már csak magát, a korábban elkészített, elosztott tanításhoz szükséges alkalmazásokat tartalmazó konténert kell elindítani a felhasználó által definiált beállításokkal. A feladat során létrehoztam egyéb leírókat is, ezek közül az egyik, a már korábban említett *copy_inventory_to_cluster.yaml fájlban található.* Ez arra szolgál, hogy az összegyűjtött IP címeket tartalmazó fájlt felmásolja a megosztott tárhelyre, amit már említettem a korábbi alfejezetben. A másik, *mount_train_volume.yaml* fájlban lévő leíróra, azért van szükség, mert a felhő által kínált tárhely még nyers formátumban van, így ez a leíró kialakít rajta egy ext4-es fájlrendszert, továbbá létrehozza a rendszer által elvárt mappaszerkezetet is. Ezekbe a mappákba kerülnek a tanításhoz tartozó adatok, mint például maga a szkript és az ehhez szükséges adathalmazok, illetve az SSH kulcsokat tartalmazó fájl is.

7.3.1. Konténerek konfigurálása Ansible-ben

Az Ansible-höz egyéb, a közösség által készített és fenntartott kiegészítő csomagok érhetők el. Ezek között van olyan, ami kimondottan Docker konténereket tud indítani és

menedzselni. A feladat során ezt a kiegészítőt használtam én is a rendszer megfelelő felépítéséhez. Az alábbi 7.5-ös ábrán az elosztott tanításhoz szükséges alkalmazásokat

```
- name: Start custom master container with GPU
  community.docker.docker_container:
    name: py_dgl_jup_gpu
    image: t0mas999/pytorch-dgl-base:gpu-latest
    network_mode: host
    pull: yes
    recreate: true
    restart: true
    restart_policy: always
    privileged: true
    ports:
     - "8888:8888"
    device_requests:
     - driver: nvidia
       count: -1
       capabilities:
        - - compute
          - utility
    env:
     JUPYTER_LAB: "true"
     JUPYTER_PASSWORD: "{{ jupyterPassword }}"
     SSH_PORT: "12345"
     NFS_SSH_PATH: "/shared/nfsssh"
    mounts:
     - target: "/shared"
       source: "/mountedvol"
       read_only: no
       type: "bind"
    state: started
  when: gpuEna
```

7.5. ábra – Konténer indítási beállításai

Első lépésként megadtam a minimum elvárt adatokat, mint például a konténer nevét, képfájl elérési helyét, újraindításra vonatkozó beállításokat. Ezek után a hálózati elérést biztosító változót úgy állítottam be, hogy magának a virtuális gépnek az IP címét kapja meg a konténer, illetve a 8888-as port legyen nyitva a JupyterLab miatt. A grafikus kártyára vonatkozó beállításoknál meg kellett adnom, hogy melyik driver-t töltse be, illetve a GPU mely képességeit fogom használni a későbbiekben. A virtuális gépben található grafikus kártyák számát is itt kell definiálni, jelen esetben az összes elérhető GPU-ra szükségem van. Ezt Ansible-ben úgy lehet beállítani, hogy a darabszámra utaló változó helyére -1-et írok. A soron következő blokkban a konténerben futó alkalmazások környezeti változóit adom meg. Ezek azért szükségesek, hogy a korábban bemutatott, konténer belépési pontjaként szolgáló szkript el tudja dönteni, hogy a felhasználó szeretne-e JupyterLab-ot indítani és ha igen akkor mi legyen a webes felületre történő

belépéshez szükséges jelszó, illetve, hogy milyen port szolgál a konténerek közötti SSH kommunikációhoz. Ezeken kívül itt adom át azt is, hogy milyen útvonalon éri el az SSH kulcsokat tartalmazó fájlt. Végül az egészet egy olyan változó vizsgálásával zárom, mely segítségével a felhasználó megadhatja, hogy a GPU-t használó képfájlt használja a rendszer, vagy csak a processzorral működő verziót töltse le.

7.3.2. Távoli gépek elérése a konfiguráláshoz

Ahhoz, hogy az Ansible el tudja végezni a szükséges konfigurációs lépéseket, egy SSH kapcsolaton keresztül hozzá kell férnie az aktuálisan konfigurálandó géphez. Ehhez szükséges, hogy a virtuális gép rendelkezzen egy publikus IP címmel, hiszen az Ansible egy másik hálózaton lévő gépen fut. Ez a master gép esetében nem jelent problémát, mert ennek már van publikus IP címe. Azonban a tanítási folyamatban résztvevő többi gépnek csak privát IP címe van. Erre az esetre kínál megoldást egy úgynevezett SSH proxy használata. Ez annyiban tér el a hagyományos megoldástól, hogy itt egy parancssori paraméter segítségével meg kell adni egy olyan gépnek az IP címét is, ami a hálózaton kívülről is elérhető, illetve a belső hálózathoz is van hozzáférése. Így a belső hálózaton lévő virtuális gépet egy ugrással éri el az Ansible. A létrehozott konfigurációs leírókat könnyedén lehet integrálni a Terraform folyamatába. Ehhez a Terraform úgynevezett lokálisan futtatható eljárásait lehet kihasználni. Mivel minden egyes blokknak valamilyen resource-hoz kell kapcsolódnia, ezért egy speciális *null_resource*-ba kell elhelyezni az ehhez hasonló konfigurációkat. Ezután parancssoros módon adtam meg az Ansible számára szükséges információkat. A fent említett folyamatnak a kódrészlete az alábbi 7.6-os ábrán látható.

```
resource "null_resource" "worker_config" {
 triggers = {
  master_id = openstack_compute_instance_v2.master.id
  worker_ids = join(",", openstack_compute_instance_v2.worker.*.id)}
 count = var.worker_node.count
 provisioner "local-exec" {
  working_dir = "./"
  command = <<EOF
        ANSIBLE_HOST_KEY_CHECKING=False ANSIBLE_SSH_RETRIES=10 \
        ansible-playbook --private-key=~/.ssh/BognarT-key.pem -u ubuntu -i
'${openstack_compute_instance_v2.worker[count.index].access_ip_v4},'
../ansible/worker_config.yaml \
        --extra-vars 'NFS_SERVER=${openstack_compute_instance_v2.master.access_ip_v4}
gpuEna=${var.user_spec.gpuEna}' \
        --ssh-common-args '-o ProxyCommand="ssh -o StrictHostKeyChecking=no -o
UserKnownHostsFile=/dev/null -W %h:%p \
        -i ~/.ssh/BognarT-key.pem
ubuntu@${openstack_compute_floatingip_associate_v2.fip_1.floating_ip}"'
  EOF}
 depends_on = [null_resource.master_train_volume_attach, ]}
```

7.6. ábra – SSH proxy használata a worker gépek konfigurálása során

8. TESZTELÉS

A rendszeren elvégzett funkcionális teszteket úgy bontottam részekre, hogy az először a leírók helyes végrehajtását, majd egy tesztkörnyezet felépítésével a tényleges elosztott tanítás működését vizsgáltam.

8.1 Infrastruktúra automatikus kiépítésének tesztelése

A funkcionális tesztek lényege, hogy az infrastruktúra valóban megfelelően kiépült-e, illetve az elvárt beállításoknak megfelelően van-e konfigurálva. Miután lefuttattam a leírókat, az alábbi 8.1-es ábrán látható kimenetet írta ki a Terraform a konzolra. Ebből az látszik, hogy minden egyes részegysége az infrastruktúrának, a leírókban definiált állapot szerint került létrehozásra.

```
Apply complete! Resources: 23 added, 0 changed, 0 destroyed.
tamas@DESKTOP-T7SVSKG:/mnt/d/SharedVM/SZAKDOLGOZAT/GNNDistEnv/terraform$
```

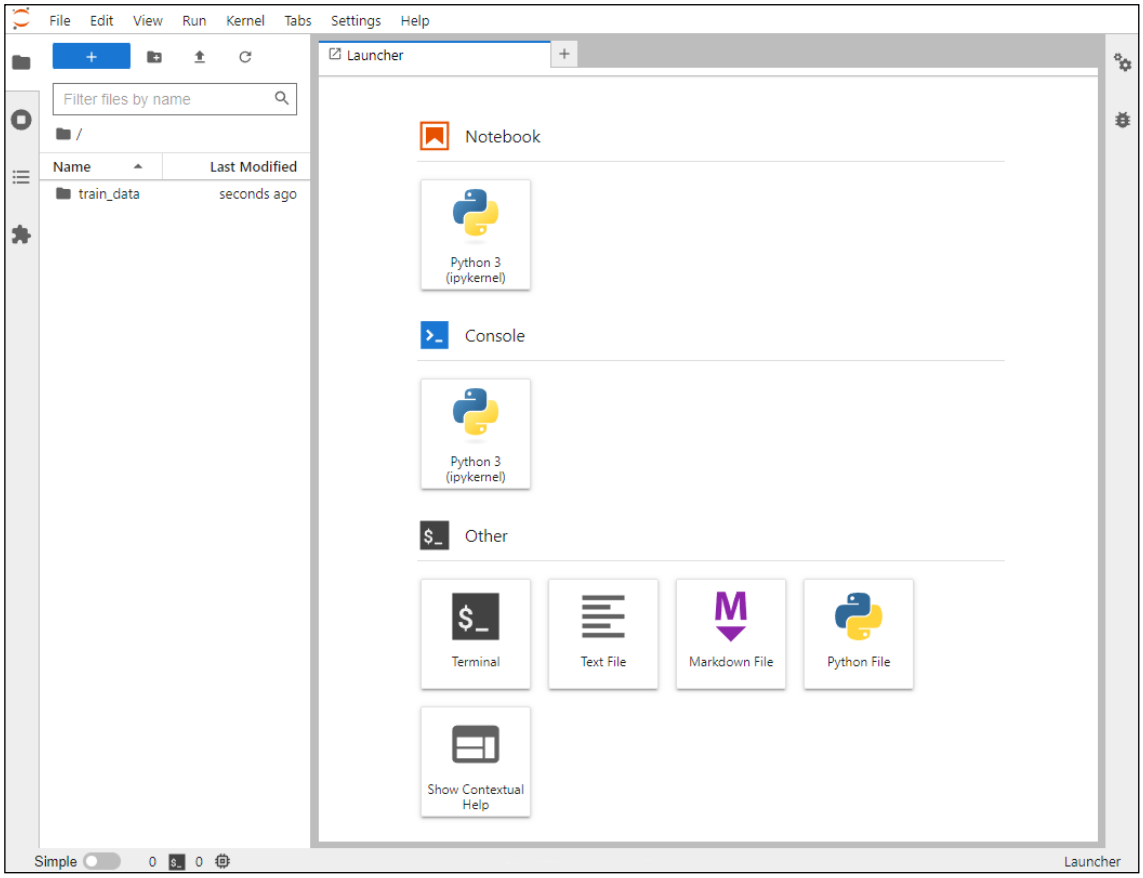
8.1. ábra – Terraform konzolos kimenete

Ez azonban önmagában kevés információ a rendszerről, így a felhő grafikus interfészén is megvizsgáltam, hogy az infrastruktúra kiépítse valóban megfelelően lezajlott. Az alábbi 8.2-es ábrán az látható, hogy 4 virtuális gép jött létre, ezek közül egynek publikus IP címe is van, amit biztonsági okokból eltávolítottam a képről. Látható továbbá, hogy valóban a felhasználó által definiált típusú erőforrásokat és képfájlokat használta fel a rendszer, illetve a korábban manuálisan létrehozott SSH kulcsot is hozzárendelte az egyes gépekhez.

Instance Name	Image Name	IP Address	Flavor	Key Pair	Status	Availability Zone	Task	Power State	Age
BognarT-worker-3	Ubuntu 20.04 LTS - NV	192.168.0.197	g2.large	BognarT-key	Active	nova	None	Running	1 hour, 7 minutes
BognarT-worker-2	Ubuntu 20.04 LTS - NV	192.168.0.91	g2.large	BognarT-key	Active	nova	None	Running	1 hour, 7 minutes
BognarT-worker-1	Ubuntu 20.04 LTS - NV	192.168.0.52	g2.large	BognarT-key	Active	nova	None	Running	1 hour, 7 minutes
BognarT-master	Ubuntu 20.04 LTS - NV	192.168.0.31, [redacted]	g2.large	BognarT-key	Active	nova	None	Running	1 hour, 7 minutes

8.2. ábra – Létrehozott erőforrások listája a felhő grafikus felületén

Miután megbizonyosodtam a rendszer helyes felépítéséről, beléptem a master gépen futó JupyterLab felületére, mely a publikus IP cím 8888-as portján érhető el. Az autentikáció során az alkalmazás az alapértelmezett jelszó helyett a felhasználó által definiált jelszót kérte, a tervezetteknek megfelelően. Az alábbi 8.3-as ábrán a JupyterLab kezdőfelülete látható. A felhasználónak lehetősége van itt megírni a tanító szkriptet, illetve rögtön futtatni is tudja egy konzolos ablakban.



8.3. ábra – JupyterLab grafikus felülete

Végül, de nem utolsó sorban teszteltem a NFS megosztás helyes működését is. Ezt úgy értem el, hogy beléptem az egyik virtuális gépbe és megvizsgáltam az elvárt mappaszerkezetet, illetve ezen mappán belül található fájlokat. Az alábbi 8.4-es ábrán látható, hogy a leírókban definiált struktúrát építette fel a rendszer, továbbá a master gépen létrehozott tanító szkriptek és adathalmazok is elérhetőek a többi gép számára is. Ezeken felül látható még, hogy a DistDGL által megkövetelt, hálózaton lévő gépek privát IP címeit tartalmazó fájlt is megfelelően létrehozta, illetve fel is töltötte a megosztott tárhelyre a Terraform.

```
root@bognart-worker-3:/shared/nfsdata# ls
MPgputest.py  __pycache__  create_partitions.py  dataset  dgltest.py  gputest.py
 inventory  launch.py  launch_cmd.txt  load_graph.py  nvidia_check.py  part_data
 partition_graph.py
root@bognart-worker-3:/shared/nfsdata# cat inventory
192.168.0.31
192.168.0.91
192.168.0.52
192.168.0.197
root@bognart-worker-3:/shared/nfsdata#
```

8.4. ábra – Megosztott fájlok elérése az egyik worker gépen

8.2 Tesztkörnyezet létrehozása

Ahhoz, hogy elosztott gráfokon végezzek tanítást, először egy megfelelő bemeneti adathalmazra volt szükségem. Erre a feladatra az OGBN által nyújtott, nyíltan hozzáférhető Arxiv nevű gráf alapú adathalmazt használtam fel. [52] Azonban mielőtt használni tudtam volna ezeket az adatokat, több részre kellett bontani. Erre azért volt szükség mert az elosztott működés eléréséhez minden, a tanítási folyamatban résztvevő gépnek rendelkeznie kell a teljes adathalmaz egy részével. Erre a feladatra a DGL dokumentációjában található particionálásra használt szkriptet használtam fel. [53]

Ezt követően elkészítettem magát a tanító szkriptet. Ehhez szintén a DGL dokumentációjában található szkripeket vettem alapul [53][54], itt azonban néhány módosítást kellett végrehajtanom. Az egyik ilyen módosítás az alábbi 8.5-ös ábrán látható. Itt ugyanis a backend kiválasztása során figyelembe kell venni, hogy van-e rendelkezésre álló grafikus kártya és ha van, akkor használja azt a szkript. Ezek az úgynevezett backendek a Pytorch belső kommunikációjáért felelősek. A Pytorch ugyan több backendet is támogat, de nem mindegyik alkalmas arra, hogy a GPU-t használja a tanítás során. Éppen ezért, amikor grafikus kártyát szeretnék használni, akkor az NCCL backendet alkalmazza a szkript a Gloo helyett, ami csak a CPU-t támogatja.

```
if th.cuda.is_available():
    device = th.device('cuda')
    th.distributed.init_process_group(backend='nccl')
else:
    device = th.device('cpu')
    th.distributed.init_process_group(backend='gloo')
```

8.5. ábra – Megfelelő backend kiválasztása a tanító szkriptben

A másik módosítás a tényleges tanítás egyes iterációiban figyelhető meg. Itt úgy alakítottam át a szkriptet, hogy képes legyen a GPU-n lévő tensorokkal dolgozni. Ezek a módosítások a mellékelt *gputest.py* fájl iterációs folyamatot végző blokkjában található. Ezen felül hozzáadtam egy teszt blokkot, mely a neurális hálózat által produkált eredményeket teszteli, az adathalmazból erre a célra kiválasztott csomópontok felhasználásával. Az alábbi 8.6-os ábrán a hozzáadott teszt blokk látható.

```
#test
tests = []
t_labels = []
with th.no_grad(), model.join():
   for step, blocks in enumerate(test_dataloader):
      inputs = blocks[0].srcdata["feat"]
      t_labels.append(blocks[-1].dstdata["labels"].cpu().numpy())

      blocks = [block.to(device) for block in blocks]
      tests.append(model(blocks, inputs).argmax(1).cpu().numpy())
   tests = np.concatenate(tests)
   t_labels = np.concatenate(t_labels)
   test_accuracy = sklearn.metrics.accuracy_score(t_labels, tests)
   print('World rank: {}, Epoch: {}, Validation Accuracy: {}, Test Accuracy: {}, Elapsed time: {:.2f} seconds'
        .format(world_rank, epoch, val_accuracy, test_accuracy, time.time()-start_time))
```

8.6. ábra – Teszt blokk módosított változata

Miután elkészültem a tanító szkripttel és az adathalmazt is particionáltam, már csak a folyamat elindítása volt hátra. Ahhoz, hogy ilyen és ehhez hasonló elosztott elven működő tanítást lehessen végezni egy speciális indító szkriptre van szükség. Ezt szintén biztosítja a DGL, így ezt használtam fel. [55] Ez tulajdonképpen úgy, működik, hogy SSH kapcsolatot létesít a tanítási folyamatban résztvevő összes géppel – ezt a korábban bemutatott inventory fájl segítségével teszi – majd elindítja rajtuk a felhasználó által létrehozott tanító szkriptet. Ezt az alábbi 8.7-es ábrán látható módon van lehetősége a felhasználónak paraméterezni.

```
python3 launch.py --workspace /shared/nfsdata \
--num_trainers 1 --num_samplers 0 --num_servers 1 \
--part_config part_data/ogbn-arxiv.json \
--ip_config inventory --ssh_port 12345  "python3 gputest.py"
```

8.7. ábra – Elosztott tanítás indításához szükséges parancs

Itt először definiálni kell azt a megosztott mappát, ami a szkript munkaterét adja meg, így az összes többi elérési út relatív útként adható meg ehhez a mappához. Ezt követően az egyes gépekben lévő grafikus kártyák száma adható meg, illetve a mintavételező és szerver folyamatok száma gépenként. Ebben az esetben ezek a DGL által meghatározott alapértelmezett értékek. Ezek után meg kell adni a particionált adatok elérési útját relatív módon. Végül a privát IP címeket tartalmazó fájl és az SSH során használt egyedi portot, valamint a tanító szkriptet elindító parancsot.

8.3 Elosztott tanítás tesztelése

Miután elindult az elosztott tanítás, a működéséről folyamatos visszaigazolást nyújt a szkript, a konzolra kiírt üzenetek formájában. Ez a konzolos kimenet látható az alábbi 8.8-as ábrán. Megfigyelhető, hogy az iteráció számláló négy soronként növekszik eggyel. Ez azért történik, mert négy gép vesz részt a tanításban, így minden egyes gépen futó folyamat aktuális állapotáról kapunk visszacsatolást. Ez látható a sorok elején lévő *world rank* nevű változó értékéből is. A kimenetek tartalmazzák továbbá a validálás pontosságát, a tesztek pontosságát, illetve az adott iterációba történő belépés óta eltelt időt.

```
World rank: 2, Epoch: 0, Validation Accuracy: 0.30845637583892616, Test Accuracy: 0.19825528763064768, Elapsed time: 3.54 seconds
World rank: 3, Epoch: 0, Validation Accuracy: 0.388911263256813, Test Accuracy: 0.2265843621399177, Elapsed time: 3.55 seconds
World rank: 0, Epoch: 0, Validation Accuracy: 0.2948993288590604, Test Accuracy: 0.21199901242696073, Elapsed time: 3.61 seconds
World rank: 1, Epoch: 0, Validation Accuracy: 0.11825503355704697, Test Accuracy: 0.07604312402271418, Elapsed time: 3.65 seconds
World rank: 3, Epoch: 1, Validation Accuracy: 0.5553765606121627, Test Accuracy: 0.4760493827160494, Elapsed time: 5.71 seconds
World rank: 0, Epoch: 1, Validation Accuracy: 0.4353020134228187, Test Accuracy: 0.35297506378075877, Elapsed time: 5.73 seconds
World rank: 2, Epoch: 1, Validation Accuracy: 0.4491275167785235, Test Accuracy: 0.36803555262941323, Elapsed time: 5.75 seconds
World rank: 1, Epoch: 1, Validation Accuracy: 0.33557046979865773, Test Accuracy: 0.2339725125504073 8, Elapsed time: 5.77 seconds
World rank: 2, Epoch: 2, Validation Accuracy: 0.4902013422818791 7, Test Accuracy: 0.41395769895481854, Elapsed time: 7.56 seconds
World rank: 3, Epoch: 2, Validation Accuracy: 0.5901463283662236, Test Accuracy: 0.5235390946502058, Elapsed time: 7.67 seconds
World rank: 0, Epoch: 2, Validation Accuracy: 0.4785234899328859, Test Accuracy: 0.3876224179079911, Elapsed time: 7.67 seconds
World rank: 1, Epoch: 2, Validation Accuracy: 0.4132885906040268 4, Test Accuracy: 0.3195621759525965, Elapsed time: 7.67 seconds
```

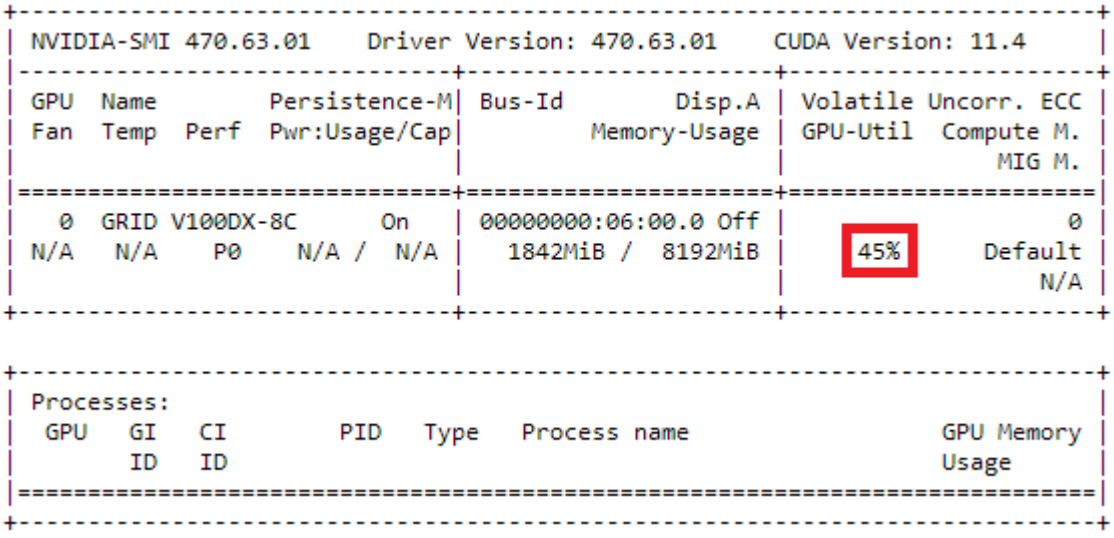
8.8. ábra – Elosztott tanítás konzolos kimenete

Végül, hogy megbizonyosodjak arról, hogy valóban a grafikus kártyák felhasználásával történik a tanítás, beléptem minden egyes gépre és az *nvidia-smi* parancs kiadásával konzolos formában kiíródnak a képernyőre a GPU-ra vonatkozó aktuális adatok. Ezek az alábbi 8.9, 8.10, 8.11 és 8.12-es ábrákon láthatók. A terhelésre vonatkozó százalékos formában megadott adatok, a piros színnel keretezett részeken találhatók.

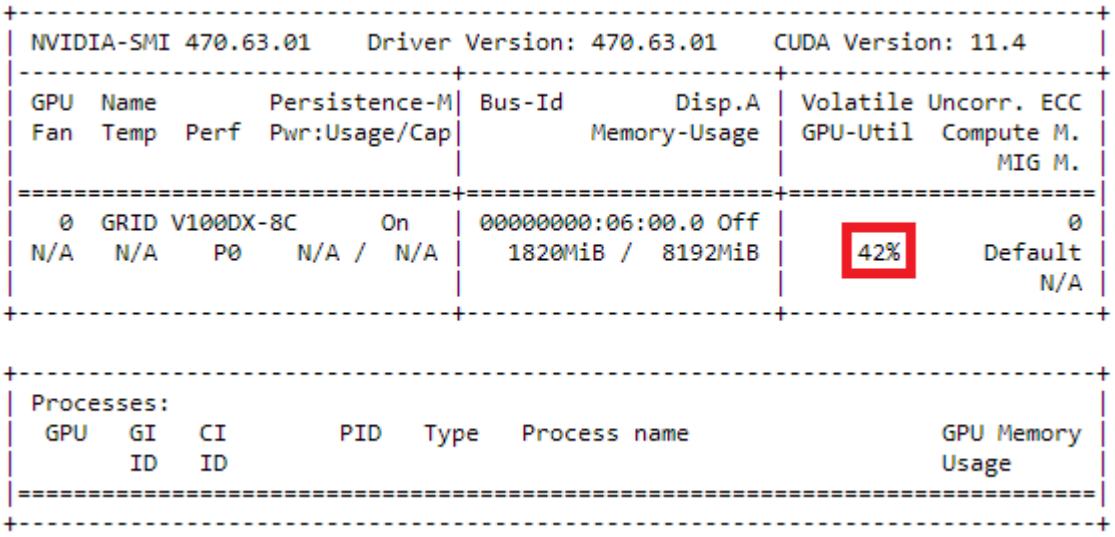
```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.63.01    Driver Version: 470.63.01    CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  GRID V100DX-8C      On   | 00000000:06:00.0 Off |                    0 |
| N/A   N/A    P0    N/A /  N/A |   1842MiB /  8192MiB |     39%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
+-----------------------------------------------------------------------------+
```

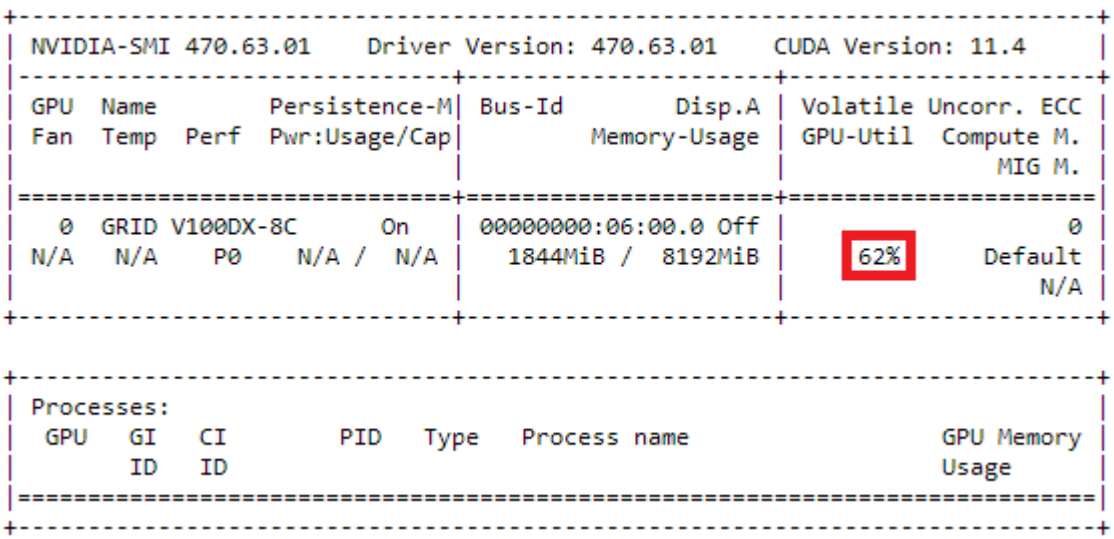
8.9. ábra – GPU terhelése a master gépen



8.10. ábra – GPU terhelése a worker-1 gépen



8.11. ábra – GPU terhelése a worker-2 gépen



8.12. ábra – GPU terhelése a worker-3 gépen

A fent található ábrák alapján elmondható, hogy a létrehozott referencia architektúra a tervezésnek megfelelő minőségben, illetve elosztott tanítási feladatok elvégzésére alkalmas módon kiépült. Továbbá a tesztelés céljából elkészített gráf neurális hálózatok elosztott tanítására írt szkript is az elvártak szerint működött.

9. TOVÁBBFEJLESZTÉSI LEHETŐSÉGEK

A referencia architektúra tervezése és implementálása során törekedtem arra, hogy a lehető legtöbb, napjainkban hozzáférhető legkorszerűbb technológiákat, illetve szemléletmódokat figyelembe véve készítsem el a gráf neurális hálózatok tanítására kiélezett rendszert. Ugyanakkor, mint minden rendszer első verzióját, úgy e referencia architektúrát is tovább lehet fejleszteni, hogy még sokoldalúbb, hordozhatóbb, átláthatóbb legyen.

Jelenleg nem áll a felhasználó rendelkezésére semmilyen eszköz, amivel vizsgálni tudja az infrastruktúra egyes elemeinek tansítás közbeni viselkedését, terheltségét. Így az egyik ilyen továbbfejlesztési irány a központosított monitorozás bevezetése lenne. Ez nagyban hozzájárulna ahhoz, hogy a felhasználó ellenőrizni tudja, hogy az aktuális tanító szkript mennyire hatékonyan használja ki a rendelkezésére álló erőforrásokat, valamint mennyire terheli a hálózatot gráf adatok gépek közötti továbbítása. Ezáltal magának a szkriptnek a hatékonysága is közvetetten növelhető, hiszen visszacsatolást kap a felhasználó a rendszer működés közbeni viselkedéséről.

A rendszer egyes részegységei által generált naplófájlokat a felhasználó jelenleg úgy tudja vizsgálni, ha az adott egység saját naplóját megnézi. Ez könnyen hosszadalmas folyamattá válhat már akkor is, ha az erőforrások száma a tízes nagyságrendbe lép. Ezt a kényelmetlennek bizonyuló felesleges lépést úgy lehet kiküszöbölni, ha ezeket a korábban említett naplófájlokat egy központi szerveren gyűjti a rendszer. Ezeket később akár szűrni is lehet valamilyen felhasználó által definiált paraméter alapján, hogy a sok automatikusan generált naplófájl között még hatékonyabban lehessen keresni.

Az előző felvetésből következik, hogy a jelenlegi, viszonylag egyszerűnek mondható fájlrendszer helyett, akár valamilyen komplexebb architektúrával rendelkező elosztott fájlrendszer is implementálható a referencia architektúrába. Ez nagyban növelné a rendszer hibatűrő, illetve a fájlrendszerre vonatkozó terheléselosztási képességeit is.

Végül, de nem utolsó sorban, mivel a gráf neurális hálózatok tanításához szükséges alkalmazásokat konténerizált formában implementáltam, így akár az egész referencia architektúra áthelyezhető egy Kubernetes környezetbe, ezzel tovább növelve a rendszer hordozhatóságát. Továbbá nagyban javítaná a rendszer rendelkezésre állását is, hiszen a Kubernetes képes automatikusan menedzselni a konténereket, így működés közben maximum pár percre eshet ki valamelyik részegység.

10. ÖSSZEFOGLALÁS

A Szakdolgozat során felvezettem, hogy hogyan épül fel egy referencia architektúra és milyen részei vannak, illetve bemutattam a gépi tanulás fejlődését a XX. századtól egészen napjainkig. Ismertettem a gráf adatszerkezetek struktúráját valamint alternatív ábrázolási módjait, továbbá, hogy e gráfokat hogyan lehet úgy alakítani, hogy illeszkedjenek egy neurális hálózat modellhez. Kitértem arra is, hogy ezek az úgynevezett gráf neurális hálózatok mely különböző tudományos területeken mutattak kiemelkedően nagy előrehaladást az utóbbi években. Foglalkoztam három elosztott mély tanulást támogató keretrendszerrel, illetve kiegészítő csomagokkal, melyek lehetővé teszik gráf neurális hálózatok tanítását is az egyes keretrendszereken. Ezt követően pár fontos szempont alapján összehasonlítottam a PyTorch, a TensorFlow, illetve a DistDGL rendszereket, majd az eredmények alapján kiválasztottam a legmegfelelőbb rendszert a feladat elvégzéséhez. Ezek után megterveztem a referencia architektúrát, ami képes elosztott módon gráf neurális hálózatok tanítására alkalmas infrastruktúrát létrehozni a Terraform és az Ansible segítségével. Ezeket követően az implementáció során megvalósítottam a tervezésnek megfelelően a teljes referencia architektúrát. Majd a tesztelést három fázisra bontva elvégeztem. A szempontok között volt az infrastruktúra automatikus kiépítésének vizsgálata, továbbá az, hogy az elkészült rendszer valóban képes-e elosztott módon gráf neurális hálózatok tanítására. Az ehhez szükséges tesztekhez kiépítettem egy tesztkörnyezetet is. Majd a különböző teszteseteket megvizsgálva arra jutottam, hogy a tesztek sikeresek voltak. Ezt követően a tesztelés végén kapott eredményeket dokumentáltam. Végül említettem néhány továbbfejlesztési lehetőséget is, melyekkel hordozhatóbbá, illetve hibatűrőbbé válhatna a referencia architektúra.

11. SUMMARY

During the Thesis, I explained how a reference architecture is built and what its parts are, and I presented the development of machine learning from the 20th century to the present day. I explained the structure of graph data structures and alternative representation methods, and how these graphs can be shaped to fit a neural network model. I also touched on that in which scientific fields these so-called graph neural networks have shown outstanding progress in recent years. I gathered three frameworks supporting distributed deep learning, as well as additional packages, which enable the training of graph neural networks on the individual frameworks. After that, I compared the PyTorch, TensorFlow, and DistDGL systems based on a few important aspects, and based on the results, I selected the most suitable system to complete the task. After that, I designed the reference architecture, which can create an infrastructure suitable for training graph neural networks in a distributed environment using Terraform and Ansible. After that, during the implementation, I created the entire reference architecture according to the design. Then I divided the testing into three phases. Among the aspects was the examination of the automatic construction of the infrastructure, and whether the implemented system is really capable of training graph neural networks in a distributed manner. I also built a test environment for the necessary tests. After examining the various test cases, I came to the conclusion that the tests were successful. After that, I documented the results obtained at the end of the testing. Finally, I also mentioned some further development options that could make the reference architecture more portable and fault-tolerant.

12. ÁBRAJEGYZÉK

13. TÁBLÁZATJEGYZÉK

14. IRODALOMJEGYZÉK

[1] Muller, G.: A reference architecture primer. Eindhoven Univ. of Techn., Eindhoven, White paper. 2008

[2] Eixelsberger, W., Ogris, M., Gall, H., Bellay, B.: Software architecture recovery of a program family. In Proceedings of the 20th International Conference on Software Engineering, 1998, pp. 508-511

[3] Sayfan, G.: Mastering kubernetes. Packt Publishing Ltd., ISBN: 1786461005, 2017

[4] "Orchestration & Scheduling Tools" (https://www.plutora.com/ci-cd-tools/orchestration-scheduling-tools), utoljára megtekintve: 2021.05.02.

[5] Brikman, Y.: Why we use terraform and not chef, puppet, ansible, saltstack, or cloudformation., 2016

[6] Belmont, J. M.: Hands-On Continuous Integration and Delivery: Build and release quality software at scale with Jenkins, Travis CI, and CircleCI. Packt Publishing Ltd., ISBN: 1789130484, 2018

[7] Emlen, S. T., Howland H. C., O'Brien, R. D.: "Frank Rosenblatt, July 11, 1928 — July 11, 1971", Cornell University

[8] Fradkov, A. L.: Early history of machine learning. IFAC-PapersOnLine, Vol. 53., 2020, pp. 1385-1390.

[9] White, T.: Hadoop: The definitive guide. O'Reilly Media, Inc., ISBN: 1449311520, 2012

[10] Esteva, A., Robicquet, A., Ramsundar, B., Kuleshov, V., DePristo, M., Chou, K., ... & Dean, J. A guide to deep learning in healthcare. Nature medicine, Vol. 25., 2019, pp. 24-29.

[11] Bhardwaj, R., Nambiar, A. R., Dutta, D.: A study of machine learning in healthcare. In 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), Vol. 2., 2017, pp. 236-241.

[12] Rzeszótko, J., Nguyen, S. H.: Machine learning for traffic prediction. Fundamenta Informaticae, Vol. 119., 2012, pp. 407-420.

[13] LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. nature, Vol. 521., 2015, pp. 436-444.

[14] "What is Deep Learning?" (https://www.mathworks.com/discovery/deep-learning.html), utoljára megtekintve: 2022.05.03.

[15] Rusk, N.: Deep learning. Nature Methods, Vol. 13., 2016, pp. 35-35.

[16] Mehlhorn, K., Naher, S., & Näher, S.: LEDA: A platform for combinatorial and geometric computing. Cambridge university press., 1999

[17] Singh, H., Sharma, R.: Role of adjacency matrix & adjacency list in graph theory. International Journal of Computers & Technology, Vol. 3., 2012, pp. 179-183.

[18] Biggs, N., Lloyd, E. K., Wilson, R. J.: Graph Theory, 1736-1936. Oxford University Press., 1986

[19] Xu, M.: Understanding graph embedding methods and their applications. SIAM Review, Vol. 63., 2021, pp. 825-853.

[20] "What are graph neural networks?" (https://bdtechtalks.com/2021/10/11/what-is-graph-neural-network), utoljára megtekintve: 2022.04.26

[21] Wu, L., Chen, Y., Shen, K., Guo, X., Gao, H., Li, S., ... & Long, B.: Graph neural networks for natural language processing: A survey. arXiv preprint arXiv:2106.06090., 2021

[22] Pradhyumna, P., Shreya, G. P.: Graph Neural Network (GNN) in Image and Video Understanding Using Deep Learning for Computer Vision Applications. In 2021 Second International Conference on Electronics and Sustainable Communication Systems (ICESC), 2021, pp. 1183-1189.

[23] "Google supercharges machine learning tasks with TPU custom chip." (https://cloud.google.com/blog/products/ai-machine-learning/google-supercharges-machine-learning-tasks-with-custom-chip), utoljára megtekintve: 2022.05.03.

[24] Abadi, M., Isard, M., Murray, D. G.: A computational model for TensorFlow: an introduction. In Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, 2017, pp. 1-7.

[25] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., ... & Zheng, X.: TensorFlow: A System for Large-Scale Machine Learning. In 12th USENIX symposium on operating systems design and implementation (OSDI 16) (pp. 265-283)., 2016

[26] Goldsborough, P.: A tour of tensorflow. arXiv preprint arXiv:1610.01178., 2017

[27] "Distributed training with TensorFlow" (https://www.tensorflow.org/guide/distributed_training), utoljára megtekintve: 2021.04.29.

[28] "tf.distribute.MirroredStrategy" (https://www.tensorflow.org/api_docs/python/tf/distribute/MirroredStrategy), utoljára megtekintve: 2021.04.29.

[29] "tf.distribute.TPUStrategy" (https://www.tensorflow.org/api_docs/python/tf/distribute/TPUStrategy), utoljára megtekintve: 2021.04.29.

[30] "tf.distribute.MultiWorkerMirroredStrategy" (https://www.tensorflow.org/api_docs/python/tf/distribute/MultiWorkerMirroredStrategy), utoljára megtekintve: 2021.04.29.

[31] "Parameter server training with ParameterServerStrategy" (https://www.tensorflow.org/tutorials/distribute/parameter_server_training), utoljára megtekintve: 2021.04.29.

[32] Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., ... & Ng, A.: Large scale distributed deep networks. In Proceedings of the 25th International Conference on Neural Information Processing Systems, Vol. 1., 2012, pp. 1223–1231.

[33] "tf.distribute.experimental.CentralStorageStrategy" (https://www.tensorflow.org/api_docs/python/tf/distribute/experimental/CentralStorageStrategy), utoljára megtekintve: 2021.04.29.

[34] Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., Dahl, G. E.: Neural message passing for quantum chemistry. In International conference on machine learning, 2017, pp. 1263-1272.

[35] Wang, M., Yu, L., Zheng, D., Gan, Q., Gai, Y., Ye, Z., ... & Zhang, Z.: Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs., 2019

[36] "DGL-LifeSci" (https://github.com/awslabs/dgl-lifesci), utoljára megtekintve: 2022.05.03.

[37] "DGL-KE" (https://github.com/awslabs/dgl-ke), utoljára megtekintve: 2022.05.03.

[38] Zheng, D., Ma, C., Wang, M., Zhou, J., Su, Q., Song, X., ... & Karypis, G.: Distdgl: distributed graph neural network training for billion-scale graphs. In 2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3), 2020, pp. 36-44.

[39] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... & Chintala, S.: Pytorch: An imperative style, high-performance deep learning library. Proceedings of the 33rd International Conference on Neural Information Processing Systems, 2019, pp. 8026–8037.

[40] "PyTorch" (https://www.nvidia.com/en-us/glossary/data-science/pytorch/), utoljára megtekintve: 2022.05.03.

[41] "Distributed communication package" (https://pytorch.org/docs/stable/distributed.html), utoljára megtekintve: 2022.05.03.

[42] "PyTorch on XLA devices" (https://pytorch.org/xla/release/1.11/index.html), utoljára megtekintve: 2022.05.03.

[43] Fey, M., & Lenssen, J. E.: Fast graph representation learning with PyTorch Geometric. arXiv preprint arXiv:1903.02428., 2019

[44] "PyG documentation" (https://pytorch-geometric.readthedocs.io/en/latest/), utoljára megtekintve: 2022.05.03.

[45] "PyTorch distributed overview" (https://pytorch.org/tutorials/beginner/dist_overview.html), utoljára megtekintve: 2022.05.05.

[46] "Distributed communication package" (https://pytorch.org/docs/stable/distributed.html), utoljára megtekintve: 2022.05.05.

[47] "Distributed data parallel" (https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html), utoljára megtekintve: 2022.05.05.

[48] "Getting started with distributed RPC" (https://pytorch.org/tutorials/intermediate/rpc_tutorial.html), utoljára megtekintve: 2022.05.05.

[49] "Writing distributed applications with PyTorch" (https://pytorch.org/tutorials/intermediate/dist_tuto.html), utoljára megtekintve: 2022.05.05.

[50] "PyTorch Geometric vs Deep Graph Library" (https://www.exxactcorp.com/blog/Deep-Learning/pytorch-geometric-vs-deep-graph-library), utoljára megtekintve: 2022.05.08.

[51] "What is Terraform?" (https://www.terraform.io/intro), utoljára megtekintve: 2022.05.11.

[52] "Node Property Prediction" (https://ogb.stanford.edu/docs/nodeprop/#ogbn-arxiv), utoljára megtekintve: 2022.12.06.

[53] "Distributed Node Classification" (https://docs.dgl.ai/tutorials/dist/1_node_classification.html#sphx-glr-tutorials-dist-1-node-classification-py), utoljára megtekintve: 2022.12.06.

[54] "Distributed training" (https://github.com/dmlc/dgl/tree/master/examples/pytorch/graphsage/dist), utoljára megtekintve: 2022.12.06.

[55] "Launching tool for DGL distributed training" (https://github.com/dmlc/dgl/blob/master/tools/launch.py), utoljára megtekintve: 2022.12.06.