Debreceni Egyetem
Informatikai Kar

Hibrid fordítás HPC környezetben

Témavezető:
Dr Kovács László
adjunktus

Készítette:
Palkovics Dénes
mérnökinformatikus BSc

Debrecen, 2019

Tartalomjegyzék

Bevezetés

A Deep learning avagy a mélytanulás a gépi tanulás neurális hálózatokat alkalmazó technikája napjaink egyik legnépszerűbb technológiája, melynek fejlesztését számos kutató intézmény és nagyvállalat végzi. Megjelent a polgári életben is. Felhőalapú alkalmazások hátterében működik, többek között a Google®online szolgáltatásaiban és már alkalmazzák a hordozható eszközök, táblagépek és mobiltelefonok biometrikus személyazonosításra.

A mélytanulás használata rengeteg számítási kapacitást igényel —ez bizonyos alkalmazások esetén költséges és nagy méretű számítógépeket jelent— így sok helyen kiszorul a használata, illetve telemetria formájában érhető el csak. Az 5G-nek hála, komolyabb alkalmazásokhoz is felhasználható lesz a felhő technológián működő gépi tanulás. Ennek ellenére igény volna arra, hogy helyben elérhető legyen ez a technika. Ilyen lehet az orvosi alkalmazás, ahol számít a magas rendelkezésre állás vagy az autonóm robotok és önvezető autók, melyeknek bizonyos helyzetekben ott is kell működniük, ahol nincs rádiókapcsolat vagy internet elérés, nem is beszélve a hordozható eszközök olyan funkcióiról melyek használata frekventált.

Mikor fellendült a kutatása, legjobb hardverek erre a feladatra a fejlett grafikus kártyák voltak, melyek processzorainak számítási kapacitása és utasításkészlete alkalmassá tette, hogy a neurális hálózatokkal kapcsolatos számításokat hatékonyan végezze. Azonban az iparban megjelentek speciális hardverek kifejezetten neurális hálózatok futtatására optimalizálva, hogy ki tudják elégíteni a megnövekedett számítási igényt, amit a technológia egyre szélesebb körű bevezetése generál. A fenntarthatóság végett azonban nem szabad kihasználatlanul hagyni a már meglévő erőforrásokat. Témavezetőm, Dr. Kovács László projektje, a HuSSar nevet viselő hibrid architektúrájú szuperszámítógép is részben ebből az indíttatásból született. A HuSSar olyan hardverekből tevődik össze, melyek szerverek komponenseként régóta ott van az iparban. Egyedi hibrid architektúrája lehetőséget ad arra, hogy a neurális hálózatokkal kapcsolatos különféle számításokat olyan processzoron futtassuk, melyek azt optimálisan képesek végrehajtani így jelentős teljesítménynövekedés érhető el vele. Ehhez szükséges még egy olyan keretrendszer, mely képes ezeket a számításokat ekképpen optimalizálni.

A Deep learning a szemem láttára fejlődött ki a kezdeti kísérletekből, a mindennapi életben is használt csúcstechnológiává. Úgy érzem leendő szakemberként most van arra alkalmam, hogy közelebbről is megismerkedjek vele, kivehessem részem a fejlesztésében. Észrevettem,

hogy az ipar is nagy erőkkel fejleszti, ezért úgy vélem, hogy ez a tudás számomra nagyon jövedelmező lehet a munkaerőpiacon is. Témavezetőm fejlesztésével, a HuSSar-ral az egyik általa tartott egyetemi kurzus során találkoztam, mikor azt megmutatta nekünk. Beszélt az eszköz felépítéséről és arról, milyen célból kezdte a fejlesztést. Továbbá látom unokaöcsém sikereit, aki ezen a területen kutat. Ezek miatt éreztem úgy, hogy ebben a témában szeretnék dolgozni, ha lehet az egyetemi tanulmányaim után is.

Ebben a szakdolgozatban szeretnék beszámolni, mit sikerült megtudnom a Deep Learningről, milyen új megvalósítások születtek az iparban és hogyan boldogultam ezekkel a technológiákkal. Eredeti célkitűzésem az Intel®fejlesztés alatt álló *nGraph* nevű környezetének fordítása és telepítése volt a fentebb említett HuSSar-ra. Ez a keretrendszer kifejezetten a neurális hálózatok olyan módú futtatására lett fejlesztve, ahol a hardver több típusú processzort tartalmaz. Ezzel szerettük volna, ha sikerül a mélytanulás során alkalmazott neurális hálózatokat az összes processzortípuson elosztottan tanítani és futtatni. Hosszas próbálkozás után sem sikerült ez ügyben eredményt elérni, azonban a munka során megismerkedtem más az Intel®által fejlesztett és fejlesztés alatt álló eszközeivel.

1. fejezet

Neurális hálózatok és a Deep Learning

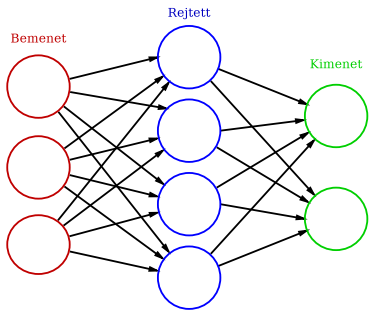
Ebben a fejezetben szeretném összegezni megszerzett tudásomat a neurális hálózatokról és a Deep Learning-ről, magyarul mély tanulásról.

1.1. A neurális hálózatok elmélete

Olyan számítási modellel, amelynek alapját az idegrendszer hálózata adja először Warren McCulloch és Walter Pitts 1943-ban foglalkozott az „A Logical Calculus of the Ideas Immanent in Nervous Activity” című publikációjukban. Később Donald Hebb tanulással kapcsolatos megfigyeléseivel elindultak a mesterséges neurális hálókkal kapcsolatos kísérletezések.[2]

A mesterséges neurális hálózatok egy viszonylag egyszerű modellen alapulnak. Minden neuron a hozzá kapcsolódó neuronok ingereinek összessége alapján ingerli a többi neuront melyekhez ő kapcsolódik, ekképpen az ingerület egy irányba halad a kapcsolatok mentén.

Hogy a hálózat áttekinthető legyen, rendezzük a neuronokat rétegekbe úgy, hogy egy réteg neuronjai az ingerületet a közvetlen felső réteg neuronjaitól kapja, és a válasz ingert a közvetlenül alatta lévő réteg neuronjainak továbbítja.



1.1. ábra. neurális hálózat réteges szerkezete [1]

[1] forrás: https://en.wikipedia.org/wiki/Artificial_neural_network

A 1.1 ábrán a csúcsok jelentik a neuronokat és az élek a szinapszisok, melyeken az ingerület vándorol. Egy hálózat 3 nagyobb részre tagolódik: **a)** bemeneti réteg **b)** rejtett rétegek **c)** kimeneti réteg. A bemeneti réteg csúcsai legtöbbször az adatot reprezentáló konstansok jelentik, tehát az egy egyszerű vektor. Egy neuronban két művelet történik: a bementek összegzése és egy aktiváló függvény kiértékelés. Az összegzést a felsőbb rétegből érkező jelekre elvégezzük:

$$s = \sum_i w_i x_i = \vec{w} \cdot \vec{x}$$

ahol x_i a felső réteg i.-ik neuronjának kimenete, w_i az i.-ik neuron szinapszisához tartozó súly, mellyel a szinapszis "erősségét" határozzuk meg. Az "s" összeghez hozzáadunk még egy b értéket, a neuron aktiválási küszöbértéke lesz. Az aktivációs függvény adja a neurális hálózat kimenetét, paramétere $s+b$. A neurális hálózatok fejlesztésekor sokféle függvényt találtak alkalmasnak aktivációs függvény gyanánt. Közös jellemzőjük, hogy inflexiós pontjuk $x = 0$ helyen van, illetve 0-ban nem deriválható függvények esetén a töréspont esik ide.

A szemléletesség kedvéért tekintsünk meg az egyrétegű perceptront, vagyis egy egyetlen rétegből álló neurális hálózatot k darab neuronnal. A bemenet legyen az $\vec{x} = (x_1, \ldots, x_n)$ vektor (a gyakorlatban bemeneti rétegként szokták hívni). A szinapszisok súlyait a $W = \{w_{ij} : i = 1 \ldots n, j = 1 \ldots k\}$ mátrix ($\vec{w}_i$ az i. bemeneti adatból kiinduló szinapszisokhoz tartozó súlyok vektora lesz), a neuronok küszöbértékeit a $\vec{b} = (b_1, \ldots, b_k)$ tartalmazza. Az aktivációs függvény f. A hálózat kimenetét, vagyis a $\vec{y} = (y_1, \ldots, y_k)$ elemeit megkapjuk a következőképpen:

$$y_i = f(\vec{w}_i \cdot \vec{x} + b_i)$$

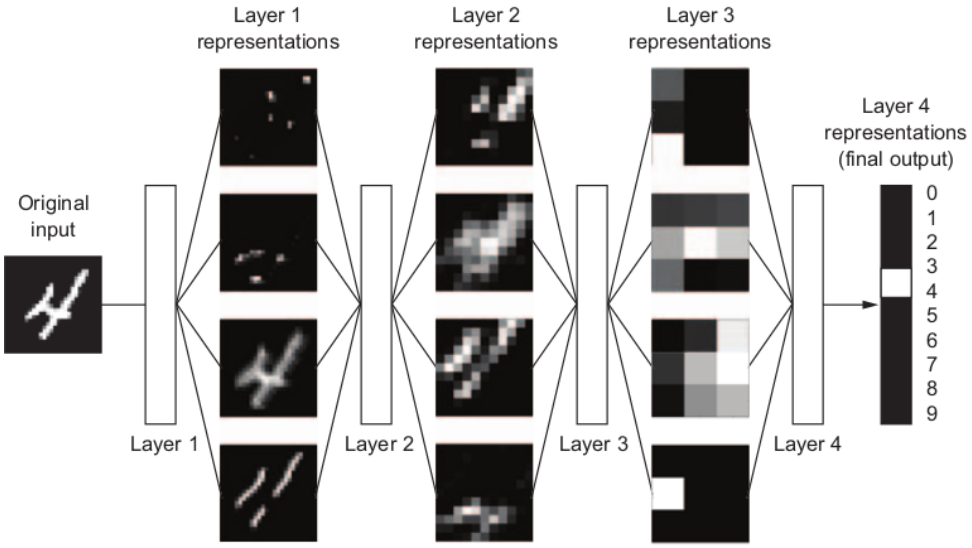
A fentiekből látszik, hogy a hálózat tervezésénél annak négy tulajdonságát kell meghatároznunk: 1. a rétegek és azok neuronjainak számát 2. a neuronok aktivációs függvényét (rétegenként egy típusú függvény az összes neuronra) 3. a szinapszisok súlyát (W) 4. a neuronok aktiválási küszöbét ($\vec{b}$). Minden réteghez külön W mátrixot és $\vec{b}$ vektort kell meghatározni. **Az egyszerűség kedvéért az egy réteghez tartozó W-t és $\vec{b}$-t együttesen nevezzük a réteg *súlyainak***. Ez nagyon sok külön meghatározandó változót jelent, tehát csak az 1. és 2. tulajdonság meghatározása elvárható. Kell egy algoritmus, mellyel az egész hálózathoz tartozó paraméterek sokasága – vagyis minden réteg súlyának paraméterei – meghatározható.

1.2. Deep Learning

A Deep Learning-ről szerzett tudásom javát F. Cholett könyvéből[1] szereztem, melynek a témához kapcsolódó részleteit alább bemutatom.

A gépi taunlás egy teljesen más programozási paradigmát jelent, ugyanis a klasszikus programozás során a feldolgozandó adatokhoz a programozó adja az adat feldolgozásának szabályait, amit végig követve a gép kiszámítja a kívánt eredményt. Ezzel szemben a gépi tanulás során a programozó az adathoz a kívánt eredményt adja meg, amiből a gép felállítja a megoldáshoz vezető szabályokat.

A Deep Learning más néven a mély tanulás a gépi tanulás egy fajtája. Chollet szerint a név arra utal, hogy a kezdeti adaton több transzformációt végrehajtva egymás után egyre közelebb kerülünk egy olyan reprezentációhoz, ami megfelel a kívánalmainknak. Ezzel kontrasztban beszélhetünk sekély tanulásról, amikor kevés, egy vagy két transzformáció után kapjuk meg az adat megfelelő reprezentációját.A neurális hálózatok rétegeltsége adja a *mélységet* a gépi tanulásban. Eredeti elgondolás szerint minden egyes neuron-réteg egyre összetettebb tulajdonságokat ismer fel a bemeneti adatból. Valójában a rétegenkénti transzformációk egyre kisebb összetettségű hipotézis térbe visznek át, a reprezentáció egyre kevesebb – a felhasználó számára fölösleges – információt tartalmaz. Minél több réteg van a hálózatban, annál *mélyebb* a modell.



1.2. ábra. Írott szám hozzárendelése az ábrázolt számértékhez
[2]

Itt kapcsolódik össze a neurális hálózat és a mély tanulás. Az 1.1 alfejezetben kifejtettem, hogy a neurális hálózat szinapszisainak paraméterezéséért felelős $\vec{w}$ súlyok és a neuronok küszöbszintjének állítására szolgáló $\vec{b}$ vektorok összes koordinátájának száma hatalmas lehet, — alkalmazástól függően több százezer, akár millió, egymástól független változóról beszélünk— tehát beállításukhoz valamilyen algoritmusra van szükség. Ezért a neurális hálózatok másik

[2]Forrás:[1]

komponense, egy tanulási algoritmus, mely beállítja ezen paramétereket. Négy megközelítés létezik, amikor gépi tanulásról van szó.

Ellenőrzött tanulás során a neurális hálózatnak felcímkézett adatokat adunk meg, tehát olyan y értéket rendelünk az x mintákhoz, amilyet szeretnénk, hogy a hálózat produkáljon. Ezen $z = (x, y)$ összerendezések halmazát *tanítókészletnek* hívjuk. A hálózat leképezi az adatot a meghatározott reprezentációvá. A tanuló algoritmus ebből és a címkéből egy *veszteség függvény* kiszámításával meghatározza, hogy mekkora az eltérés, a valamilyen értelemben vett távolság a kapott és az elvárt eredmény között. Ez alapján frissíti a $\vec{w}$ súlyokat és $\vec{b}$ vektorokat.

Ellenőrizetlen tanulás, mely során az adatokat nem címkézzük fel, hanem arra vagyunk kíváncsiak, hogy miféle összefüggések állnak fenn közöttük. Ezt a módszert adatbányászat során alkalmazzák.

Az *Önellenőrzött tanulás* hasonló az ellenőrzötthöz, azonban az adatok felcímkézését nem emberi erővel végezzük, hanem az adatokból állítjuk elő valamilyen heurisztikát felhasználva. Egyik alkalmazási területe az autóenkóderek tanítása.

A *Megerősítéses tanulás* egy újfajta megközelítése a neurális hálózatok alkalmazásának. Ennél a metodikánál a hálózatot egy ágens alkalmazza, így a hálózat bemenete az ágens által megfigyelt környezet a kimenete pedig valamilyen cselekedet, beavatkozás és tanítás során az ágens igyekszik valamilyen környezetbeli értéket maximalizálni. Gyakori alkalmazás valamilyen játékot játszó ágens, ahol azt tanulja, adott helyzetekre milyen reakcióval tudja maximalizálni játékbeli pontszámát. Vizsgálódásomat az *ellenőrzött tanulásra* korlátoztam, így a továbbiakban ennek tükrében folytatom dolgozatomat.

1.3. Függvények, algoritmusok

Az alábbiakban szeretném megfogalmazni a neurális hálózatokban alkalmazott tipikus függvényeket és algoritmusokat.

1.3.1. Neuronok aktivációs függvényei

Mint korábban kifejtettem minden neuron kimenete egy függvény kiértékelése, melynek paramétere a bemenetek súlyozott összege. Ezt a függvényt hívjuk aktivációs függvénynek.

A szigmoid függvény Az utolsó, kimeneti neuronok rétegének aktivációs függvényeként alkalmazzuk, ahol a várt eredmény egyetlen valószínűségi érték. Ez bináris osztályozási problémák esetén alkalmazandó, tehát a program célja, hogy egy bemeneti adatról eldöntse, hogy az egy bizonyos kategóriába esik-e vagy sem, illetve erről mekkora "magabiztossággal" döntött.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{1.1}$$

A softmax függvény A kimeneti réteg aktivációs függvénye. D dimenziójú x vektorok koordinátáit normalizálja, másként fogalmazva egy tetszőleges D elemű szám n-est azon D elemű n-esek halmazába képezi, melyek elemeinek összege 1. Így tehát a x koordinátái egy diszkrét valószínűségi eloszlás értékkészlete.

$$\sigma(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{D} e^{x_i}}, \quad i = 1, \ldots, D \tag{1.2}$$

Éppen ezért többosztályos problémáknál használatos.A hálózat egy mintára adott válasza annak a diszkrét valószínűségi változónak az eloszlása, mely a minta egy adott kategóriába tartozásának a valószínűségét adja meg.

ReLu Teljes néven *rectified linear unit* függvény vagy közismerten rámpafüggvény a rejtett neuron rétegek aktivációs értéke szokott lenni. Először 2011-ben mutatták be, hogy hatékonyabban taníthatóak a neurális hálózatok, mintha csak szigmoid függvényt használnánk neuronok aktivációs függvényeként.[13]

$$f(x) = max0, x \tag{1.3}$$

(a) Szigmoid függvény (b) ReLu függvény

1.3. ábra. Aktivációs függvények

1.3.2. Veszteség függvények

A neurális hálózat tanításához szükséges meghatároznunk egy veszteségfüggvényt, mely megadja a hálózat kimenetének átlagos eltérését az elvárt eredményhez képest adott súlyok mellett. Formálisan egy $c : W \times B \mapsto \mathbb{R}^+$ függvény, W, B, a súlyokat és küszöbértékeket egybefogó vektorok halmaza. Ezen többváltozós függvény képe a *veszteségfelület*.

Átlagos négyzetes hiba Többosztályos problémánál a neurális hálózat egy valószínűségi változó eloszlása az összes osztályon. $\vec{y}$ vektor a hálózat válasza a $\vec{x}$ bemenetre, $\vec{y'}$ pedig a kívánt kimenet vektora –egy egységvektor, melynek 1 értékű koordinátája reprezentálja a megfelelő osztályt. Erre az esetre olyan veszteségfüggvényt alkalmazhatunk mely ekvivalens a $(x, y) \in Z$

tanítási készletből számított MSE átlagos négyzetes hibáinak átlagával.

$$MSE(\vec{y}, \vec{y'}) = \frac{1}{n} \sum_{i=1}^{n} ||y_i - y'_i||^2$$

$$c(w, b) = \frac{1}{m} \sum_{j=1}^{m} MSE(\vec{y}_j, \vec{y'}_j)$$

A fenti egyenletekben n az kimeneti vektor dimenziója, m a tanító minták száma.

Bináris keresztentrópia Bináris osztályozási feladatnál olyan módszert használhatunk a veszteség vagy hiba érték számításához, melynél az y és y' minták kereszt entrópiáját határozzuk meg

$$H(y, y') = -\sum_{x \in X} y(x) \log_2 y'(x)$$

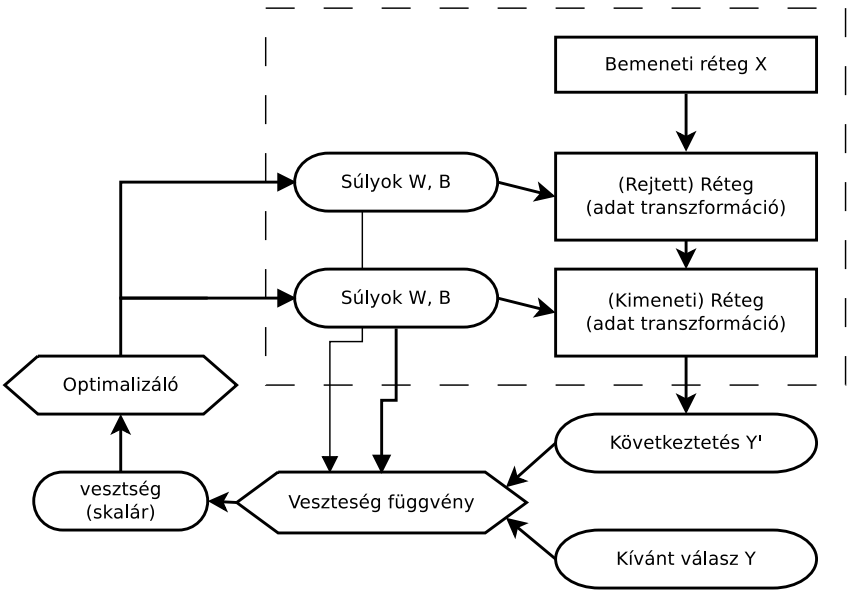
Kategorikus keresztentrópia Az átlagos négyzetes hiba helyett veszteségfüggvényként alkalmazhatjuk a keresztentrópia Többosztályos feladat

1.3.3. A hálózat tanító algoritmusai: az optimalizálók

Korábban említettem, hogy egy neurális hálózat tanításán azt az eljárást értjük, amely során a hálózat paramétereit változtatjuk. A tanítást egy optimalizáló algoritmus végzi, melynek bemenete az előző fejezetben tárgyalt veszteségfüggvények valamelyikének eredménye, kimenete pedig a hálózat új paraméterei. A gyakorlati megvalósítás során általában valamilyen gradiens alapú szélsőérték kereséssel minimalizáljuk a hibát. Mivel a veszteségfelület értelmezési tartománya többdimenziós vektorok halmaza, a függvény szélsőértékét a gradiens fogalmával definiálhatjuk, ami tulajdonképpen a derivált általánosítása. Tehát egy olyan eljárást kell alkalmazni, amely meg keresi azon w_l és b_k paramétereket, ahol a $c(\vec{w}, \vec{b})$ függvény értéke a legkisebb.

Ezt iteratív módon visszük véghez. Az iterációkat *eposzoknak* (angolul: epoch) nevezi a szakirodalom. Minden eposzban az optimalizáló hatására a kimeneti hiba csökken, a hálózat *következtetése* egyre közelebb kerül az elvárthoz. A tanítás sebesség (angolul: learning rate) az optimalizálók egy másik paramétere, mellyel azt befolyásoljuk, hogy egy eposzban a hálózat paramétereit mekkora mértékben változtatjuk. Ezen paraméter változtatásánál fontolóra kell vennünk két tényezőt. A gradienst Nagy tanítási sebesség gyorsabb gradiens csökkenést eredményez

Sztochasztikus gradienscsökkentés



1.4. ábra. A neurális hálózat tanításának folyamata

négzetes közép visszaterjesztése

1.3.4. A kernel-trükk

2. fejezet

Mély tanuló alkalmazások fejlesztése

A mély tanulás, azon belül is a mesterséges neurális hálózatok gyors fejlődését és részben népszerűségét a hozzá készített programozási keretrendszereknek köszönheti, melyek java szabad hozzáférésű. Ezek a keretrendszerek arra hivatottak, hogy támogassák a neurális hálózatok fejlesztését új programozási módszertant adva. Már létező programozási nyelvekre épülnek, leginkább python-ra. Ezekben a keretrendszerekben egyszerűen implementálhatunk neurális hálózatokat úgy, hogy egyfajta nyelvi eszközkészletet adnak neurális hálózatok definiálására.

2.1. TensorFlow

2.2. Keras

A Keras egy python nyelven írt programkönyvtár, vagy ahogy önmagát hívja ”magas szintű neurális hálózat API”[8]. Érdekessége, hogy más olyan keretrendszerekkel együttesen használható, amelyek a Keras-hoz hasonlóan magas absztrakciós szinten biztosítják a hálózatok implementálását, mint például a TensorFlow. A Keras-ról szerzett tudásom javát Chollet könyvéből és a keretrendszer dokumentációjából szereztem.[1][8].

2.2.1. Neurális hálózat definiálása

Keras-ban egy neurális hálózatot *model*-nek hívunk (gyakran máshol is így neveznek egy konkrét neurális hálózatot). Egy *model* létrehozásához a `Keras.models` modulban definiált metódusokkal lehetséges. A keretrendszerben az adatokat tenzorokként kell reprezentálnunk, ezért érdemes a *numpy*[1] nevű python csomaggal együtt használni. A keretrendszer tetszőleges alakú tenzorokat képes kezelni, tehát nincs megkötés arra vonatkozóan, hogy egy réteg bemenete

[1] lásd:`https://numpy.org/`

vektor, mátrix vagy kiterjedtebb struktúrában – magasabb dimenziójú tenzorban – szerepeljen. A következő kódokban szeretném szemléltetni a Keras használatának módját.

A 2.2.1 kód egy két rétegből álló neurális hálózat definiálását mutatja be.

2.1. Listing. Neurális hálózat rétegeinek definiálása

```
1 from keras import models
2 from keras import layers
3
4 network = models.Sequential()
5 network.add(layers.Dense(15, activation='relu', input_shape=(784,)))
6 network.add(layers.Dense(10, activation='softmax'))
```

Ezen lista 4. sorában inicializáljuk a hálózatot, az utána következő sorokban új neuron rétegeket adunk a hálózathoz.Keras-ban különböző előre definiált rétegkapcsolatok vannak. Itt a `layers.Dense` osztály egy sűrűn kapcsolt réteget implementál. A réteg bemenetként $28 * 28 = 784$ elemű vektorokból álló mátrixot tud fogadni, másik dimenziója nem meghatározott. Ez a kötegelt adatfeldolgozás szempontjából fontos, tehát a bemeneti vektorok folyamát egyik dimenziójában nem meghatározott méretű tenzorral definiáljuk. A réteg kimenete egy 32 dimenziós vektor. A következő rétegnél nem adtuk meg a bemenet méretét, a keretrendszerben ez implicit módon rendelődik a réteghez.

Mindkét réteghez tartozik egy *activation* paraméter, mely string típus kell legyen, és a réteg neuronjaihoz tartozó aktivációs függvényt adja meg. A példában a `'relu'` és `'softmax'` string a 1.3 fejezet (1.3) és (1.2) függvényére utal, azaz ezen bemeneti paraméterek esetén olyan `Dense` objektum jön létre, mely ilyen aktivációs függvényekkel rendelkező neuron réteget valósít meg.

A `Keras.layers` modulban A `Dense` osztályon kívül implementálva vannak konvolúciós, összefésülő, zaj, stb. rétegek, melyek a fenti módon tetszés szerint egymásra szervezhetőek a keretrendszerben, így szinte tetszőleges hálózat alakítható ki.

2.2.2. Hálózat betanítása és következtetés futtatása

A megalkotott hálózatot a `network` objektum definiálja. Hogy tanítható legyen el kell látni egy optimizálóval és egy veszteségfüggvénnyel, melyek együtt adják a hálózatot betanító algoritmust. A `compile()` metódus „összerakja” a neurális hálózatot a betanítóval, a `fit()` pedig elvégzi a tanítást, és *epoch*-ok számának megfelelő méretű tömböket tartalmazó objektum referenciájával tér vissza, mely tömbökben össze vannak gyűjtve a veszteségfüggvény értékei és a felhasználói metrikák eposzonként.

2.2. Listing. Hálózat betanítása

```
network.compile(optimizer = 'sgd',
loss = 'mean_squared_error',
metrics = ['acc'])

history = network.fit(train_datas,
          train_labels,
          epochs=20,
          batch_size=512)
```

A `train_datas` és `train_labels` a tanítókészletet alkotó minták és azok címkéi, a várt kimeneti értékek. Ezek elemszáma kötelezően meg kell egyezzen. A *batch_size* paraméterrel, ami az egy *epoch*-ban egyszerre feldolgozandó adatokat jelenti. A betanítás végén a `network` egy *betanított*, a célfeladat megoldására felhasználható neurális hálózat lesz. Alkalmazásához a `predict()`metódus használatos,mely a paraméterként megadott bemeneti adatokhoz tartozó predikciókkal tér vissza.

2.3. Listing. Következtetés

```
result = network.predict(datas)
```

2.2.3. Hatékonyságvizsgálat

3. fejezet

Speciális platformok megjelenése

3.1. hybrid deep learning

A ha

3.2. nGraph

A most leírtak alapját az nGraph hivatalos dokumentációja adja[5]. Az nGraph az Intel®által fejlesztett programkönyvtár és egy futtatási környezet/fordító készlet melyet mély tanuló projektekhez. Legszembetűnőbb tulajdonsága, hogy képes többféle hardver architektúrán futtatni és beépíthető számos keretrendszerbe. Ezek azok a jellemzők, amiket kerestünk témavezetőmmel, hogy mély tanulást tudjunk végeztetni hatékonyan a *HuSSar-on*. Ezen túlmenően a jövőbeli fejlesztések az iparban egyre jelentősebben igényelik, hogy a modern MI rendszerek skálázhatóak legyenek, mert egyrészt a neurális hálók egyre komplexebbé válnak, másrészt az általuk feldolgozott adatok mennyisége is rohamosan növekszik.

Jelenleg két bevett gyakorlat van a mély tanulás felgyorsítására:

1. **Dedikált hardver tervezése a mély tanulással kapcsolatos számításokhoz** – Sok vállalkozás tervez *Alkalmazásspecifikus integrált áramköröket* (ASIC) neurális hálózatok betanítására és futtatására.

2. **Szoftver optimalizáció** – Programkönyvtárakat tartalmazó fejlesztési keretrendszerek fejlesztése, melyek képesek a hálózatokkal kapcsolatos számításokat több szálon, optimalizáltan futtatni. Az nGraph, mint fordító is egy ilyen megoldás.

3.2.1. Motiváció

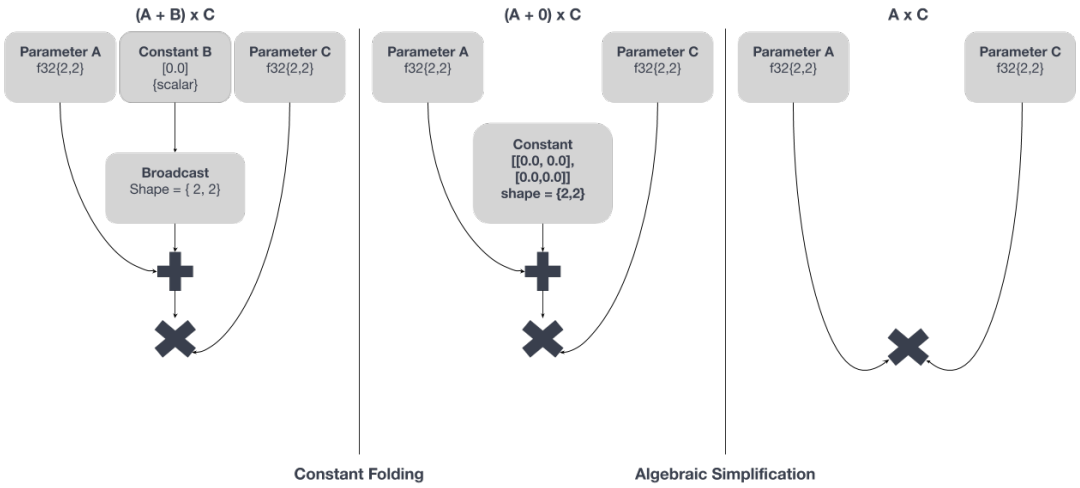
A mai legmodernebb szoftveres megoldás mély tanulásra az, ha integrálunk kernel könyvtárakat[1] mély tanulási keretrendszerekbe. Ilyen integráció lehet például ha a Tensorflow keretrendszer alatt az Nvidia CuDNN könyvtárát használjuk. A kernel könyvtárak egy adott célarchitektúrára optimalizált kernelekből és egyéb műveleti szintű optimalizálásokból állnak, ezekkel érve el teljesítménynövekedést. Azonban a kernel könyvtáraknak három fő problémája van:

1. Nem biztosítanak gráf szintű optimalizálást
2. A keretrendszerek integrációja a kernel könyvtárakkal nem skálázható
3. A szükséges lefordított kernel könyvtárak száma növekszik, ahogy új processzorok jelennek meg

Az nGraph fordító megoldás az első két problémára, és a *PlaidML*-el ötvözve kiküszöbölhető a harmadik probléma.

3.2.2. Gráf szintű optimalizálás

Az nGraph dokumentációjában áll egy példa arra vonatkozóan, hogyan lehet, hogy egy mély tanulási keretrendszer integrálva egy kernel könyvtárral a számításokat optimálisan végzi, mégis a számítási gráf a csúcsait alkotó műveletek szempontjából mégsem optimális. A fenti ábra



3.1. ábra. két optimalizálás módszer: konstans összehajtás és algebrai egyszerűsítés a gráfon. [2]

[1] Programkönyvtár, mely neurális hálókkal kapcsolatos elemi, *mag* függvényeket tartalmazzák
[2] forrás: [5]

bemutatja, hogyan egyszerűsíthetünk az A,B és C tenzorokat feldolgozó, $(A+B)*C$ tenzorműveletet végrehajtó számítási gráfon. Fordítási időben megállapítható, hogy B egy skalár konstans, így a *konstans összehajtásnak* nevezett optimalizálás elvégezhető, és a 2 dimenziós vektorrá való kiterjesztés művelete elhagyható (helyette inkább közvetlenül létrehozunk egy 2×2 tenzort). Ebben a példában $B = 0$ skalár volt, így a belőle létrejött tenzor egy nullmátrix, így az semleges a kifejezés kiértékelésének szempontjából. *Algebrai egyszerűsítést* végezve az $(A + 0) * C$ leegyszerűsíthető az $A * C$ kifejezésre, így összesen két csúccsal csökkentettük a gráfunkat. Ez az optimalizáció tehát a számítási gráf szintjén lett elvégezve. Így belátható, hogy a mély tanulási keretrendszerbe integrált kernel programkönyvtárak nem optimális futást végeznek, hiába a műveletek szintjén elért optimalizálás.

3.2.3. Skálázható keretrendszer integráció

Ahogy gyarapodik a mély tanuláshoz használható gyorsítókártya architektúrák és keretrendszerek száma, a meglévő mély tanulást alkalmazó fejlesztési platformok bővítése egyre több munkát igényel és egyre nő a hibák megjelenésének a valószínűsége. Az integráció kapható készen, szakértő fejlesztőcsapatoknak kell implementálnia. Minden új keretrendszert manuálisan kell integrálni a meglévő hardverek kernel könyvtárával és minden újonnan megjelenő hardvercsalád meghajtó programkönyvtárát be kell integrálni egyesével a meglévő keretrendszerekbe. Ez a munka önmagában is hatalmasra tud nőni, de egy sok eszközből álló összeállítás nagyon törékeny és költséges a fenntartása. Az nGraph úgy oldja meg ezt a problémát, hogy ún. *hidakat* alkalmaz, amikkel integrálható valamelyik mély tanulási keretrendszerbe. A híd megkapja a keretrendszerben megalkotott számítási gráfot vagy ahhoz hasonló struktúrát és átalakítja egy ún. *közbenső reprezentációvá*[3]. Ezzel kaptunk egy egységes, platformfüggetlen számítási gráfot, így nem kell egy új programkönyvtárat beintegrálni minden egyes meglévő keretrendszer alá, elegendő csak az, hogy az nGraph-ban, mint programkönyvtárban implementált *primitív műveleteket* támogassa az új programkönyvtár.

3.2.4. Növekvő kernel szám

Egy kernel könyvtár integrálása egyszerre több mély tanulási keretrendszerrel nehéz feladat és egyre komplexebbé válik, ahogy növekszik az optimális teljesítményhez szükséges kernelek száma. Régen a mély tanulással kapcsolatos kutatások egy kis számú *primitív* számítást használtak, mint a konvolúció, általános mátrixszorzás, stb. Az MI kutatás előrehaladtával és az ipari mély tanuló alkalmazások továbbfejlesztésével, a szükséges kernelek száma (k) exponenciálisan nő. Ez a szám a processzor architektúrák számán (h), adattípusokon (t), műveleteken (p) és az egyes paraméterek számosságán (p) alapul ($k = h \times t \times m \times p$).

[3]IR: Intermediate Representation

Hardver	Művelet	Adattípus	Paraméterek
CPU	konvolúció	16 bites lebegőpontos	NCHW vagy NHWC
GPU	MatMul	32 bites lebegőpontos	2D, 3D és 4D tenzorok
FPGA	Normalizálás	8 bites egész	...
...	...	...	

3.1. táblázat. Néhány példa, tényezőnként hányféle esetre kell külön fordítani kernelt könyvtárt

Ezen probléma megoldásához jön képbe a PlaidML. Ez egy *tenzor fordító*[4], mely azt célozza, hogy képes legyen neurális hálózatokat tanítani és futtatni bármilyen típusú hardveren. Más szavakkal segíti a magas szintű keretrendszerek (Keras, ONNX, nGraph) integrálni olyan ezsközökkel, melyekhez nincs meg a szükséges támogatás vagy a meglévő szoftverkészlet hozzájuk szigorúan linceszelt.[10][4]

Az nGraph tehát integrálható a PlaidML-el. Elsősorban az nGraph a platform független IR-rel igyekszik orvolsoni a skálázható backend-el kapcsolatos kihívást. A PlaidML ezt megtámogatja azzal, hogy képes az IR-ből származó gráfokból LLVM, OpenCL, OpenGL, CUDA és Metal kódot generálni melyek a megfelelő hardveren futtathatóak. Így egy magas szintű keretrendszerben írt neurális háló lefordul Intel és AMD processzrokon valamint grafikus processzorokon, az nVidia processzorain, továbbá az Apple cég által feljelsztett eszközökön.

Az nGraph gráf szintű optimalizációját ráadásul kiegészíti automatikusan a PlaidML alacsonyabb szinten, ezzel teljesítmény növekedést érve el.

Összegzésül tehát az nGraph feldarabolja a neurális hálózathoz tartozó számítási gráfot processzor architektúrának megfelelően, majd ezen gráfokat a PlaidML lefordítja a megfelelő kódokra, melyeket aztán a célprocesszorokra lefordítunk és futtatunk.

3.3. Myriad X és az Intel Neural Computer Stick 2

3.4. Google Edge TPU

Az Intelhez hasonlóan a Google is tervezett célprocesszort és hozzá fejlesztő panelt, mély tanulást alkalmazó projektekhez. A Google Edge TPU-ról és a Coral USB Accelerator-ról a Heartbeat internetes magazinban megjelent cikkben értesültem.[7]

Ugyan azt az igényt igyekszik kielégíteni, mint a korábban bemutatott Myriad X processzort és az Intel Neural Compute Stick 2. A valós idejű mély tanuló alkalmazások igénylik az offline módon történő számítást kis fogyasztás mellett. Az Intel-től eltérően a Google ún. Tensor Processing Unit-ot, egy saját fejlesztésű gyorsítókártyát használ a neurális hálózatokkal kapcsolatos

[4]Olyan fordító, melynek nyelve arra lett fejlesztve, hogy főleg tenzorműveleteket igénylő számításokat tudjunk hatékonyan programozni

számítások optimalizálására. Ezeket a társprocesszor kártyákat a Google a saját adatközpontjaiban használja főként és elérhetővé teszi őket a *Google Cloud Platform* nevű szolgáltatásán keresztül. Nevéből adódóan a kártya architektúrája lehetővé teszi, hogy tenzorműveletek tudjanak hatékonyan végrehajtani, utasításkészletük kifejezetten támogatja a neurális hálózatokat. Ilyen művelet a mátrix szorzás, konvolúció és az aktváló függvények alkalmazása az **??** alfejezetben tárgyalt módon.

Az Edge TPU már egy alkalmazásspecifikus integrált áramkör más néven ASIC.

3.5. Új gyorsítók: Intel Nervana Neural Network Processor

Összefoglalás

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Irodalomjegyzék

[1] François Chollet, **Deep Learning with Python**, Manning Publications, 2018.

[2] Altrichter Márta & Horváth Gábor & Pataki Béla & Strausz György & Takács Gábor & Valyon József, **Neurális hálózatok**, Panem, 2006, [Szakkönyv],

[3] McCulloch, W.S. & Pitts, W., **A Logical Calculus of Ideas Immanent in Nervous Activity**, Bulletin of Mathematical Biophysics, 1943, doi:10.1007/BF02478259.

[4] **PlaidML – Home**, 2019. október 20., Vertex.AI.,
`https://vertexai-plaidml.readthedocs-hosted.com/en/stable/`

[5] **Introduction — Documentation for the nGraph Library and Compiler stack**, 2019,
`https://ngraph.nervanasys.com/docs/latest/introduction.html`, [meglátogatva: 2019. október 08.]

[6] Michael A. Nielsen, **Neural Networks and Deep Learning**, Determination Press 2015.
`http://neuralnetworksanddeeplearning.com/` [online, meglátogatva: 2019. október 30.]

[7] **Edge TPU: Hands-On with Google's Coral USB Accelerator**,
`https://heartbeat.fritz.ai/edge-tpu-google-coral-usb-accelerator-cf0d79c7ec56`,
[meglátogatva: 2019. október 23.]

[8] **Keras Dokumentáció**,
`https://keras.io/`, [meglátogatva: 2019. október 23.]

[9] **NervanaSystems/ngraph: nGraph - open source C++ library, compiler and runtime for Deep Learning**, GitHub repository, 2019,
`https://github.com/NervanaSystems/ngraph`,

[10] **plaidml/plaidml: PlaidML is a framework for making deep learning work everywhere**, GitHub repository,
`https://github.com/plaidml/plaidml`, [meglátogatva: 2019. október 20.]

[11] **Constant folding — Wikipedia, The Free Encyclopedia**, 2019, https://en.wikipedia.org/w/index.php?title=Constant_folding&oldid=914455114, [meglátogatva: 2019. október 09.]

[12] **PlaidML — Wikipedia, The Free Encyclopedia**, 2019, https://en.wikipedia.org/w/index.php?title=PlaidML&oldid=898300482, [meglátogatva: 2019. október 5.]

[13] **Rectifier (neural networks) — Wikipedia, The Free Encyclopedia** 2019, https://en.wikipedia.org/w/index.php?title=Rectifier_(neural_networks)&oldid=923288576 [Meglátogatva: 2019. október 29.]

Függelék

F.1. Példa