Szakdolgozat

Palkovics Dénes

Debrecen
2019

Debreceni Egyetem
Informatikai Kar

Hibrid fordítás HPC környezetben

Témavezető:
Dr Kovács László
adjunktus

Készítette:
Palkovics Dénes
mérnökinformatikus BSc

Debrecen, 2019

Tartalomjegyzék

Bevezetés **1**

1. Neurális hálózatok és a Deep Learning **3**
1.1. A neurális hálózatok elmélete . . . 3
1.2. Deep Learning . . . 4
1.3. Függvények, algoritmusok . . . 6
1.3.1. Neuronok aktivációs függvényei . . . 6
1.3.2. Veszteség függvények . . . 7
1.3.3. A hálózat tanító algoritmusai: az optimalizálók . . . 8
1.3.4. A kernel-trükk . . . 10

2. Mély tanuló alkalmazások fejlesztése **11**
2.1. TensorFlow . . . 11
2.2. Keras . . . 11
2.2.1. Neurális hálózat definiálása . . . 11
2.2.2. Hálózat betanítása és inferencia futtatása . . . 12
2.2.3. Hatékonyságvizsgálat . . . 13

3. Speciális platformok megjelenése **15**
3.1. hybrid deep learning . . . 15
3.2. nGraph . . . 15
3.2.1. Motiváció . . . 15
3.2.2. Gráf szintű optimalizálás . . . 16
3.2.3. Skálázható keretrendszer integráció . . . 17
3.2.4. Növekvő kernel szám . . . 17
3.3. Myriad X és az Intel Neural Computer Stick 2 . . . 18
3.3.1. Az NCS2 használata OpenVINO keretrendszerben . . . 19
3.4. Google Edge TPU . . . 24
3.5. Új gyorsítók: Intel Nervana Neural Network Processor . . . 25
3.5.1. NNP-T . . . 25

3.5.2. NNP-I 28

Irodalomjegyzék **30**

Függelék **32**

F.1. Példa 32

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

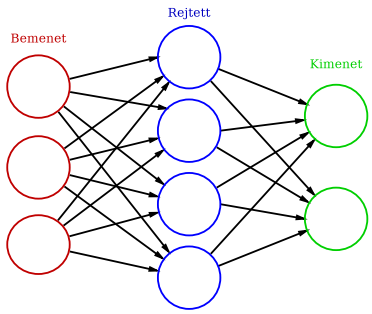
Ebben a fejezetben szeretném összegezni megszerzett tudásomat a neurális hálózatokról és a Deep Learning-ről, magyarul mély tanulásról.

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:

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

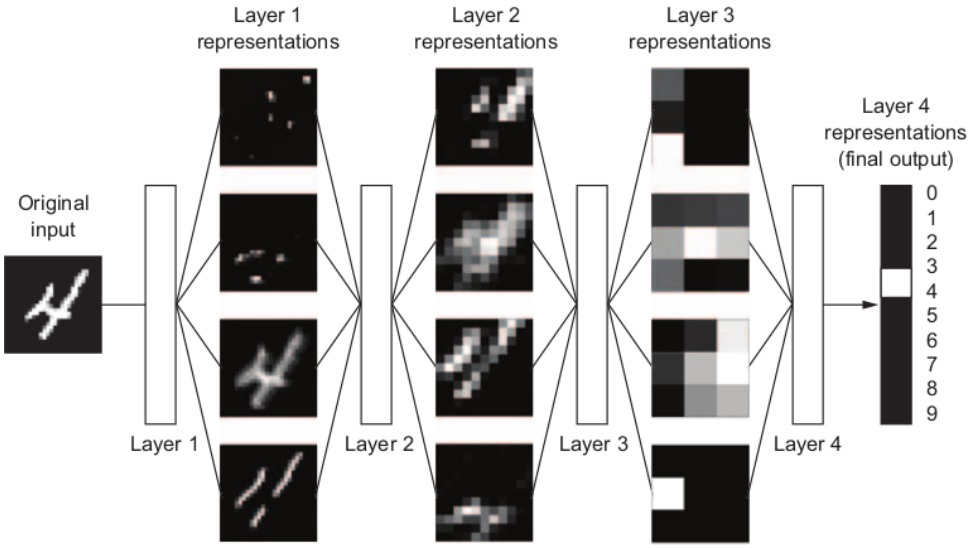
A fentiekből látszik, hogy a hálózat tervezésénél annak négy tulajdonságát kell meghatároznunk: 1. a rétegek és azok neuronjainak számát 2. a neuronok aktivációs függvényét (rétegenként egy típusú függvény az összes neuronra) 3. a szinapszisok súlyát (W) 4. a neuronok aktiválási küszöbét ($\vec{b}$). 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.[20]

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

(a) Szigmoid függvény (b) ReLu függvény

1.3. ábra. Aktivációs függvények

1.3.2. Veszteség függvények

A neurális hálózat tanításához szükséges meghatároznunk egy $c(\vec{y}, \vec{y'})$ veszteségfüggvényt, mely megadja a hálózat kimenetének eltérését az elvárt eredményhez képest adott súlyok mellett. Ezt az eltérést egy skalárértékként képezi le.

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 $(\vec{x}, \vec{y}) \in Z$

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.

Kereszt-entrópia Osztályozási feladatoknál olyan módszert használhatunk a veszteség vagy hiba érték számításához, melynél az $\vec{y}$ és $\vec{y'}$ kereszt entrópiáját határozzuk meg. Ilyen jellegű feladatoknál az említett vektorok valószínűségi eloszlások.

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

A mély tanuláshoz használt keretrendszereknél gyakran másként implementálják a kereszt-entrópiát kétosztályos és többosztályos esetre. Ezekre az implementációkra *bináris kereszt-entrópia* és *kategorikus kereszt-entrópia* néven hivatkoznak.

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. Neurális hálózatoknál gradienscsökkentésen alapuló technikákat alkalmazunk. A módszer alapját az az elképzelés adja, hogy a meghatározunk egy $C(\vec{w}_1, b_1, \ldots, \vec{w}_L, b_L)$ függvényt, mely a hálózat összes paramétere alapján meghatároz egy E hibaértéket, ami a tanítási folyamat egy ciklusa során kapott veszteségértékek átlaga. Ezen függvény képe egy úgymond *hibafelület*, melynek megkeressük minimum helyeit tanítás során. Gyakorlatban nem kivitelezhető a globális minimum meghatározása, és lokális minimumok keresését is iteratív módszerekkel célszerű végezni. Többváltozós függvények esetén a gradiens vektor ellentettje ($-\nabla f$) segítségével meghatározható, hogy adott pontban – azaz megadott függvényparaméterek esetén – a paramétereket milyen mértékben változtassuk, hogy a függvény kimeneti értékét a legdrasztikusabb mértékben csökkentsük.

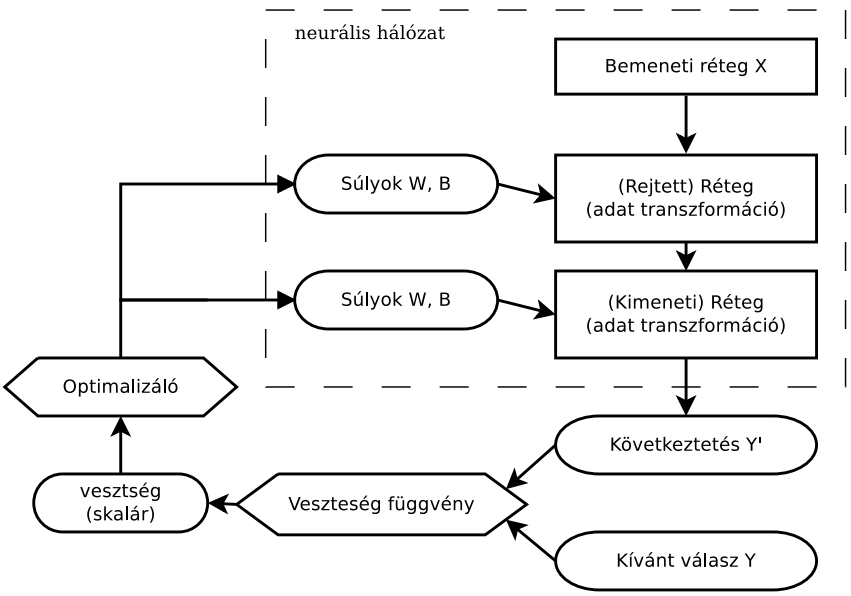
Belátható, hogy C ekvivalens a c-k átlagával.

$$\frac{1}{n} \sum_{i=1}^{n} c(y_i, y'_i) \equiv C(\vec{w}_1, b_1, \ldots, \vec{w}_L, b_L)$$

A gradienscsökkentés naiv megközelítése pedig a következő:

$$\vec{W}_i = (\vec{w}_{i1}, b_{i1}, \dots \vec{w}_{iL}, b_{iL})$$
$$\vec{W_{i+1}} = \vec{W}_i - \nabla C(\vec{W}_i)$$

Ahol $\vec{W}_i$ a tanítás i-edik ciklusában az L rétegű hálózat összes paraméterének vektora.



1.4. ábra. A neurális hálózat tanításának folyamata

A tanítási folyamat iterációit *eposzoknak* (angolul: epoch) nevezi a szakirodalom. Minden eposzban az optimalizáló hatására a kimeneti hiba csökken, a hálózat *következtetése* egyre közelebb kerül az elvárthoz.

Sztochasztikus gradienscsökkentés A naiv megközelítést alapul véve meghatározunk egy η tanítás sebességet (angolul: learning rate), mellyel azt befolyásoljuk, hogy egy eposzban a hálózat paramétereit mekkora mértékben változtatjuk. A tanítási sebesség az optimalizálók egy másik paramétere, tehát más eljárás esetén is alkalmazzuk valamilyen formában.

$$\vec{W_{i+1}} = \vec{W}_i - \eta \nabla C(\vec{W}_i)$$

Ezen paraméter változtatásánál fontolóra kell vennünk két tényezőt. Nagy tanítási sebesség gyorsabb gradiens csökkenést eredményez, ugyanakkor az elmozdulás a hibafelületen nagy kitérésekkel történik a minimumpont irányába. Túl kicsi tanítási sebességgel – azon túl, hogy megnövekszik az eposzok szám – ugyanakkor az optimalizálás "beragadhat" egy kedvezőtlen lokális minimum helyen.

négzetes közép visszaterjesztése Konstans tanulási rátát nehezen lehet jól megválasztani, ezért olyan eljárásokat dolgoztak ki, ahol ezen érték adaptálódik a tanítási folyamat során, hogy kiküszöbölje az előző bekezdésben említett nagy kitéréssel történő konvergenciát és a "beragadást". Az egyik ilyen módszer az, hogy a tanulási rátát elosztjuk az előző ciklusokban számított gradiensek nagyságának négyzetes közepével.

$$v(w, y) = \gamma v(w, y-1) + (1-\gamma)(\nabla C(W_i))^2$$

Ahol γ az ún. felejtési tényező amivel szabályozhatjuk, hogy mekkora befolyása legyen az újabb gradienseknek.
A hálózat paramétereit a következőképpen frissítjük:

$$W_{i+1} = W_i - \frac{\eta}{\sqrt{v(w,y)}}\nabla C(W_i)$$

Tanítás során a veszteségfüggvény sztochasztikusan konvergál a nullához és ezzel együtt a hálózat egyre pontosabban következtet a tanító készlet adataiból. A tapasztalat azonban az, hogy ez nem feltétlenül igaz más, a tanítás során nem használt adatokra, sőt elérkezhet a hálózat egy olyan állapotba, amikor a használat közben rosszabbul teljesít, mint a tanítás végén. Ilyenkor a hálózat specializálódik a tanító adathalmazra végső esetben akár az egyes adatpontokra a hipotézis téren. Ezt a jelenséget, amikor a hálózat jobban következtet a tanítás során használt adathalmazon *túlilleszkedésnek* nevezzük és *alul-illeszkedésnek* azt, mikor a hálózat nem generalizál elég jól, más szóval túl kevés tanítási cikluson esett még át. Hogy a hálózat a hipotézis térből a megfelelő függvények – nem törekszünk az adatok egy tökéletes leképezésére – valamelyikét modellezze, szükséges a tanítást felügyelni, hogy az ne illeszkedjen se túl, se alul.

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”[9]. 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][9].

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. kód. 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 inferencia 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. kód. Hálózat betanítása

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

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

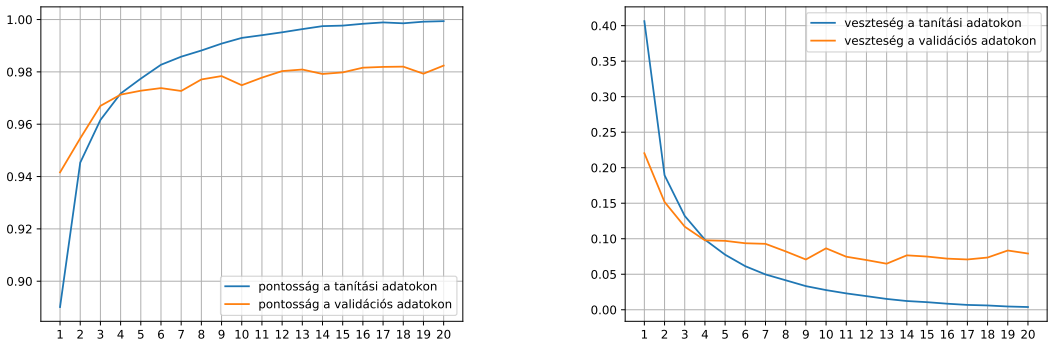
A `train_datas` és `train_labels` a tanítókészletet alkotó minták és azok címkéi, a várt kimeneti értékek. Ezek elemszáma kötelezően meg kell egyezzen. A *batch_size* paraméterrel, ami az egy *epoch*-ban egyszerre feldolgozandó adatokat jelenti. A betanítás végén a `network` egy *betanított*, a célfeladat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

2.3. kód. inferencia

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

2.2.3. Hatékonyságvizsgálat

A tanítás felügyeletére a tanításhoz rendelkezésre álló adatok egy részét külön kell választanunk a tanítókészlettől. Ez a készlet lesz a validálási készletünk, mellyel minden tanítási ciklus végén leellenőrizzük, hogy hogyan teljesít a hálózat olyan adaton, amit még „nem látott”. A validálást Keras-ban megtehetjük a tanítással egy lépésben, a `fit()` algoritmussal, ha a `validation_data` opcionális paraméterének megadunk egy `(data, label)` `tuple` típusú változót. Ekkor a metódus olyan referenciával tér vissza (2.2.2 kódrészletben ez a `history`), melyen keresztül a tanítási és validálás statisztikák is beolvashatóak. A 2.2.2 szakaszban említett metrikák a validálási folyamathoz is rögzítve lesznek.



2.1. ábra. Példa kimutatás a Keras által gyűjtött statisztikákról

3. fejezet

Speciális platformok megjelenése

3.1. hybrid deep learning

3.2. nGraph

A most leírtak alapját az nGraph hivatalos dokumentációja adja[6]. 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 keretrend-

[1]Programkönyvtár, mely neurális hálókkal kapcsolatos elemi, *mag* függvényeket tartalmazzák

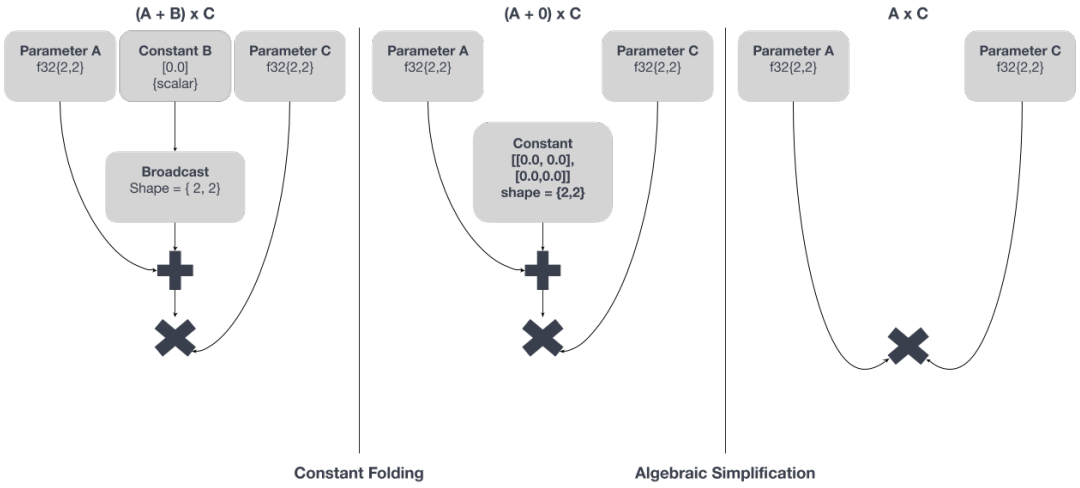
szer alatt az Nvidia CuDNN könyvtárát használjuk. A kernel könyvtárak egy adott célarchitektúrára optimalizált kernelekből és egyéb műveleti szintű optimalizálásokból állnak, ezekkel érve el teljesítménynövekedést. Azonban a kernel könyvtáraknak három fő problémája van:

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]

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).

[2] forrás: [6]

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 ($k = h \times t \times m \times p$).

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

[3] IR: Intermediate Representation

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

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

szavakkal segíti a magas szintű keretrendszerek (Keras, ONNX, nGraph) integrálni olyan ezközökkel, melyekhez nincs meg a szükséges támogatás vagy a meglévő szoftverkészlet hozzájuk szigorúan linceszelt.[17][5]

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.

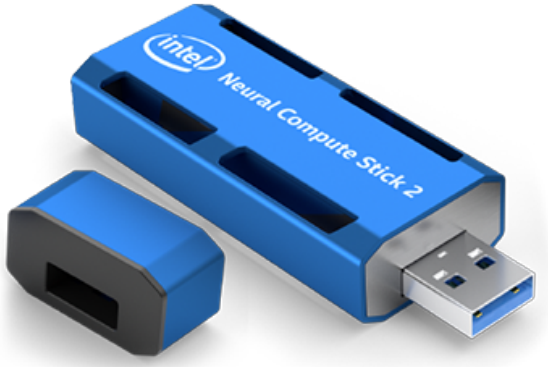
3.3. Myriad X és az Intel Neural Computer Stick 2

Ma az iparban mély tanulással működő modern alkalmazások nagy méretű és teljesítményű számítógépeket igényelnek. Erre az igényre válaszul felhő szolgáltatást nyújtó vállalatok külön platformot építettek ki. Az ilyen kliens-szerver architektúrájú gépi tanulás hatékony lehet, azonban a valós-idejű alkalmazásoknál és hordozható eszközökben történő felhasználásnál komoly hátrányt jelenthet a hálózati késleltetés. Hogy ilyen területeken is alkalmazható legyen a mély tanulás, hardvergyártók olyan kis energiafogyasztású *alkalmazás-specifikus integrált áramkörök* fejlesztésére törekedtek, melyek képesek neurális hálózatokat hatékonyan futtatni. Ilyen az Intel által kifejlesztett Myriad X lapka is, melyet főleg *gépi látáshoz* terveztek.

Maga a Myriad egy Videójel feldolgozó processzor széria, melynek 3. generációja a Myriad X architektúrája kiegészült egy úgynevezett *Neural Compute Engine-el*, mely a betanított hálózatok futtatását hivatott gyorsítani. Ezzel együtt a lapka 16 darab 128-bit VLIW processzor-

magok, ún. SHAVE magot és 2,5MB 400GB/s sávszélességű memóriát tartalmaz. Számításokat 8bites egészek és 16bites lebegőpontos váltózókon tud végezni.[5]

A Myriad X kifejlesztésén túlmenően a vállalat tervezett egy gyorsítókártyát is ehhez a processzorhoz, melyet Neural Compute Stick (NCS) névvel forgalmaz. Az Intel kiadta a második verzióját az eszköznek Neural Compute Stick 2 (NCS2) névvel és az első verziót nem forgalmazz már. Ezért a továbbiakban a leírtakat az NCS2-re kell érteni. Az eszköz USB 3.0 -ás interfésszel rendelkezik, így sok gazdagéppel használható, többek között Raspberry Pi-vel is. Meghajtására az Intel OpenVINO eszközkészlete használható, melyet gépi látáson alapuló alkalmazások fejlesztésére készített a vállalat. Lehetőség van egy platformon több NCS2 modul együttes használatára, így egyszerűen növelhető a rendszer teljesítménye.



forrás:https://software.intel.com/en-us/neural-compute-stick

3.2. ábra. Neural Compute Stick 2 gyorsító

3.3.1. Az NCS2 használata OpenVINO keretrendszerben

Az alábbiak alapját az Intel OpenVINO eszközkészletének dokumentációja adja.[7] A dokumentációban a felépített neurális hálózatokra modell néven hivatkoznak, amit most én is átveszek.

Az OpenVINO eszközkészlet alkalmas arra, hogy gyorsan készítsünk olyan alkalmazásokat, melyek valamilyen módon emulálják az emberi látást. Képesek vagyunk segítségével az NCS2-n kívül más Intel gyártmányú hardverrel dolgozni, például Intel CPU-n és integrált GPU-n. Továbbá népszerű keretrendszereket támogat, mint a TensorFlow, Caffe, MXNet vagy az ONNX. Az eszközkészlet komponensekei közül az alábbiakat emelném ki:

Inference Engine Python API, aminek segítségével különböző gyorsítókártyák megszólíthatók egységes módon.

[5]forrás: https://www.movidius.com/myriadx

Model Optimizer parancssoros alkalmazás modelleket definiáló állományok átkonvertálására. Ahhoz, hogy a különböző keretrendszerekben megalkotott modelleket egységesen tudja kezelni az Inference Engine, szükség van azok átkonvertálására egy platformfüggetlen struktúrába.

OpenCV az OpenCV közösségi fejlesztésű verziójának Intel hardverekre fordított változata.

Az OpenVINO-val való munka folyamatának három fő lépése van: Betanított Neurális hálózat beszerzése vagy implementálása; a hálózat átkonvertálása a Model Optimizer-rel; Inference Engine-re épülő program implementálása, mely beolvassa az átkonvertált állományokat és átadja a neurális hálózat modelljét a célhardvernek;

Modell átkonvertálása

Mint már korábban említettem a platform különböző neurális hálózatok implementálására kifejlesztett keretrendszereket támogat. Ezek a keretrendszerek képesek a definiált és betanított hálózatokat valamilyen állomány formátumban elmenteni. Hogy ezeket az állományokat minden gyorsító eszközön egységesen legyen képes kezelni az *Inference Engine* szükséges az állományokat átkonvertálni a *Model Optimizer* segítségével. Ez egy Python-ban írt kereszt-platformos parancssoros alkalmazás, mely a beolvasott modell állományokban tárolt hálózatot elmenteni egy *közbenső reprezentációvá*[6], melyre a továbbiakban *IR-ként* fogok hivatkozni. Az átkonvertálás során a Model Optimizer analizálja a modellt[7] és olyan korrekciókat hajt végre rajta, hogy optimális módon hajtódjon végre az egyes gyorsítókon.

Egy TensorFlow-ban implementált neurális hálózat átkonvertáláshoz az alábbi lépéseket kell végrehajtani:

Model Optimizer konfigurációja Az alkalmazás előre elkészített szkriptekkel települ a különböző keretrendszerekhez történő konfigurációhoz. A szkriptek a `<fÅŚkÃűnyvtÃąr>/deployment_tools/model_optimizer/install_prerequisites` könyvtárban találhatóak. TensorFlow-val való munkához az `install_prerequisites_tf.sh` – Windows operációs rendszeren ugyanezen fájlnéven `.bat` kiterjesztéssel – konfigurációs szkriptet kell lefuttatni.

TensorFlow modell befagyasztása Ez egy keretrendszer specifikus lépés. Amikor Pythonban definiáljuk a hálózatot, rendszerint elmentjük azt egy *inference graph* fájlban. Ez általában olyan módon történik, hogy a hálózat tanítható marad, tehát annak paraméterei, mint változók tárolva vannak az állományban. A Model Optimizer csak úgy tud dolgozni ezen az állományon,

[6] Intermediate Representation (IR)

[7] betanított, kvázi használatra kész neurális hálózat

ha az úgymond be van *fagyasztva*. A keretrendszerben befagyasztott modell kimentésének műveletét a következő kódrészlet mutatja:

```
1  import tensorflow as tf
2  from tensorflow.python.framework import graph_io
3
4  frozen = tf.graph_util.convert_variables_to_constants(sess, ↩
       sess.graph_def, ["output_node"])
5  graph_io.write_graph(frozen, './', 'model.pb', as_text=False)
```

ahol:

`sess` egy TensorFlow `Session` példánynak a referenciaváltozója

`["output\_node"]` a kimeneti csomópontok nevének listája. Egy befagyasztott hálózat csak azokat a csomópontokat fogja tartalmazni a az eredeti `sess.graph_def`;

`'model.pb'` a létrehozandó modell fájl neve;

`as\_text` boolean típusú változóval megadható, hogy a `write_graph` metódus a bináris vagy karakteres formátumban írja ki a gráfot.

Modell átkonvertálása Ennél a pontnál használjuk a Model Optimizer-t, azaz ekkor történik meg a konverzió. Az alkalmazás a `<fÅŚkÃűnyvtÃąr>/deployment_tools/model_optimizer` könyvtárba települ. Hogy a `model.pb` állományunkat átkonvertáljuk a következő parancsot kell kiadni:

```
python3 mo_tf.py --input_model model.pb
```

A kimenet maga az IR, két állomány: egy xml kiterjesztésű fájl, amiben a hálózat topológiája van definiálva és egy bináris fájl, ami a hálózat paramétereit – súlyait és küszöbértékeit – tartalmazza. A program további argumentumaival specializálhatjuk a konvertálás folyamatát.

Inference Engine

Az OpenVINO eszközkészlet tartalmaz egy futtatókörnyezetet az *Inference Engine-t*, egy programkönyvtár csomagot melynek központi része a `libinference_engine.so`[8] könyvtár. A könyvtárcsomag C++ nyelven íródott, és programozói interfésze ehhez, továbbá Python nyelvhez készült. Az utóbbihoz készült API szerényebb funkcionalitással bír. Feladati az IR-ben tárolt modell beolvasása, az inferencia optimalizálása a célhardveren történő végrehajtáshoz és annak futtatása. A hardvercsoportokkal történő kommunikációs eljárások külön programkönyvtárakba – ahogy a dokumentáció hivatkozik rájuk *pluginokba* – lettek szervezve.

[8] Windows operációs rendszeren természetesen .dll kiterjesztésű és a fájlnév nem tartalmazza a "lib" karaktersort és ez igaz a többi programkönyvtár elnevezésére is

GPU plugin (libclDNNPlugin.so)	Intel HD Graphics és Intel Iris Graphics
CPU plugin (libMKLDNNPlugin.so)	Intel Xeon AVX2 és AVX512 utasításkészlettel, Intel Core AVX2 utasításkészlettel, Intel Atom SSE utasítással
FPGA plugin (libdliaPlugin.so)	Intel Vision Accelerator eszköz Intel Arria 10 FPGA-val (Speed Grade 1 és 2), Intel Programmable Acceleration kártya Intel Arria 10 GX FPGA-val
VPU plugin (libmyriadPlugin.so)	NCS és NCS2, Intel Vision Accelerator eszközök Myraid processzorral
GNA plugin (libGNAPlugin.so)	Intel Speech Enabling Developer Kit, Amazon Alexa Premium Far-Field Developer Kit, Intel Pentium Silver J5005, Intel Celeron J4005, Intel Core i3-8121U
Multi-Device plugin (libMultiDevicePlugin.so)	egy modell több eszközön történő parallel inferenciájára
Heterogenous plugin (libHeteroPlugin.so)	modell automatikus szétosztása több eszközre (akkor szükséges, ha néhány hálózati réteg típust nem minden eszköz támogat)

3.2. táblázat. Plugin típusok és támogatott hardverek

Az Inference Engine használatával Python nyelven ismerkedtem meg, így alább ezen nyelvi API használatát mutatom be. Az interfészt a következő osztályok szolgálatatják:

`ExecutableNetwork` osztály reprezentál egy modell példányt, amit betöltöttük a pluginba és készen áll az inferenciára;

`IECore` egy Inference Engine objektum. Segítségével egy egységes interfészen keresztül kezelhetjük a pluginokat;

`IENetLayer` tartalmazza a főbb információkat a modell rétegeiről, továbbá eszköz a rétegek néhány paraméterének módosításához;

`IENetwork` minden információt tartalmaz az IR fájlokból beolvasott modellről, és annak paramétereinek módosítását teszi lehetővé;

`IEPlugin` a plugin-ok fő interfésze. Célja, hogy inicializálja és bekonfigurálja a plugin-t;

`InferRequest` egy interfészt kínál az inferencia futtatásának kérésére az `ExecutableNetwork` osztályon keresztül, továbbá kezeli az inferencia végrehajtását és a kapott adatokat;

`InputInfo` a modell bemeneti rétegének tulajdonságait tartalmazza: `layout` karaktersorozat, a (bemeneti) tenzor elrendezése[9], `shape` szám n-es a tenzor alakja, `precision` az adatok számábrázolás szerinti típusa;

`LayersStatsMap` a Python `dict` osztályának leszármaztatása, mely újraimplementálja a `update()` metódust, hogy módosítható legyen a rétegek finomhangolásának statisztikái;

`LayerStats` a rétegek finomhangolási statisztikáinak konténere;

`OutputInfo` a modell bemeneti rétegének tulajdonságait tartalmazza hsonlóan az `InputInfo` osztályhoz;

Egy elemi alkalmazás implementációjához, mely inferenciát végez az NCS2 eszközön szükség van az `IENetwork` és `IECore` vagy `IEPlugin` osztályokra továbbá létrejön egy `ExecutableNetwork` egyed, amivel az inferenciát elvégeztetjük. A modell feltöltése az eszközre, majd az inferencia futtatása a 3.3.1 kód szerint tehető meg.

3.1. kód. Inference Engine használata PYthon-ból

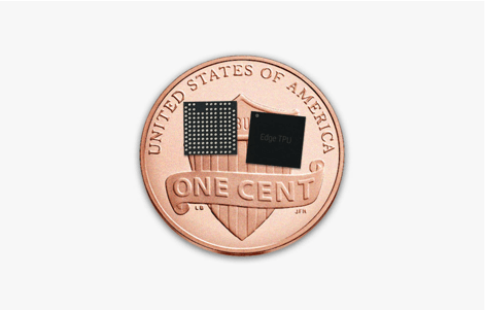
```
1 from openvino.inference_engine import IENetwork, IEPlugin
2 net = IENetwork(model='<dir>/model.xml', weights='<dir>/model.bin')
3 plugin = IEPlugin(device='MYRIAD')
4 exec_net = plugin.load(network=net)
5 i_layer_key = list(net.inputs.keys())[0]
6
7 if net.inputs[i_layer_key].shape == list(samples.shape) :
8   res = exec.net.infer(inputs={i_layer_key : samples})
```

Az `IEPlugin` rendelkezik egy `load()` metódussal, mely egy `IENetwork` objektumtól megkapja a hálózat adatait és példányosít egy `ExecutableNetwork` egyedet. Ezen objektum `infer()` metódusával végeztethetjük az inferenciát szinkron módon. Lehetőség van aszinkron üzemmódra az `async_infer()` metódussal. Mindkét metódus használata esetén át kell adni egy táblázat típusú paramétert (Python `dict` osztály) melynek kulcsértékei a bemeneti rétegek nevei (ez egyezik az IENetwork inputs attribútumának kulcsával), a kulcsokhoz rendelt értékek pedig a minták megfelelő alakú, `numpy.ndarray` típusú tenzora. A `res` változó szintén egy táblázat lesz, amiben a kimeneti rétegek neveihez, mint kulcsokhoz van rendelve az inferencia eredménye.

[9]Elrendezés: a tenzorként reprezentált minták tömbjében az adatok milyen elrendezésben szerepelnek. Például egy képfeldolgozáshoz használt hálózat bemeneti rétege esetén az 'NHWC' elrendezés azt jelenti, hogy a beolvasott tömb 1. tengelye mentén a mintákon(sample Number) való bejárást, 2. és 3. tengelye mentén a magassági(Height) és szélességi(Width), negyedik tengelye mentén a színcsatornák(Channel) szerinti bejárást végezhetünk

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.[8] A Google TPU-ról szerzett ismereteim a wikipédia „Tensor Processing unit” cikkéből szereztem.[21]



forrás: https://cloud.google.com/edge-tpu/

3.3. ábra. Edge TPU

3.4. ábra. Az Edge TPU-t építették az USB interfésszel ellátott Coral USB Accerlerator-ba.[8]

A Google a Tensor Processing Unit™-ot, egy saját fejlesztésű alkalmazásspecifikus integrált áramkört használ a neurális hálózatokkal kapcsolatos számítások optimalizálására. Ezek lényegében mátrix műveletekre számítására fejlesztett processzorok CISC utasításkészlettel. Három generációt ért meg, az első még csak 8-bites egészeken tudott műveleteket végezni, azonban már a második generáció képes volt lebegőpontos számításokra is. A Tensor processing unit-okkal felszerelt gyorsítókártyákat a Google 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 konvolúció és az 1.1 alfejezetben tárgyalt aktváló függvények alkalmazása a mátrix szorzáson túl.

Az Edge TPU a cég szerverei által használt TPU-hoz viszonyítva kisebb méretben és energiafogyasztásban. Ugyan azt az igényt igyekszik kielégíteni, mint a korábban bemutatott Myriad X processzor. Azzal összehasonlítva viszont képes neurális hálózatok tanítására is korlátozott

mértékben. Az Edge TPU részét képezi a Google Coral eszközkészletének, mellyel alternatívát nyújtanak a felhőalapú Cloud AI szolgáltatással szemben. A Coral különböző fajtájú hardvert nyújt a felhasználóknak, mindegyik központi eleme az Edge TPU. Az eszköz programozásához használható API-t az Edge TPU Python könyvtár (`edgetpu` Python modul) valósítja meg, amely képes TensorFlow Lite-ban alkotott betanított neurális hálózatok futtatására. A könyvtár kulcsfontosságú interfészeit a következő osztályok képezik.Képosztályozási feladatokhoz a `ClassificationEngine` használatos, vizuális objektumazonosításra a `DetectionEngine`. Az `ImprintingEngine` és `SoftmaxRegression` az ún. *transzfer tanuláshoz* alkalmazható, vagyis előre betanított hálózatok a feladatnak megfelelő újratanítását végezhetjük el vele.

3.5. Új gyorsítók: Intel Nervana Neural Network Processor

Az Intel idén augusztusban mutatta be a Hot Chips konferencián kifejezetten mély tanulásra Spring Crest és Spring Hill kódnevek alatt fejlesztett rendszerchipeket, a Neural Network Processor for Training-et (NNP-T) és a Neural Network Processor for Inference-et (NNP-I). Mint nevük is mutatja ezek kifejezetten mély tanulás gyorsításra lettek fejlesztve és úgy tervezték őket, hogy jól skálázható rendszert alkossanak. Ezeket a hardvereket a konferencián elmondottak alapján mutatnám be.[13][15] Ebben az alfejezetben szereplő ábrák a konferencián levetített fóliákból származnak.[12][14]

3.5.1. Neural Network Processor for Training

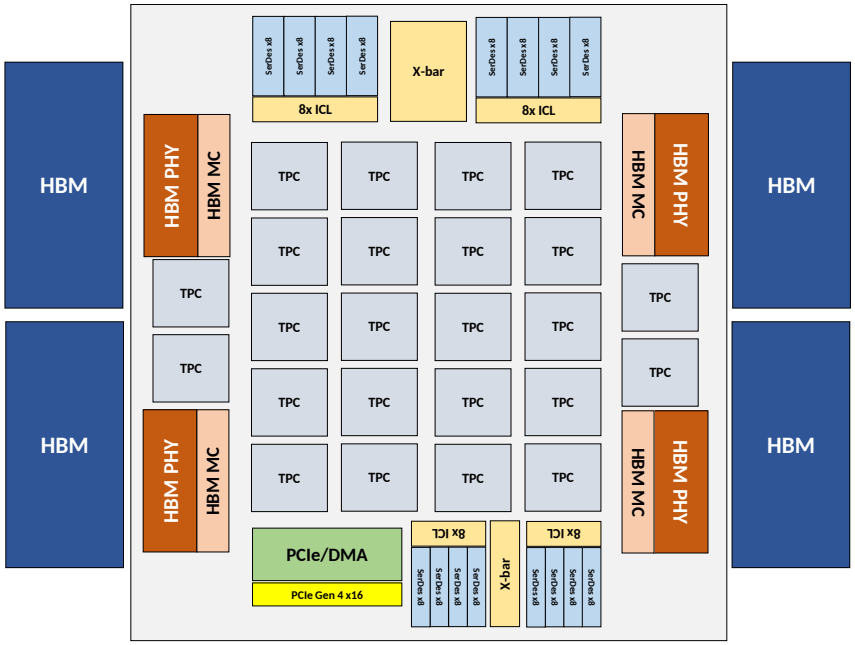
Az NNP-T egy egylapkás rendszer vagy rendszerchip, melynek fejlesztésénél fontos szempont volt az egyensúly a számítási teljesítmény, a perifériák közötti kommunikáció sebessége és memória használat között. Ezt támogatja a lapka memóriájába már betöltött adatok minél hatékonyabb újrafelhasználásának elősegítése, továbbá a kötegelt adatfeldolgozás optimalizálása. Könnyen skálázható rendszert lehet építeni belőle, ugyanis ezt beépített megoldások segítik. A mély tanulás területén megfigyelhető trendek figyelembevételével tervezték, hogy a jövőbeli munkafolyamatokat is támogassa.

Hardver tulajdonságai

Rendszerchip A chip 4. generációs 16 sávos PCIe végponttal rendelkezik. A nagy memóriakapacitás lehetőségét lapkán 4 darab HBM2[10] memória foglalattal oldották meg. A lapkák közötti kommunikációra SerDes interfész 64 sávval is implementálva lett. A számításokat

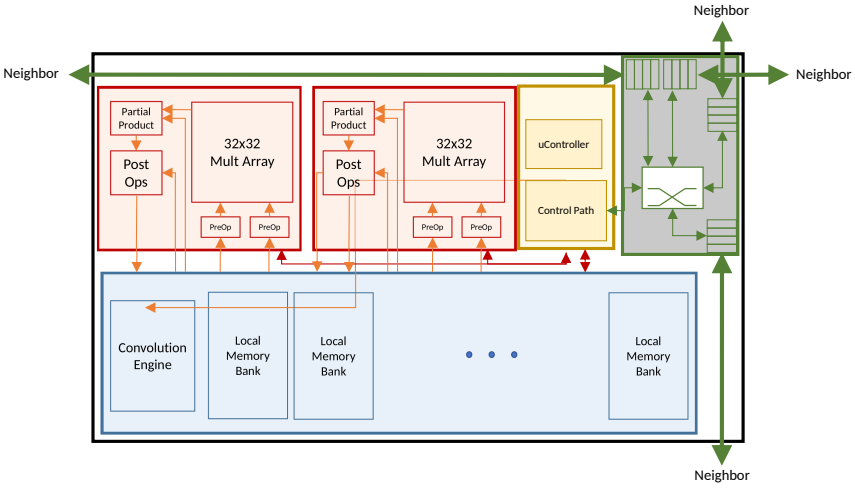
[10]High Bandwidth Memory: DRAM modulok térben egymásra pakolását alkalmazó RAM interfész típus. HBM2 esetén a legnagyobb kapacitás 8GB lehet egy tokban. Leginkább videó kártyákban használatos

24 darab tenzor processzormag végzi(TPC), és a vállalat szerint számítási kapacitásuk összesen 119 TOP/s (billió utasítás másodpercenként).A lapkán elhelyeztek 60 MB osztott memóriát is.



3.5. ábra. Az NNP-T egylapkás rendszer (SoC) felépítése

Tenzor processzor (TPC) Egy TPC magon belül két darab mátrixszorzó tömb van, melyek egyenként 32×32-es méretű mátrixok tud szorzást végezni, a konvolúció műveletet pedig külön egység végzi. Minden mag továbbá rendelkezik 2,5 MB Cache memóriával, melyek sávszélessége 1,4 TB/s. A 3.6 ábrán látható zöld színnel jelzett útválasztó a magok közötti kommunikációért felelős.

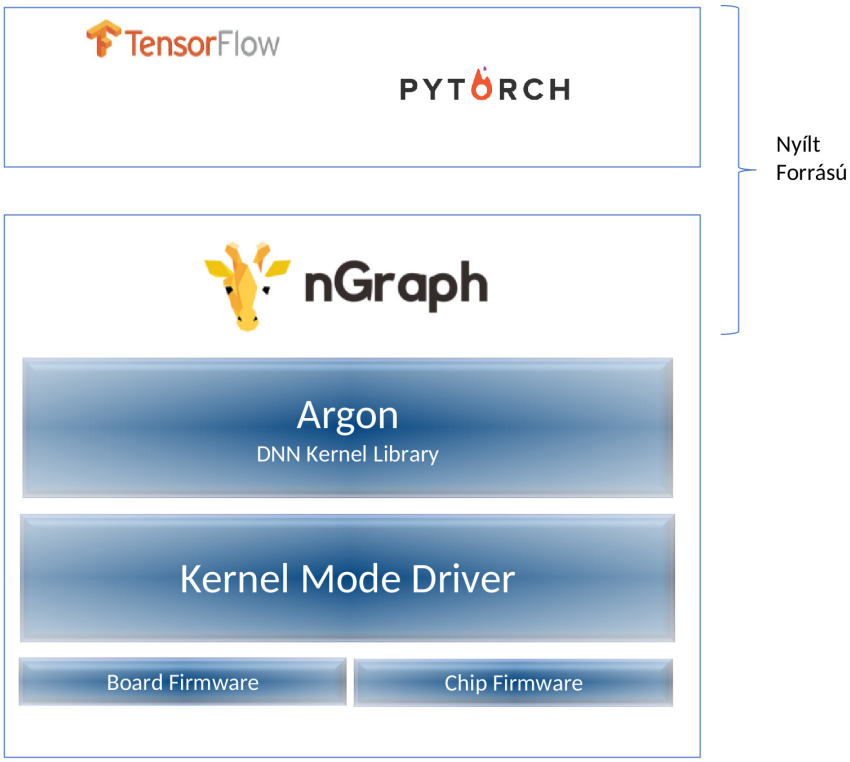


3.6. ábra. Egy TPC felépítése

hardverkártya tulajdonságai Az Intel az NNP-T rendszerchipet egyedi PCIe és OAM alaktényezőjű kártyán fogja forgalmazni, melyre egy rendszerchipet szerelnek összesen 32 GB HBM2-2400 típusú memóriával – melynek elméleti sávszélessége 1,22 TB/s – BGA tokozásban. A rendszerchip órajele az 1,1GHz-et is elérheti. A kártya 4. generációs PCI Express x16, valamint SPI, I2C csatolófelülettel és általános célú I/O kivezetéssel rendelkezik. A központi egység léghűtésű átlagos használat esetén 150 és 250 W közötti fogyasztással. A kártyán kivezetésre került 64 SerDes sáv, melynek aggregált sávszélessége 3,58 Tb/s. Ezen interfész segít megvalósítani több lapka összehangolt működését, mellyel maximum 1024 csomópontot lehet összekötni úgy, hogy azok összes TPC magja egy nagy hálózatként kezelhető, ezzel elérve a masszív párhuzamosítást.

Az NNP-T programozásához használatos szoftver együttes

Az alkalmazásfejlesztés az ilyen rendszerchipekkel szerelt kártyákra egyedi eszközkészlettel lehetséges. A vállalat ajánl egy teljes fejlesztői szoftverkészletet nyílt forrású komponensekkel. A 3.7 ábra az egyes komponenseket hardver és felhasználó közötti viszonya szerint rétegezve mutatja be. Ezek közül az nGraph-ot és a magas szintű keretrendszerek közül a TensorFlow-t, már bemutattam korábban, illetve pár szóban összefoglalom az Argon programkönyvtárat.



3.7. ábra. A fejlesztésnél használható szoftveregyüttes

NNP-T Programozási modell

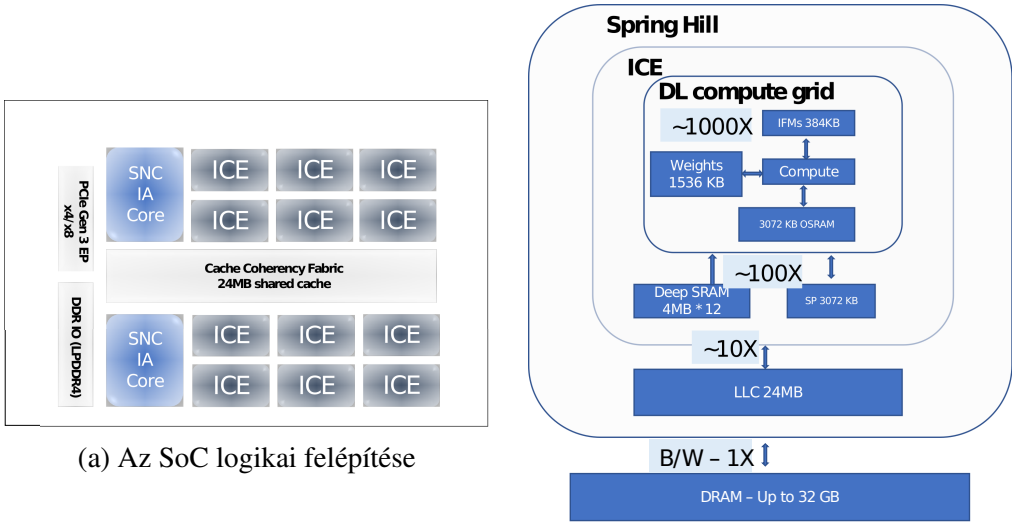
Az Utasításkészlet szintjén is neurális hálózatokra lett optimalizálva a rendszerchip azzal, hogy a hálózat tanításakor alkalmazandó tenzor műveletekhez natív utasítások tartoznak, úgy mint transzponálás, tenzor darabolása, stb. Az utasításkészlet is ezen utasításokra van korlátozva, azonban ez bővíthető egyedi mikrokontroller kutasításokkal. A bővíthetőség azt a célt szolgálja, hogy a jövőbeli mély tanulásra épülő munkafolyamatok támogatottsága is megoldható legyen utasítás szinten. Az Intel hierarchikus elosztott programozási modellt használ, ami jól skálázható, így alkalmazható egy rendszerchip TPC magjaira, vagy akár több lapka együttes programozáskor. Ezt segíti, hogy a magok közti kommunikáció és szinkronizáció elemi utasításai konzisztensek lapkán belül és több lapka között. Azon túl, hogy a hardver az adatokat tenzoronként kezeli, a tenzorokon belüli elemi változókat Bfloat16[11] és 32 bites lebegőpontos típusként tudja értelmezni

3.5.2. Neural Network Processor for Inference

Betanított neurális hálózatok futtatására alkalmazható rendszerchip, melyet az NNP-T-vel együtt kezdett el fejleszteni a vállalat. Ennél a lapkánál a cél a kis fogyasztás volt. A gyártó szerint Wattonkénti 4,8 billió számítást képes végezni másodpercenként. A számítási teljesítmény a fogyasztás hangolásával természetesen változó. (Jelenleg M.2 és PCIe alakényezőjű kártyákon forgalmazzák. Előbbi 12 wattos teljesítményű, míg az utóbbi teljesítménye 75 W. Számítási kapacitásuk 50 és 170 TOP/s)

Hasonlóan az NNP-T-hez ez a rendszerchip is egyedi processzormagokat tartalmaz, összesen 12-t. A mag megnevezése *Inference Compute Engine*, röviden ICE. Az lapka (3.8a ábra) kiegészül még 24 MB cache memóriával, melyen az összes mag osztozik. Az ICE magok mellett két Intel IA Core processzormag is részét képezi a rendszerchipnek. Ezek párban az ICE-k munkáját hangolják össze a fordítótól érkező utasítások alapján, – ide kapcsolódik, hogy *Ahead Of Time* fordítás végezhető az eszközre – tehát a programozó által közvetlenül ez a két processzormag elérhető. Egy ICE két számítást végző blokkból áll: egy *Deep Learning compute grid*-nek nevezett több számítási egységből álló hálózatból, – melyek a neurális hálózatok modellezése során igényelt műveletek javát végzik – illetve egy programozható vektor processzorból AVX és VNNI utasításkészlettel. A vektor processzor jóvoltából tetszőleges utasításokkal bővíthető a mag.

[11] Az IEEE 754 félszeres, azaz 16 bites pontosságú lebegőpontos típustól annyiban tér el, hogy az exponens 5 helyett 8 bites. Ez az Intel állítása szerint azon túl, hogy memória hely takarékos, tanításkor jól konvergáló változókat biztosít



(a) Az SoC logikai felépítése

(b) Memória menedzsment

3.8. ábra

Egy ICE 4 MB lokális SRAM memóriát kapott. Ezen ICE magok képesek 16 bites lebegőpontos valamint 8, 4, 2 és 1 bites egész változókat kezelni. Az NNP-I lapka 3. generációs PCIe (x4 vagy x8) végponttal kapcsolódik a befogadó kártyához, továbbá egy LP-DDR4 memória ültethető a hordozó lapkára az S . Azért, hogy az adatmozgatást minimalizálják a memória allokáció 4 szinten van rendezve hierarchikusan, ahogy az 3.8b ábrán látható. Ezt a rendszerchipet ugyanazzal a szoftver együttessel tudjuk programozni, mint az NNP-T SoC-t.

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

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

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

[7] **OpenVINO™ toolkit Documentation**, 2019, `docs.openvinotoolkit.org/2019_R3/index.html` [meglátogatva: 2019. november 4.]

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

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

[10] **Intel® Nervana™ Neural Network Processors**, `www.intel.ai/nervana-nnp/` [meglátogatva: 2019. november 07.]

[11] **At Hot Chips, Intel Pushes 'AI Everywhere'**, 2019, `newsroom.intel.com/news/hot-chips-2019/` [meglátogatva: 2019. November 07.]

[12] Andrew Yang, **Deep Learning Training At Scale Spring Crest Deep Learning Accelerator (Intel® NervanaTM NNP-T)**, 2019, www.slideshare.net/insideHPC/deep-learning-training-at-scale-spring-crest-deep-learning-accel [letöltve: 2019. november 07.]

[13] Intel AI, **Introducing Intel Nervana Neural Network Processors for Training**, 2019, www.youtube.com/watch?v=Wj4ifr3DDFg [megtekintve: 2019. november 10.]

[14] Ofri Wechsler & Michael Behar & Bharat Daga, **Spring Hill (NNP-I 1000) Intel's Data Center Inference Chip**, 2019, www.slideshare.net/insideHPC/spring-hill-nnpi-1000-intels-data-center-inference-chip/ [letöltve: 2019. november 07.]

[15] Intel AI, **Introducing Intel Nervana Neural Network Processors for Inference**, 2019, www.youtube.com/watch?v=fTMIjQePWog [megtekintve: 2019. november 10.]

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

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

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

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

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

[21] **Tensor processing unit — Wikipedia, The Free Encyclopedia**, 2019, en.wikipedia.org/w/index.php?title=Tensor_processing_unit&oldid=923241091 [Meglátogatva: 2019. november 4.]

Függelék

F.1. Példa