SZAKDOLGOZAT

Jeney Máté

Debrecen
2021

Debreceni Egyetem
Informatikai Kar

Gépi tanulás Python által vezérelt FPGA alapú PYNQ kártyán

Témavezető: Dr. Oniga István László

egyetemi docens

Készítette: Jeney Máté

Mérnökinformatikus BSc hallgató

Tartalomjegyzék:

1. Bevezetés

Jelen korunk egyik kialakulóban lévő, feltörekvő technológiája az agy-számítógép interfészek (Brain Computer Interface - BCI). Ezen technológia nagyon sokrétű, egy alkalmazási területe lehet mozgásszabadságukat elvesztett, idős, vagy betegséggel küzdő személyeknek az életminőségét különböző módokon javítani[1].

A felhasznált eszköz, amit a Debreceni Egyetem Informatikai Karának Informatikai Rendszerek és Hálózatok tanszéke biztosított, az az OpenBCI cégnek, az "EEG Electrode Cap Starter Kit" nevű, 16 csatornás, és 16 csatorna használata esetén 125Hz-es mintavételezési frekvenciával rendelkező elektródákkal ellátott sapka, adó és vevő kombinációja[2]. Több megközelítés is lehetséges a fent említett életminőség javítására, két féle módszer, ahonnan én megkörnyékeztem a problémát az egy kezdetleges, teszt jellegű megvalósítása monitorozásnak és irányításnak.

A monitorozást három különböző tudatállapot vizsgálatával valósítom meg. A három tudatállapot a meditatív, aktív figyelem, és zene hallgatása, ezen utóbbi megfeleltethető az elme relaxált, nyugodt állapotának.

Az irányítás rész megvalósítása során a kiindulási pont korlátozott mozgásképességű személyeknek a motorikus funkcióinak az igen egyszerű formában történő pótlása volt. Ennek a kivitelezéséhez az EEG sapka elektródáira izommunkával (arcmimikával) zajt applikálva lehetséges specifikus, adott izomcsoport együttes aktiválására jellemző adatokat generálni, majd az adott mimikai motívumhoz egy kimenetet társítani. Esetemben a kimenetek a balra, jobbra, előre, állj és „üresjárat". Ezen kimenetek segítségével egy átalakított RC (Remote Controlled - távirányítású) autót irányítok.

A bemeneti adatok és a kívánt kimenet közötti transzformációt, adatfeldolgozást egy konvolúciós neurális hálózat (Convolutional Neural Network - CNN) segítségével valósítom meg, amely a gépi tanulási módszerek egy lehetséges eszköze. Neurális hálózatok segítségével statisztikai módszerekkel lehetséges egy modellnek, reprezentációnak megtanulnia a bemeneti adataiban rejlő mintázatot, majd új adatokon a már megtanult mintázatelemek segítségével egy becslést adni arról, hogy milyen kimenet tartozhat az éppen aktuális bemenethez. Az általam használt neurális hálózat típusa konvolúciós neurális hálózat, amelyet a fent említett kettő megközelítési módszerre jellemző adatokkal tanítottam be.

A szakdolgozatom másik fontos eleme a TUL cég, "PYNQ Z2"-nevű fejlesztőkártyája, amit szintén a Debreceni Egyetem Informatikai Karának Informatikai Rendszerek és Hálózatok

tanszéke biztosított a szakdolgozatom megírásának időtartamára. Ez a fejlesztőkártya gyors prototípustervezésre van optimalizálva, könnyen kezelhető, Jupyter Notebook alapú kezelőfelülettel, és nagy számú tutoriallal van ellátva[3] [4] [5]. Igénybevételének az oka, hogy az általam alkalmazott, már betanított neurális hálózatot nem CPU (Central Processing Unit)-n, vagy GPU (Graphics Processing Unit)-n kívántam inferenciára, azaz élő, a neurális hálózat által addig még nem látott adatok alapján egy kimenet predikciójára használni, hanem egy FPGA-n. Tipikusan a fent említett CPU, vagy GPU a célplatform inferencia esetén, azonban nagyobb energiahatékonysággal, célspecifikusan lehetséges a hálózatot az átalakításakor paraméterezni annak megfelelően, hogy milyen célhardverre készül az implementációja az átalakítani kívánt neurális hálózatnak[6]. A konvolúciós neurális hálózatot amit használtam, a Google Tensorflow nevű Python moduljának a Keras nevű API-jával (Application Programming Interface) készítettem és tanítottam, majd egy ponton kialakítottam a QKeras Python könyvtár segítségével egy kvantált, azaz lebegőpontos számábrázolás helyett fixpontos számábrázolású reprezentációját a modellnek, amit szükséges volt újra betanítani.

Ezt az új reprezentációt, modellt adtam tovább a hls4ml nevű Python könyvtárnak, aminek a segítségével lehetséges átalakítani neurális hálózat modelleket magasszintű szintézis (High-Level Synthesis - HLS) formátumba, és a szintézis után, mint szellemi tulajdont (Intellectual Property - IP) lehetséges beilleszteni a Vivado fejlesztői környezetbe[7], vagy kész Vivado projektet létrehozni belőle. Szakdolgozatomnak egy fontos aspektusa vagy szemléletmódja egy olyan rendszerösszeállítási módszer, amely a frameworkok, vagy a háttérben zajló folyamatok mélyreható ismerete nélkül lehetővé teszi egy, a témával még csak ismerkedő személynek a dizájnok gyors iterálhatóságát, fejleszthetőségét. Ebben ezen hls4ml könytár sokat segített, ennek a követelménynek teljességgel megfelel.

A PYNQ-Z2 eszközön található egy Xilinx gyártmányú "ZYNQ" 7020, rendszer a chipen (System on Chip - SoC) tokozású FPGA-CPU hibrid, amelynek a hibrid dizájnból fakadó nagyfokú flexibilitásával lehetséges egy közös platformon használni magasszintű programozási nyelveket, és FPGA-n implementált, azon gyorsított részfeladatokat a programozói környezetből, mint függvényeket meghívni[8].

Ezen fenti eszközök segítségével kapom el élőben, adatkeretekbe szervezve az OpenBCI EEG Electrode Cap Kit-től jövő információt. Az adatok előfeldolgozását, zajmentesítését, a neurális háló bemenetének megfelelő formátumba átalakítását hagyományos módszerrel, CPU-n végzem el, majd ezt az átalakított adatcsomagot adom át a PYNQ-Z2 eszköznek, ami elvégzi a

szükséges kalkulációkat, majd egy kimenetet ad vissza, amit irányítás esetén soros porton küldök el az RC autó irányító egységeként szolgáló ESP32 modulnak, ami az általa kezelt aktuátoroknak küldi a kapott irányítási információnak megfelelő parancsokat, ezáltal képes irányítani az autót.

A tudatállapotot vizsgáló esetben a PYNQ-Z2-ig az irányítási esettel megegyező az általam használt metódus, a kimenetet szintén soros porton kapom vissza, azonban az adatokat összegyűjtöm, és az adott tudatállapot mérésére szolgáló idő lejárta után ellenőrzöm hogy mekkora százalékban adott helyes predikciót az FPGA-n futó hálózat.

2. Elméleti háttér

2.1. Gépi tanulás

A gépi tanulás jelentése olyan módon programozni számítógépeket, hogy azok adatokból képesek tanulni. Azon tudományterület, amely megadja a számítógépeknek a lehetőségeket, hogy képesek legyenek tanulásra, explicit programozás nélkül.

Például a spam filter egy olyan gépi tanulást használó program, amely elég spam (például felhasználók által jelzettek) emailt, és emellett elég nem-spam, azaz ham típusú üzenetet látva képes megtanulni hogyan tudja a spam üzeneteket észlelni és kategorizálni.

A gépi tanulási módszerek nagyon jók olyan problémák megoldására, amelyek sok finomhangolást igényelnek, vagy nagymennyiségű szabály köti őket. Gyakran képesek ezen módszerek a tradicionális megközelítésnél jobban teljesíteni egy egyszerűbb leírással. Olyan komplex problémákkal küzdenek meg amelyeket hagyományos módszerekkel nem lehetséges megoldani. Jók változékony környezetekben, új adatokhoz, körülményekhez képesek alkalmazkodni. Komplex problémák, és nagy mennyiségű adat esetén rálátást nyújt ezekre.[9]

Néhány példaalkalmazás:

- Gyártósoron a termékekről készült képek automatikus osztályzása
- Tumorok detektálása agyszkenneken
- Hírek automatikus klasszifikálása
- Személyes segéd, vagy chatbot létrehozása
- Hang alapú vezérlés
- Termék, tartalomajánló rendszerek
- Intelligens botok létrehozása számítógépes játékokhoz

Sok típusa és altípusa létezik, fő csoportjai a supervised, és az unsupervised rendszerek, előbbinek a két variánsa a regresszió és a klasszifikáció, míg utóbbinak a klaszterezés, vizualizáció, dimenzionalitáscsökkentés, asszociációs szabály alapú tanulás.

A főbb felmerülő problémák a gépi tanulás alapú rendszerekben:

- A traning set méretének elégtelensége
- Az adatok alacsony minősége
- Irreleváns tulajdonságok

- Túltanítás és alultanítás (underfitting, overfitting)

A supervised, felügyelt típus egy fontos része a neurális hálózatok, szakdolgozatomban ezzel foglalkoztam kizárólag. Ismertebb algoritmusok még ebben a kategóriában a lineáris regresszió, logisztikus regresszió, random erdők, döntési fák, és k-legközelebbi szomszédok. Ismertebb felügyeletlen, unsupervised algoritmusok például a K-Means klaszterezés, izolációs erdő, PCA, Kernel PCA, Eclat.[10]

2.2. FPGA

Az FPGA megnevezés a Field Programmable Gate Array rövidítése, amelynek a magyar megfelelője körülbelül a helyszínen programozható kapumátrix. A programozható áramkörök kifejlesztésének a mozgatórugója a digitális áramkörökdizájnok összetettségének az egyre növekedő mértéke volt, szükség lett olyan eszközökre, amelyekkel a digitális áramkörök leírása elszakad a papíron tervezéstől, és valamilyen struktúra szerint programatikus módon leírhatóvá válhatnak. Az első újrakonfigurálható áramkörök voltak az EEPROM, PLD, CPLD, majd 1985-ben az 1984-ben alakult Xilinx cég XC2064-ös FPGA-ja lett az első FPGA a piacon[11].

Az FPGA-k ún. CLB-k, Configurable Logic Block (konfigurálható logikai blokk) mátrixa, amelyek programozható összeköttetések segítségével kapcsolódhatnak egymáshoz. Ezen hardveres blokkok többféle módon felépülhetnek, a gyakori összetevői LUT (Look Up Table), FF (Flip-Flop), MUX (multiplexer), Full Adder. Ezen blokkoknak a vezetékek és a kapcsolómátrix segítségével megfelelő módon történő összekapcsolása bármely digitális dizájn létrehozására alkalmassá teszi ezen eszközöket. A design leírására általában úgynevezett RTL (Register Transfer Level) nyelveket használunk, amelyeknek a hagyományos programozási nyelvekhez nagyon kevés közük van, elemi fő különbség a kettő között hogy míg a hagyományos programozási nyelvek, például C, C++, Rust szekvenciális alapú kódvégrehajtást biztosítanak, addig az RTL nyelvek, például Verilog, SystemVerilog, VHDL, a mérnök által leírt, szöveg alapú specifikációját biztosítják, különböző előre definiált nyelvi elemek segítségével a hardver kívánt működésének.

Nagyfokú párhuzamosságot tesz lehetővé ez a fajta leírásmód, így minden olyan alkalmazási területen, ahol szükséges feltétele a rendszer megfelelő működésének a hasonló, vagy ugyanolyan műveletek parallelizált nagyon nagy számú végrehajtása, igen hatékonyan alkalmazhatóak FPGA-k. Ilyen például neurális hálózat inferencia gyorsító,

jel/video/kép/hangfeldolgozás, nagy sávszélességű rendszerekben például base stationokben való felhasználás, ASIC (Application Specific Integrated Circuit) prototípustervezés, ADAS (Adaptive Driver Assistant System - adaptív vezetéssegítő rendszerek), sugárzástoleráns gyártástechnológia vagy tokozás esetén űreszközökben is felhasználható[12] [13].

2.3. ZYNQ

A Xilinx cég által kifejlesztett ZYNQ eszközök olyan CPU-FPGA hibridek, SoC-k, melyeknek a tokozásában egymás mellé van elhelyezve a processzor és a programozható logikai rész. Ezen nagyfokú közelség lehetővé teszi a kábelezés hosszának a minimalizálását. A kettő térfél nagyteljesítményű AMBA AXI interkonnektrendszer segítségével van összekötve.

Az AMBA segítségével integrált részegységek nagyon gyorsan tudnak adatokat cserélni, jóval gyorsabban mint olyan rendszerek esetén, ahol például a nyáklapon (Printed Circuit Board - PCB) külön van egy szoftveres feldolgozást biztosító/segítő nagyteljesítményű processzor vagy mikrokontroller, és valahol máshol ugyanezen nyáklapon található egy FPGA, ami egyedi funkciókat lát el, például telekommunikációs bázisállomások, nagyteljesítményű digitális mérőműszerek, oszcilloszkópok esetén szükséges FPGA-kkal biztosítani az adatfeldolgozás azonnaliságát[14].

Egy lehetséges megoldásnak tűnhet a problémára az FPGA-ban közvetlenül implementálni egy processzort, a Xilinxnek például MicroBlaze, vagy PicoBlaze mikrokontrollereit, léteznek nyílt forráskódú dizájnok is, például PicoRV32, DarkRISCV, a probléma ezzel az opcióval, hogy egyrészt az FPGA erőforrásaiból igen sokat elvesz egy-egy ilyen ún. “core", másrészt pedig a sebessége is korlátozott, néhány száz MHz-en clockolnak általában az FPGA részegységei. Egy másik lehetséges megoldás lehet a fent említett egy nyáklapon külön processzor, külön FPGA megoldás, amivel a probléma azonban a PCB huzalozásának nagyléptékű komplexitásnövekedése, elérhető maximális adatátviteli ráta csökkenése és késleltetési problémák, amelyeknek ha lehetséges a kiküszöbölésük valamilyen mértékben, az tovább bonyolítja a PCB dizájnt.

Fő részegységeinek a nevei Programmable Logic (PL), és Programmable System (PS).

2.3.1. PL

A programozható logikai egység, az FPGA rész a ZYNQ SoC-n belül. Néhány fontos paramétere, amelyek a felhasznált ZYNQ eszköz függvényében változnak:

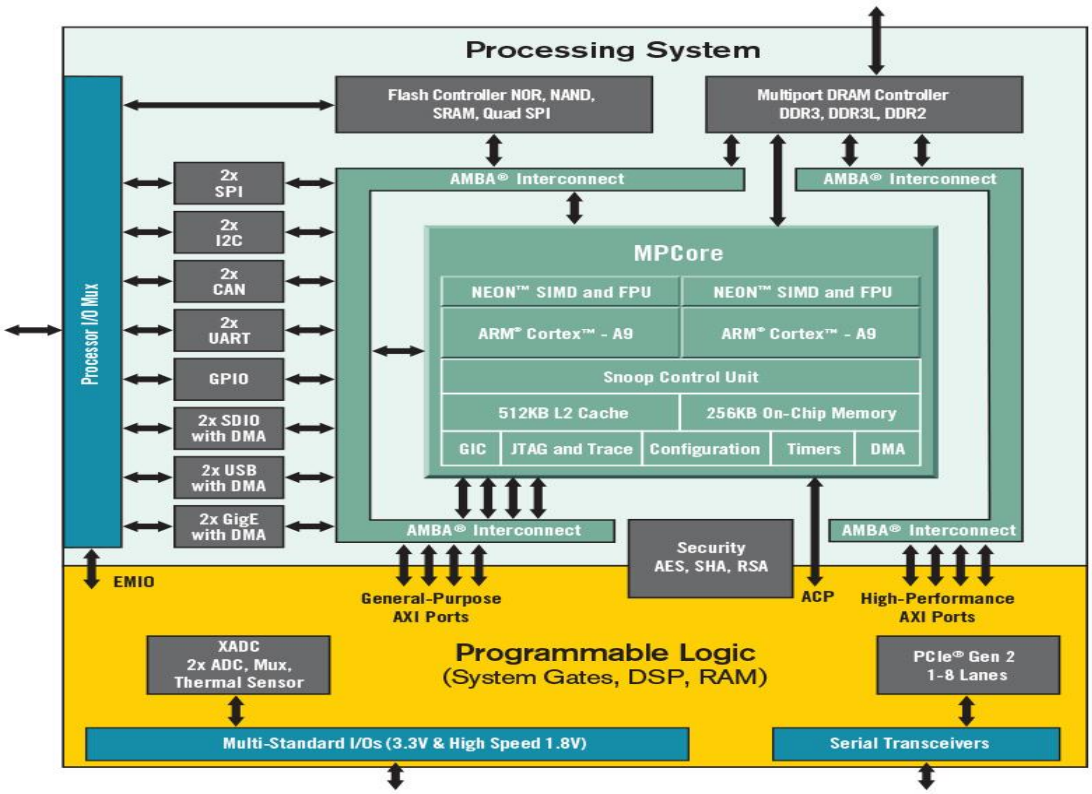
„Ekvivalens FPGA a Xilinx 7-es sorozatból: Artix-7 / Kintex-7
Logikai cellák száma: 23 ezer – 444 ezer
LUT-ok száma: 14400 – 277400
Block RAM mennyisége: 1.8 Mb - 26.5 Mb
DSP szeletek száma: 66 – 2020”[15]

2.3.2. PS

A szoftveres oldalért, a PL felkonfigurálásáért felelős processzor, az adott ZYNQ eszközök függvényében a modellek közötti különbséget a processzormagok száma, és a frekvencia jelenti:

„Processzormag: Single/Dual Core ARM Cortex A9 MPCore 766 MHz – 1000 MHz
L1 Cache: 32KB utasítás, processzoronként 32KB adat
L2 Cache: 512KB
Adatmemória (On-Chip Memory): 256KB
Támogatott másodlagos memóriatípusok: DDR3, DDR3L, DDR2, LPDDR2
DMA Csatornák száma: 8
Perifériák: 2x UART, 2x CAN 2.0B, 2x I2C, 2x SPI, 4x 32b GPIO
Perifériák beépített DMA-val: 2x USB 2.0 (OTG), 2x Tri-mode Gigabit Ethernet, 2x SD/SDIO”[16]

ábra 1
A két maggal rendelkező ZYNQ rendszerek blokkdiagramja[17]



2.4. PYNQ

A Xilinx cég a ZYNQ, és a ZYNQ UltraScale+ eszközökhöz létrehozott olyan egy keretrendszert, aminek a segítségével csökkentették a programozható logikai áramkörök létrehozásának, rendszerbe integrálásának a nehézségi fokát, segítségével addig kizárólag szoftveres tapasztalattal rendelkező mérnökök, architektek válthatnak a korábbi módszereknél jóval gyorsabban nézőpontot a hardveres gyorsítás felé. Ezen eszközrendszer neve PYNQ, avagy Python Productivity for ZYNQ.

A rendszer által használt programozási nyelv a Python, ami magas absztrakciós szintje révén gyors, kezdők számára is könnyen kezelhető, és produktív lehetőséget biztosít hardveresen gyorsított ún. overlayek integrálására.[18] [19]

A PL-ben használt logikai áramköröket a szoftveres könyvtárakhoz hasonlítható overlayek reprezentálják, ezen logikai gyorsítókat, az API-jukon keresztül lehetséges kezelni. Az overlayek, azaz a hardveres blokkok létrehozásához nem elégséges a Jupyter Notebook

Pythonos, vagy terminálos felületein navigálni, szükséges hozzá Vivado Design Suite, és/vagy Vivado HLS, vagy újabb verziók esetén ezek Vitis-beli megfelelői.

2.5. PYNQ-Z2

A Taiwani TUL cég készítette a PYNQ-Z2 eszközt, ami egy költséghatékony, nagyszámú interfésszel ellátott fejlesztőkártya. Specifikációi:

„ZYNQ eszköz: XC7Z020-1CLG400C

Kétmagos, 650MHz-es ARM Cortex A9 CPU

8 DMA csatornával rendelkező memóriakontroller

Nagy sávszélességű perifériás kontrollerek: 1G Ethernet, USB 2.0, SDIO

Kis sávszélességű perifériás kontrollerek: SPI, UART, CAN, I2C

Programozható a következőkről: JTAG, Quad SPI, microSD

Az FPGA, programozható logikai rész Xilinx Artix-7-nek megfelelő

13300 logikai szelet, 6 LUT-os technológiával

220 DSP szelet

XADC

DDR Memória: 512MB DDR3

Flash memória: 16MB Quad-SPI Flash

Áramellátása vagy microUSB-ről, vagy 7 és 15 volt közötti tápegységgel lehetséges DC tápfeszültségcsatlakozó segítségével

Internet: 10/100/1000 Ethernet

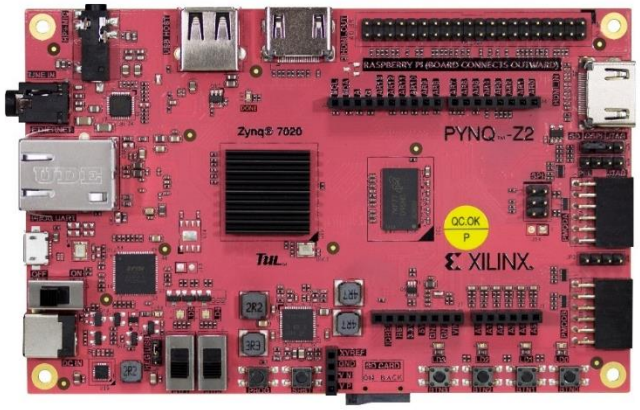
Az USB Host a PS-hez, az ARM processzorhoz van kapcsolva, a programozható logika a PS-en keresztül képes csak elérni.

Egyéb I/O-k listája: 6 LED, 4 nyomógomb, 2 csúszókapcsoló

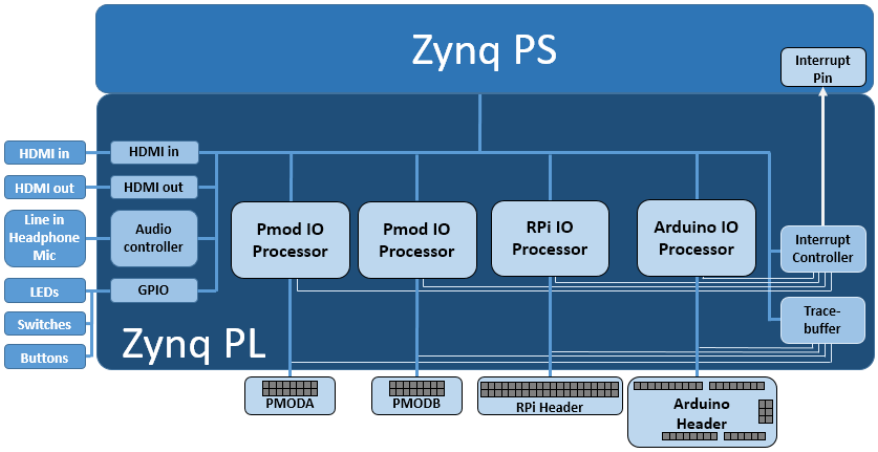
GPIO interfészek: Arduino Shield konnektor, Raspberry Pi konnektor, 2 Pmod port”[20]

A fejlesztőkártya használatához szükséges PYNQ rendszer microSD kártyára írása után, azt a fejlesztőkártya microSD foglalatába helyezve a beágyazott linux rendszer bootja, a base overlay felkonfigurálása után az eszköz használatba vehető.

ábra 2
A PYNQ-Z2 eszköz kinézete[21]



ábra 3
A PYNQ-Z2 eszköz blokkdiagramja[22]



2.6. hls4ml

A hls4ml Python modul eredete a CERN-ben található nagy hadronütköztető detektorainak az adatfeldolgozóképességének a maximalizálása volt, az LHC-ben a legnagyobb (ns nagyságrendű) felbontással rendelkező érzékelőknél használnak ML (Machine Learning - gépi tanulás) alapú algoritmusokat, fő motiváció a rendszer üzemeltetésénél, tervezésénél, hogy a rendszer rendelkezzen igen alacsony késleltetéssel, és emellett legyen nagy az adatáteresztőképessége.

A hls4ml csomag képes átalakítani neurális hálózatok különféle módokon leírt reprezentációját HLS segítségével szintetizálható formátumba átalakítani, majd a szintetizált IP-t lehetséges kész blokként beilleszteni Vivado környezetbe. A csomag tervez támogatni Mentor Catapult alapú ASIC-flow-ot is, azaz lehetséges lesz vele ASIC szimulációt végezni FPGA-kon.

Az FPGA erőforrásainak optimális kihasználásának az érdekében több lehetséges, egymással kombinálható technikát is alkalmaz a hls4ml.[23] [24] [25]

2.6.1. Kvantálás I.

FPGA-k esetén fixpontos számábrázolással dolgozunk, és a neurális háló szerkezetének külön-külön rétegeiben explicite meg tudjuk mondani, hogy hány biten legyen reprezentálva mind az egész rész, mind a törtrész, ezáltal lehetséges finomhangolni adott rétegeket.[26]

2.6.2. Parallelizáció - DSP felhasználtság:

A hls4ml megad egy paramétert, amit "ReuseFactor"-nak neveztek el, és ezzel lehetséges szintén a háló rétegeiként külön-külön beállítani, hogy egy darab szorzó egység hányszor legyen használva a rétegben található szorzásokhoz. Minél nagyobb ez a ReuseFactor érték, annál kisebb az adatáteresztőképesség és annál nagyobb a késleltetés, cserébe azonban kevesebb helyet foglal a dizájn az FPGA alkotóelemeiből, és ahogy nő a szám, egyre kisebb mértékben párhuzamosítottak a számítások. Hasonlóképpen, minél kisebb ez a szám, annál több helyet foglal a dizájn majd az FPGA-ból, cserébe nagyobb sebességgel lesz képes működni, és kisebb lesz a késleltetés. 1-re állítva az értéket azt jelentjük ki, hogy a rétegben található összes multiplikációhoz legyen rendelve külön szorzóegység, így teljesen párhuzamos lesz a végrehajtás.[27]

2.6.3. Kompresszió - Weight Pruning ("súly metszés"):

Az egyik fajta neurális hálózat optimalizációs technika a háló "metszése", ilyenkor megpróbáljuk a neurális háló futtatása közben végbemenő számítások számát csökkenteni a rétegek közötti kapcsolatok számának csökkentésével. Ezt a tensorflow sparsity toolkitje úgy oldja meg, hogy azon paramétereket kinullázza a neurális hálóban, amelyeket a hálók közötti haszontalan kapcsolatnak ítél meg. A hls4ml egy tutorial diasorában 70%-os kompresszióval 70%-kal képesek lecsökkenteni a használt DSP-k számát.[28]

2.6.4. Kvantálás II.

A QKeras a Keras egy olyan kiterjesztése, amelynek segítségével lehetséges kvantált neurális hálózatreprezentációkat létrehozni, ezen felül a modul lehetővé teszi kis mennyiségű kód

megváltoztatásával Keras által definiált neurális hálózat rétegeket egy-az-egyben lecserélni a réteg QKeras verziójára.[29]

2.7. Neurális Hálózatok, CNN

„Az első mesterséges neuronhálózatokat, 1943-ban vezette be Warren McCulloch és Walter Pitts. A biológiai neuronok egy nagyon leegyszerűsített modelljét vezették be, ez később a mesterséges neuronként vált ismertté. A modelljükben egy neuron aktiválja a kimenetét ("tüzel"), ha egy megadott mennyiségtől nagyobb számú bemenete aktiv. Egy ilyen leegyszerűsített modellel is lehetséges különféle logikai műveleteket végezni.

Az egyik legegyszerűbb mesterséges neuronhálózat architektúrát Frank Rosenblatt találta meg 1957-ben. Egy úgynevezett threshold linear unit (TLU)-n, vagy másképp linear threshold unit (LTU)-n alapszik. A ki-és bemenetek a korábbi bináris formátummal ellentétben számok, és minden bemeneti összeköttetés egy súllyal van társítva. A TLU a bemeneteinek a súlyozott összegét számolja ki: ($z = w_1x_1 + w_2x_2 + \ldots + w_nx_n = \mathbf{x}^T\mathbf{w}$). Ezután az összegre alkalmazza a lépcsőfüggvényt, és a kimeneten megjelenik az eredmény: $h_w(\mathbf{x}) = \text{step}(z)$, ahol $z = \mathbf{x}^T\mathbf{w}$.

Lineáris algebra segítségével hatékonyan képesek vagyunk kiszámolni egy mesterséges neuronréteg kimeneteit több példányra egyszerre.:

$h_{\mathbf{W},\mathbf{b}}(\mathbf{X}) = \phi(\mathbf{XW} + \mathbf{b})$

Ahol,

X jelzi a bemeneti mátrixot, minden példány (instance) esetén egy sort és minden kimeneti tulajdonság (feature) esetén egy oszlopot tartalmaz.

A **W** súlymátrix tartalmazza az összes kapcsolódó súlyt, kivéve a bias neuronokból érkezőeket. Bemeneti neurononként egy sora, és adott rétegben lévő neurononként egy oszlopa van.

A **b** bias vektor tartalmazza a bias neuronok és a mesterséges neuronok közötti súlykapcsolatokat. Mesterséges neurononként egy úgynevezett "bias term"-je van.

A ϕ függvénynek a neve az aktivációs függvény. Ha a mesterséges neuronok TLU-k, akkor ez a függvény lépcsőfüggvény.

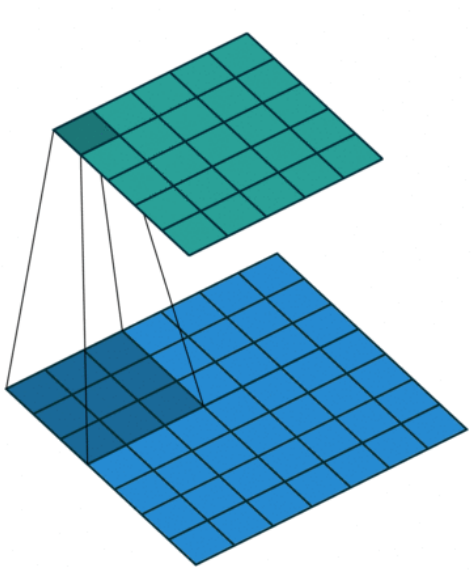
A Perceptronoknak nevezett egyszerű neuronhálózatok csak olyan problémákat képesek megoldani amelyek lineárisan szeparábilisak, azonban ezt a hiányosságokat lehetséges feloldani azzal, ha elkezdünk rétegeket felépíteni a Perceptronokból. Az így létrejövő hálózat neve MLP, vagy Multilayer Perceptron. Ezek felépítése a következő: egy bemeneti réteget követ egy vagy több réteg TLU, majd egy kimeneti réteggel zárul.

A konvolúciós neurális hálózatok (CNN) az agy vizuális kortexének a vizsgálatából eredeztethetőek, és a 80-as évek óta használják őket képfelismerésre. Jelenleg nagyon sokrétű az alkalmazásuk, önvezető autókban, kép alapú keresőalgoritmusokban, video alapú felismerő rendszerekben használják, de a CNN architektúra nincs képekre limitálva, lehetséges felhasználni NLP-re, azaz természetes beszédfeldolgozására, hangfelismerésre, vagy mint ezen szakdolgozat esetén, idősoros adatok feldolgozására.

Sok neuron a vizuális kortexben egy kicsi, úgynevezett lokális receptív mezővel rendelkezik, azaz a látómezőnek csak egy kis területére érkező stimulusra reagál, és vannak olyan neuronok, amelyek csak a körszerű alakzatokra vagy függőleges vonalakra, vagy ezek kombinációira aktiválódnak. Továbbá vannak olyan neuronok, amelyek az alacsonyszintű, egyszerű információt feldolgozó neuronok "kimeneteteire" kapcsolódnak rá. Ezen réteges architektúra vizsgálata vezetett a neokognitronhoz, és ebből fejlődött ki a ma ismert konvolúciós neurális hálózat. Yann LeCun et al. 1998-as cikke mérföldkő volt, mely bevezette a LeNet-5 architektúrát, amit bankok a kézzel írott számjegyek ellenőrzésére széleskörben használtak. Két fontos építőkövet emelt be ez a köztudatba, a pooling, és a konvolúciós rétegeket."[30]

Egy konvolúciós rétegről a következő animáció nyújt ismereteket:

ábra 4
A konvolúciós operátor működése[31]



Látható az animált ábrán, hogy a kék, bemeneti rétegben a kép adott számú pixelein, egy megadott méretű filter halad át. A kimeneti, zöld rétegben nincs a bemeneti réteghez képest

pixelenkénti egy-az-egyben megfeleltetés, azaz a kimeneti mátrix nem kell hogy ugyanakkora legyen mint a bemeneti mátrix. A sötétkékkel jelzett mozgó operátor, úgynevezett kernel jelzi, hogy a zöld réteg adott (sötétzölddel jelzett) neuronjai a kék rétegből mely neuronok kimeneteihez csatlakoznak. Ezen architektúra és ezen rétegek többszöri egymásra építésével lehetséges kialakítani a vizuális kortexhez működésben hasonló adatfeldolgozást, ahol a több rétegből álló felépítmény alján találhatóak a kicsi, alacsony szintű tulajdonságok detektálásáért felelős neuronok, míg a hálózat tetején találhatóak a komplex, magaszszintű tulajdonságok detektálásáért felelősek[32].

A neurális hálózatok (újra)áttörése 2012-ben következett be, amikor az AlexNet architektúra nagyon nagy léptékben, 17 százalékra lecsökkentette a hibarátát az ImageNet kihíváson[33].

2.8. TensorFlow / Keras

A TensorFlow egy numerikus számításokra alkalmas könyvtár, amely gépi tanulásra van finomhangolva, ezáltal könnyedén lehet erre használni különböző platformokon mind CPU, GPU, mind TPU-kon. A Google Brain csapata készítette, és a Google Cloud, Google Speech, Google Photos mind használja. 2015 novemberében nyílt forráskódúvá[34], és mára a legnépszerűbb mélytanulásra alkalmas könyvtárrá vált. A magja hasonló a NumPy-hoz, de támogat GPU-n és TPU-n futtathatóságot, elosztott számításokat, optimizációt, és sok API-t, mint például tf.keras, tf.data, tf.image, tf.signal. Alacsonyszinten C++-ban implementáltak a TensorFlow ún. operációk, lehetséges sajátokat is írni a C++ API segítségével[35].

A Keras neurális hálózatmodellek tanítására és leírására készített magasszintű API, kifejlesztése során a gyors kísérletezés, azok iterációja volt a cél. A TensorFlow-n fut. Használják a NASA-nal, a CERN-ben, az NIH-ban, és olyan cégek mint a Waymo, YouTube, Netflix, Uber, Yelp alkalmazzák[36].

EEG:

Az EEG (elektroencefalográfia, vagy electroencephalography) egy elektrofiziológiás technika, amellyel az emberi agyból származó elektromos aktivitást lehetséges mérni[37]. Több klinikai használata is kialakult mára, ilyenek például görcsök identifikálása, különféle agyi funkciók vizsgálata[38]. Ez egy noninvazív módszer, amely során az adatok csak kiolvasásra kerülnek, az agysejtekre nincsen stimulus applikálva.

EMG:

Az EMG (elektromiográfia, vagy electromiography) egy olyan technika, amely az izomtevékenységből származó elektromos impulzusokat képes detektálni. Ovosi esetekben képes megkülönböztetni, hogy az izomvesztés vagy gyengeség idegrendszeri, avagy izomrendszeri eredetű-e[39].

2.9. BCI eszközök

A Brain-Computer-Interface vagy agy-számítógép interfész egy biotechnológiai kiterjesztése az embernek, amelynek a célja az agyi aktivitás mérése és feldolgozása által elérni a kívánt funkciót. Az agyi aktivitás mérése, feldolgozása, és a kívánt funkció is lehet többféle. Az aktivitás mérésének metódusait két fő csoportra lehet osztani, invazív, és neminvazív technikákra. Invazív technológiáról beszélünk, ha az emberi szervezetbe helyezzük el azon egységet ami méri az agyi aktivitást, ilyen például a Neuralink cég által bemutatott "Link" eszköz, amely mikron nagyságrendű szálakat helyez el az agykérgen a mozgásért felelős neuronok közelében, amellyel az elektromos akciós potenciált képes mérni. Ehhez szükséges a koponyán egy lyukat ütni, majd egy robottal beültetni az agyszövetbe az elektródákat. Ezzel szemben a noninvazív technikákhoz nem szükséges az emberi testen beavatkozásokat végrehajtani, elég például EEG/EMG esetén egy elektródákkal ellátott sapkát felhelyezni, vagy az elektródákat a bőrre fixálni. Ezen utóbbi eszközök segítségével nem lehetséges az invazív eszközökhöz mérhető felbontást produkálni, többszöri felhelyezés esetén változhat a mért neuronok csoportja, kevéssé pontosak, azonban nincs szükség műtéti beavatkozásra, nem létezik még felhasználói kereskedelmi forgalomban lévő invazív eszköz, ezzel szemben nem invazívból több eszköz is elérhető, például: OpenBCI Electrode Cap, OpenBCI Ultracortex, EMOTIV EPOC. Szakdolgozatomhoz az OpenBCI Electrode Cap Kitet használtam.[40]

Az alábbi képen látható rajtam ezen eszköz oldalnézetből:

ábra 5
OpenBCI EEG Electrode Cap oldalnézetből

2.10. HLS

A folyamatosan gyorsuló technikai fejlődésnek része az elektronikai eszközök fejlődése, amely magával vonzza az azokat alkotó komponensek komplexitásának a növekedését. A jelenlegi IoT, AI, ML, 5G, és video/képfeldolgozó képességekkel ellátott eszközökben található gyorsítók leírása egyre komplexebb, a tranzisztorok száma növekszik, az eszközökben pédául az energiaellátás irányításáért felelős modulok egyre komplexebbek. Ezen komponenseket a tervezési fázisban szükséges valamilyen formátumban leírni, és erre az ipar leggyakrabban a VHDL, vagy a Verilog RTL nyelveket használja. A probléma ezzel a megoldással az absztrakciós szint alacsony szintje, a folyamat lassúsága, az iterációk váltakozásának a lassúsága, a verifikációval töltött időtartam növekedése, a produktivitás csökkenése. Ezen negatívumoknak a hatására jött létre a HLS, vagy magasszintű szintézis, amely C, C++, MATLAB, SystemC, nyelven történő leírást alakít át hardveresen szintetizálható RTL modellé. Segítségével a hardvermérnökök gyorsan tudnak iterálni a lehetséges designokon, magasabb absztrakciós szinten képesek viszonyulni a rendszerhez, így a szoftvermérnökök is jóval hamarabb férhetnek hozzá egy teszt FPGA-n a rendszer prototípusához. A szakdolgozatom során a Xilinx cég Vivado HLS 2019.1-es verzióját használtam. [41] [42] [43] [44]

2.11. Meditáció

Az ősi indiai eredetű hagyományok különféle módszereket fejlesztettek ki a testi-szellemi egészség fenntartására. A fő kategorizációs csoport a *yoga,* amely összekapcsolódást jelent. A yoga nyolc részre osztását Patanjali vezette be, és ma mint nyolcfokú jógalétra ismeretes. Ennek a jógalétrának az 5., 6., és 7. lépcsője feleltethető meg a ma meditációként ismert metódusoknak. Céljuk az elme elcsendesítése, az érzékek szabályozása, egy nyugodt tudatállapot elérése. Több módszer létezik a meditatív állapot elérésére, léteznek légzőgyakorlatok, figyelemösszpontosítást igénylő nyitott, avagy csukott szemes, gondolatok kizárását célzó gyakorlatok, a ma jógaként leggyakrabban eladott jógaászanák-testhelyzetek gyakorlása, mantrák, azaz egy szó, szavak, vagy hangok repetitív ismétlése, vizualizációs technikák, awareness, avagy flow típusú módszerek. Nagyszámú kutatás implikálja, hogy sokrétű testi, és mentális eredetű betegségek tüneteit képes enyhíteni, mint például: alvászavarok, IBS, depresszió, szorongás, magas vérnyomás. A szakdolgozatom során legfőképpen a mantra meditációra koncentráltam, ahol a *hare krishna hare krishna krishna krishna hare hare hare rama hare rama rama rama hare hare* szavakból álló, 16 szavas Hare Krishna mantra ülő helyzetben történő repetíciója a meditatív folyamat, amit gyakoroltam az adatok gyűjtése során. [45 46 47 48]

3. A projekt bemutatása

3.1. A kezdeti rendszer összeállítása

Elsőnek felkutattam a probléma keretrendszerbe foglalásának a lehetséges opcióit, a terv eredetileg több, PYNQ-n alkalmazható, gépi tanulásra alkalmas módszer összehasonlítása, majd a megfelelő kiválasztása volt. Az általam alkalmazott hálózat konvolúciós neurális hálózat, igy céltudatosan kerestem ennek megfelelő keretrendszereket és találtam a következőket: Vitis-AI, DNNDK/DPU-PYNQ, hls4ml. A dokumentációk és a fórumokon olvasható felhasználói tapasztalatok alapján kiderült, hogy a Vitis-AI és a DNNDK/DPU-PYNQ frameworkök nem kompatibilisek a PNYQ-Z2 eszközzel, más PYNQ eszközzel képesek működni, azonban nem volt hozzáférésem más PYNQ eszközhöz csak a PYNQ-Z2-höz, így a hls4ml képességeit figyelembevéve úgy döntöttem az lesz a megfelelő, gyors prototípustervezés szellemiséggel egyező módszer a neurális hálózat FPGA-n implementálhatóvá alakításához.

A hls4ml framework tutorial jellegű opcióinak kipróbálása után az OpenBCI Electrode Cap Kittel foglalkoztam, az alapvető működés kipróbálására elsőnek az OpenBCI oldaláról letölthető OpenBCI GUI-t használtam, a GUI jelzi hogy mekkora mértékben tér el a normálistól az adott csatorna ellenállása, és ha ez az érték elér egy határátéket, akkor kiírja az adott csatornán, hogy “Railed”, és nem lesz látható hullámforma az adott csatornának megfeleltetett gráfon, csak egy egyenes vonal. A későbbiekben a GUI-t adatfelvételek előtt használtam a csatornák jelminőségének ellenőrzésére. A tápellátását az eszköznek először egy olyan DC-DC konverterrel próbáltam megoldani aminek a bemenete egy laptop töltő, de a környezetből rendkívül nagy mennyiségben vett fel elektromágneses zajt, ami miatt a felvételek használhatatlanok voltak. Ekkor rájöttem hogy a zajt a tápellátás zajszűrésének a hiányossága okozza, és váltottam egy LiFePO 9.6V-os cellapakkra, aminek a használatával megszűnt a zajprobléma. Az OpenBCI GUI kevés konfigurálhatósági lehetőséget biztosított, emiatt a kezdeti kipróbálás után váltottam a Brainflow könyvtár Python nyelvű API-jára, amely használatával megírtam az első verzióját az adatfelvételre kihegyezett Python szkriptnek.

Ezen szkript segítségével elkezdtem felvenni az első adathalmazokat, amelyeket mint teszt kívántam alkalmazni a tudatállapotokat összehasonlító, egyéneket monitorozáson keresztül figyelő életminőségjavító megközelítésben.

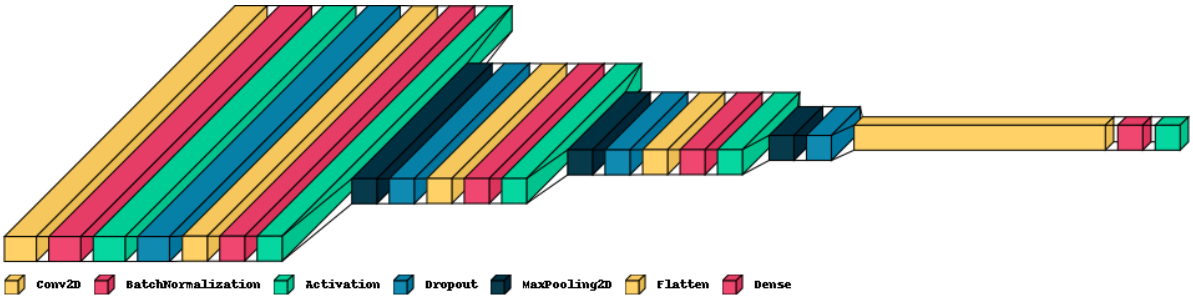
Ezek után elkezdtem kísérletezni a másik, bénult embereket segíteni képes módszer vizsgálatával, a mimikán keresztüli információfelvétellel. Ehhez internetről szabad

felhasználású, ingyenes képeket töltöttem le adott irányokba mutató, nem egységes anyagú, elhelyezkedésű, formájú nyilakról készült fotókat. Ezen első verzió során a megfeleltetést a következő módon csináltam: jobbra irány - jobb szemöldökömet emeltem fel, balra irány - bal szemöldököm emeltem fel, előre irány - mind a kettő szemöldököm felemeltem, stop/állj - állkapcsomat összeszorítottam. Python nyelven létrehoztam a kimeneti osztályokat számokkal reprezentálva, minden irány a neurális háló kimenetén egy-egy számnak feleltethető meg. Ezen kezdeti tesztekkor nem vettem figyelembe, hogy az OpenBCI EEG sapka 16 csatorna használata esetén csak 125Hz-es mintavételezési frekvenciára képes, és 512-esével gyűjtöttem be az adatokat, ami azt jelentette, hogy egy kép kevéssel több mint négy másodpercig volt jelen a kijelzőn, és ezalatt végig kellett a kívánt iránynak megfelelő mimikai akciót kitartanom. A képmegjelenítést matplotlib és a subprocess Python könyvtárak segítségével oldottam meg. Az adatok begyűjtése közben a Brainflow Python API-jának jelfeldolgozó függvényei közül a következőket használtam:

DataFilter.detrend(data[i], DetrendOperations.CONSTANT.value)
DataFilter.remove_environmental_noise(data[i], 125, NoiseTypes.FIFTY.value)
DataFilter.perform_highpass(data[i], 125, 3.0, 4, FilterTypes.BUTTERWORTH.value, 0)

A data[i] jelent egy-egy csatornát a 16-ból a mérés során egy adott adatablakban.
Ugyanezen előfeldolgozási pipeline hajtódik végre inferencia során, és mind a két esetben a CPU-n végeztem el a pipeline lépéseit jelentő előfeldolgozó függvényeket. Az OpenBCI eszköz továbbító egysége (Cython - Daisy) által generált mátrix egy nem használt oszlopába az adott kép megjelenítése során, az adatfelvétel közben beírtam a képhez tartozó osztályt reprezentáló decimális számot. Ezek után az alábbi képen látható architektúrájú neurális hálózat a Keras modullal történő leírása után betanítottam a neurális hálózatot.

ábra 6
A kezdeti neurális hálózat felépítése



A teszt célja még nem a hls4ml-lel történő átalakítás volt, így egy másik projektben használt átalakított távirányítós autón elhelyezkedő ESP32 modulnak küldtem el USB-n keresztül UART soros porton a neurális hálózat által generált kimenetet a Python Serial könyvtárának a segítségével, ahol adott, az UART-on érkező karakterek megfelelő kombinációinak hatására vagy a kormányzásért felelős szervómotornak kellett volna jobbra, illetve balra mennie, avagy az előre, illetve hátramenetelért felelős DC motornak kellett volna elindítania az autót, vagy megállítania.

A tudatállapotok vizsgáló esetben sem az előfeldolgozási pipeline-on, sem a neurális hálózat struktúráján nem változtattam, ebben az esetben is létrehoztam egy inferenciára használható tesztrendszert, majd kipróbáltam a három vizsgálni kívánt állapot közben.

3.2. A kezdeti rendszerről levont konzekvenciák, módosítások

Hamar kiderült ezen első tesztből, hogy nem lesz megfelelő 512 adatpont megérkezésére várni minden irányítási parancsra, így a neurális hálózat betanítására szolgáló Jupyter Notebookban a tanításért felelős cellákat módosítottam olyan módon, hogy elég legyen 64 adatpont beérkezésére várni, és már ezután történjen meg a predikció. Ezen megoldással elméletben másodpercenként kis híján kétszer lehetséges irányítási parancsokat kiküldeni az irányítandó eszköznek. Ezen teszt során már bíztatóbb, de még mindig használhatatlan eredmények születtek, jobbra kanyarodni az esetek nagy többségében képes volt az autó, a többi parancsra leggyakrabban fals negatívok születtek, ami azonban nem volt meglepő, mivel a felvett adatok mennyisége nagyon kevés volt, és a randomizáló függvény, aminek a segítségével a lehetséges osztályok közül megjelent egy-egy kép, leggyakrabban a “jobbra” iránynak megfelelő képet választotta.

A tudatállapotokat vizsgáló módszer esetén az 512 adatpont használatával nem voltak problémák, maga az adatfelvétel is hosszabb ideig zajlik, nincs szükség kisméretű adatablakokra, így a továbbiakban ezt meghagytam ekkorának. A tesztek közben gyakran találkoztam azzal a problémával, hogy a háromból két osztályt egészen magas, legtöbbször 75% feletti hatékonysággal tudott klasszifikálni, míg a maradék osztályon nagyon alacsony pontosságot, 5% alattit produkált. Kis idő után rájöttem, hogy az úgynevezett data bias nevű problémát tettem bele a rendszerbe azáltal, hogy nem figyeltem arra, hogy egyforma mennyiségben legyenek reprezentálva az adott osztályt reprezentáló adatok. Ezt ekkor még nem több adat felvételével oldottam meg, hanem megnyírbáltam azon osztályok példányszámát amik a legjelentősebb mértékben voltak reprezentálva a datasetben, és amikor mind a három osztály 33%-os súllyal volt jelen, akkor traineltem újra. A kapott hálózat bíztató eredményeket produkált, az esetek több mint kétharmadában az adott osztályra a helyes predikciót adta.
Az adatelőfeldolgozási pipeline esetében mind a kettő felhasználási esetben látszott, hogy van még mit javítani rajta. A hálózatok tanítás során lassan, nehezen konvergáltak egy olyan minimum felé, ami nem adott megfelelő minőségű eredményeket, a predikciós pontosság mértéke nem volt kielégítő.

3.3. Külső adatgyűjtés kísérlet

Ezután kerestem, és találtam vállalkozó kedélyű embereket a három tudatállapot vizsgálatára, akik közül egy személyről (N.) kizárólag meditáció közbeni adatokat, másik három személlyel pedig mind a három osztálynak megfelelő adatokat vettem fel. Az eszköz hiányosságait is tapasztaltam eközben, N., akiről csak meditációt vettem fel jó minőségű adatokat szolgáltatott, a jel tiszta, zajmentes volt, kopasz fejbőre miatt. Amikor azonban a három másik személlyel vettem fel az adatokat, egyrészről mindőjüknek hosszú a haja, másrészről a helyszín ahol felvettük az adatokat (IK-Inkubációs Ház), elektromágnesesen nagymértékben szennyezett. Az OpenBCI GUI-val lehetséges ellenőrizni az adott elektróda ellenállását, vezetőképességét, és a GUI egy másik ablakában lehet látni a beérkező jelek FFT plotját, ahol a GUI-ban található notch filter bekapcsolása után is hatalmas csúcs volt látható 50Hz-nél, és hasonló problémáról kaptam információt, ugyanebben a laborban található oszcilloszkóp bekapcsolásakor azon is látható az 50Hz-es zaj. A hosszú haj azt jelentette, hogy az elektróda nem volt közvetlen közeli kapcsolatban a fejbőrrel, csak az elektródagélnek a hajtömeg miatt hiányos médiumán keresztül, így csak nagyon silány jelet tudott közvetíteni, és direkt kapcsolat hiányában az

elektromágnesességgel számottevően szennyezett helységből még nagyobb mértékben felvette a zajt, mint az eredmények mutatják annyira, hogy nem voltam képes az elérhető filterekkel a zaj hatását eliminálni. Az alábbi képen látható amint az egyik kísérleti alanynak az OpenBCI EEG Cap felhelyezése után elektródagéllel töltöm fel az elektródákat egy fecskendő segítségével.

ábra 7
Külső adatgyűjtés

A hosszú haj miatt a három személy esetében átlagosan 3-6 csatorna teljesen értelmezhetetlen adatokat produkált. A legjobb adatokat azon személyből voltam képes kinyerni akinek a legkevésbé volt tiszta a haja, mivel az elektróda az ő esetében tudott a legközelebb kerülni a fejbőréhez. Tekintve, hogy minden magamon, és N.-en felvett adatpont egyike sem volt olyan mennyiségben zajos mint a három személlyel felvett, így nem láttam értelmét a training setbe belerakni a róluk felvett adatokat. Hasznos lett volna ha tudom az adataikat használni, ugyanis mind a zenehallgatás és a tudatos figyelem médiuma más volt az esetükben, itthon a saját adataim felvétele közben kizárólag fejhallgatóval vettem fel a zenehallgatást reprezentáló

adatokat, míg velük hordozható hangszóróval történt ez, a tudatos figyelmet itthon oktató, tudományos, vagy elemző videókra figyeléssel oldottam meg, míg a három személy esetén egy "mesélő" mondott el valamilyen tudományos témáról egy rövid szóbeli előadást a másik személynek. A meditáció módszertana is különbözött az általam alkalmazottól, vizualizációs, vezetett meditációt tartott az egyik kísérleti alany a másik két személynek külön-külön, míg ő önmagától merült el egy vizualizációs típusú technikát alkalmazva a meditációjában. A következő képen látható amint az ő fején van az OpenBCI EEG Cap, és éppen feltöltöm az elektródákat elektródagéllel.

ábra 8
Külső adatgyűjtés 2

Ezen többletadatok a training setbe való beszámításával egy általánosabb, jobban generalizáló neurális hálózatmodellt lett volna lehetséges betanítani.

3.4. Az adatfeldolgozás finomhangolása

A felismerések után elkezdtem a Brainflow DataFilter operációinak a legjobb kombinációját megkeresni, ezt Jupyter Notebookban 10-15 neurális hálózat tanításával oldottam meg, más és más sorrendben, vagy más paraméterekkel használtam az adatokat filterező metódusokat és

függvényeket, elkezdtem továbbá kísérletezni az ablakmérettel is. A legjobb megoldást a következő pipeline nyújtotta:

DataFilter.detrend(data[i], DetrendOperations.CONSTANT.value)
DataFilter.remove_environmental_noise(data[i], 64, NoiseTypes.FIFTY.value)
DataFilter.perform_wavelet_denoising(data[i], 'haar', 2)
DataFilter.perform_highpass(data[i], 64, 3.0, 4, FilterTypes.BUTTERWORTH.value, 0)
data[i]=RobustScaler(quantile_range=(25,75),
copy="False").fit_transform((data[i]).reshape(-1, 1)).reshape(-1)

Ugyanezen adatfeldolgozási lépéseken mennek keresztül a tudatállapotokat vizsgáló esetben is az adatok, a különbség annyi csupán, hogy ahol 64 látható a fenti pipeline-ban, oda 512 kerül, mivel ott más méretű adatablakkal dolgoztam. Ezután a hls4ml könyvtár dokumentációját, és a GitHubon elérhető tutorial anyagokat kezdtem mélyebben tanulmányozni, amelyeknek a mentén kívántam az általam kreálandó rendszert kiépíteni.

3.5. A neurális hálózatkonverzió

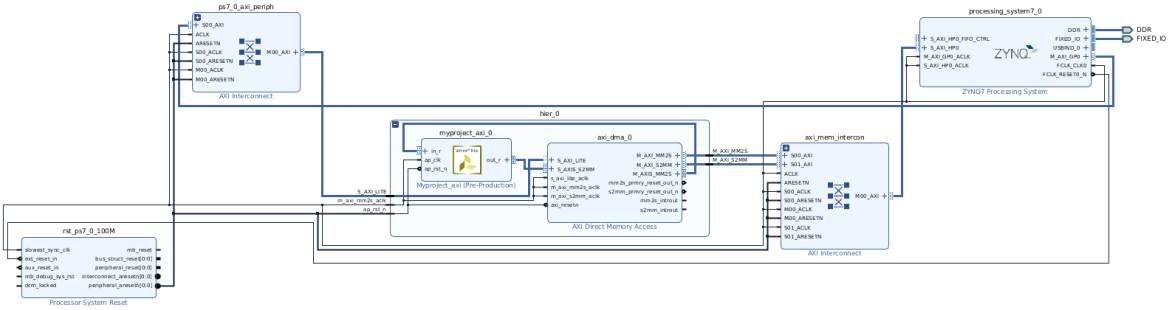
3.5.1. A kiindulási pont, a kezdeti hls4ml teszt létrehozása

A legmegfelelőbb opciót a design flow összerakására a GitHubon megtalálható hls4ml-tutorial egy forkjában találtam meg, egy CERN-ben dolgozó PhD. kutató létrehozott egy, a main branchbe a szakdolgozat írásának időpontjában be nem csekkolt, to-be-under-review verziót ahol konvolúciós neurális hálózatokat alakít át hls4ml segítségével[49]. Ezen a branchen található egyik tutorial notebook az SVHN-cropped datasetre készült konvolúciós neurális hálózatot hozott létre egy kvantált verzióval együtt. Nem volt azonnal működőképes, a virtuális Python környezetet felépítő yaml file által installált hls4ml verziója magasabb, mint ami a tutorial írásakor volt, így át kellett írni a kód egy részét, hogy a Keras, és QKeras segítségével létrehozott neurális hálózatmodell átalakítása végbemenjen. Miután ezt végigfuttattam, a hls4ml dokumentációjában találtam egy olyan átalakítási paramétert, amelynek segítségével az átalakítás során egy teljes, a neurális hálózat HLS-ben készült IP-jét Vivado Block Design-ban integráló kész Vivado projektet hoz létre.

3.5.2. Az átalakított neurális hálózat kimenete

Az alábbi képen látható a kreált block design.

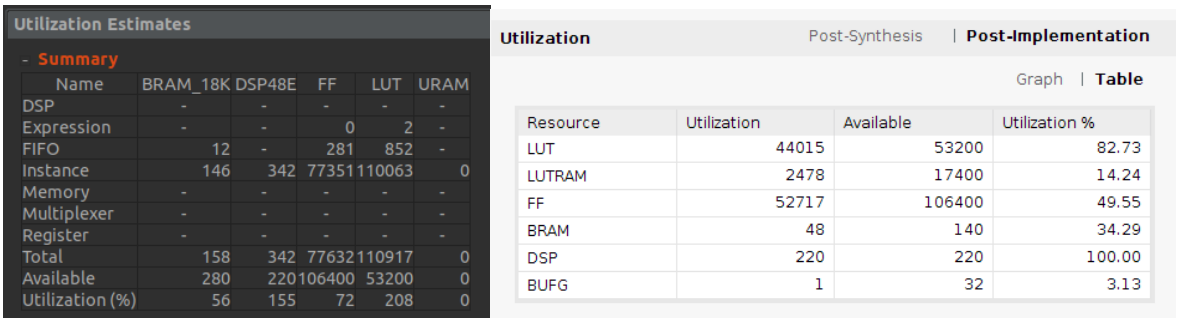
ábra 9
A tesztrendszer átalakításából származó block design kinézete



Bal oldalon lent látható a Processor System Reset, attól jobbra, felfelé található egy AXI Interconnect, ami felelős a rendszer összeköttetéseit biztosító ARM AMBA AXI protokollú összeköttetéshálózat absztrakciójáért, ettől jobbra található a hier_0 logikai egység ami magában foglalja a DMA modult, és a HLS által kreált IP-t. Ettől jobbra található egy újabb AXI Interconnect modul, majd jobbra, fent található a ZYNQ Processing Systemet reprezentáló blokk, amelyet, mint általában a blokkokat lehetséges duplakattintás után módosítani, de jelen esetben ez nem szükséges, a hls4ml kimenetébe nem szeretnénk belenyúlni, specifikusan a PYNQ-Z2-re készítette el ezt a Block Design-t.

Egy érdekesség ami szemembe ötlött, hogy a HLS jelentősen túlbecsüli az IP-nek, az FPGA fabricban elfoglalt területét. A következő két képen látható bal oldalt a HLS által, az IP-re megadott kihasználtsági becslés, és jobb oldalt a Vivado által megadott, tényleges helyfoglalása a teljes rendszernek.

ábra 10
A tesztredszer erőforrásigényei a HLS, és a Vivado becslésében



Utilization Estimates

- **Summary**

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	2	-
FIFO	12	-	281	852	-
Instance	146	342	77351	110063	0
Memory	-	-	-	-	-
Multiplexer	-	-	-	-	-
Register	-	-	-	-	-
Total	158	342	77632	110917	0
Available	280	220	106400	53200	0
Utilization (%)	56	155	72	208	0

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	44015	53200	82.73
LUTRAM	2478	17400	14.24
FF	52717	106400	49.55
BRAM	48	140	34.29
DSP	220	220	100.00
BUFG	1	32	3.13

Mint látható, a szükséges DSP-k számát 1.55-ször nagyobbra, míg a szükséges Look-up-Table-ök számát több mint kétszer akkorának becsüli a Vivado High Level Synthesis, a Flip-Flopok, és a felhasznált Block Ram mennyiségében nem téved túl sokat.

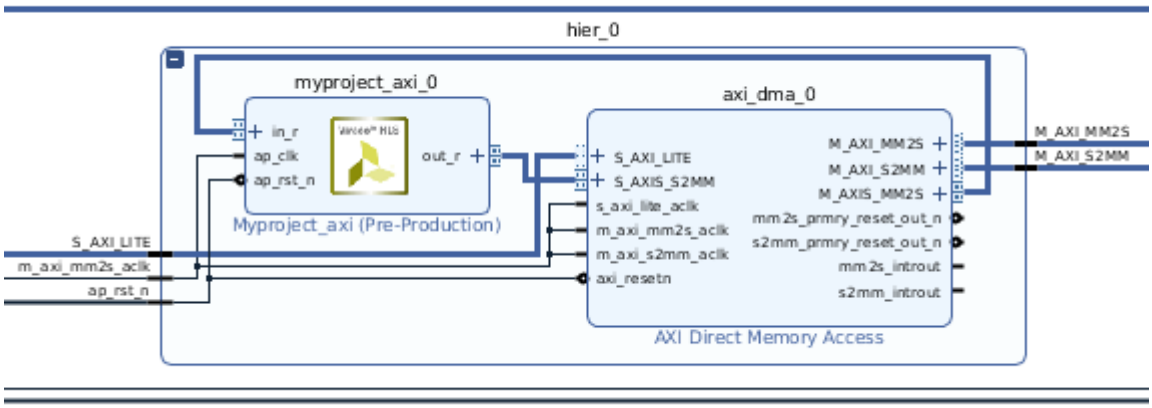
A hls4ml.converters.convert_from_keras_model segítségével lehetséges egy neurális hálózat modellt átalakítani, és ennek az objektumnak a "backend" paraméterét kell "VivadoAccelerator"-ra állítani, és ekkor készíti el a teljes, készen használható Vivado projektet. A Vivado környezetben az elkészített projekt megnyitása után lefuttattam a bitstream generálást, majd a File -> Export Hardware menüpontra kattintva exportáltam a hardvert leíró objektumokat. A .hdf kiterjesztésű objektum archive managerrel történő megnyitása után a .hwh és .bit kiterjesztésű fileokat kiszedtem az archívumból, majd a PYNQ futó Jupyter Notebook környezetet futtató szerverre felcsatlakoztam, és a bitstreammel egyetemben feltöltöttem a fileokat egy, az eszközön található könyvtárba.

3.5.3. A PYNQ-Z2 eszközön futtatása a teszt neurális hálózatnak

Ezek után a Block Design-ban található nevek alapján kikerestem az inferenciához szükséges IP-t, ami a Direct Memory Access-t jelenti jelen esetben, mivel mint a fenti képen látható, a DMA kapcsolja össze a neurális hálózatot realizáló HLS IP-t AXI Interconnecteken keresztül a ZYNQ Processing Systemet jelentő "IP"-vel, és ezzel együtt a memóriával. Az elérési útja a DMA-nak, az Overlaynek "hls4ml_design" néven FPGA fabric-ban történő realizálása, és ezáltal a Python környezetben, mint példányosított változó létrehozása után "hls4ml_design.hier_0.axi_dma_0" lett, mivel egy "hier_0" nevű logikai egységbe foglalja az átalakítás során a könyvtár a neurális háló IP-t és a DMA-t a hls4ml backend paraméterének VivadoAccelerator-ra történő beállítása esetén. Az alábbi ábrán látható ezen hier_0 logikai

egység Block Design-beli reprezentációja, kinyitás, azaz a modul bal felső sarkában található plusz gombra kattintás után.

ábra 11
A HLS IP-t és az AXI DMA-t egységbe foglaló logikai egység, a hier_0



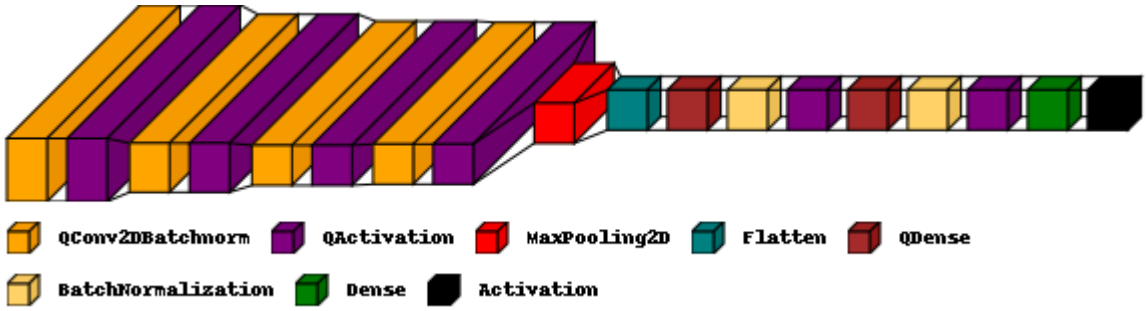
A teszthez szükséges képeket korábban áthelyeztem a kártyára, a numpy save rutinjának segítségével .npy formátumban, majd np.load()-dal egy változóba elhelyeztem négy darab tesztképet, mind egy-egy számot ábrázol.

A DMA használatához szükséges egy bemeneti, és egy kimeneti puffer, ezeket az xlnk segítségével oldottam meg. Ezen pufferek inicializálása után elhelyeztem a bemeneti pufferben az adott tesztképet, majd a példányosított DMA sendchannel.transfer() metódusával átadtam az IP-nek a memóriában összefüggő tömbként szereplő adatokat, és recvchannel.transfer() metódusával visszakaptam a neurális hálózat kimenetét. Ezen teszt sikerrel lezajlott, minden tesztképre a megfelelő kimenetet produkálta az FPGA-n futó neurális hálózat.

3.5.4. Az általam használt neurális hálózat konverziójának a megvalósítása

A hálózatátalakítás teszt sikeressége miatt elkezdtem a korábban elkészített távirányítós autót kontrollálhatóvá tévő tesztprojekt magjaként szolgáló neurális hálózatot a PNYQ-Z2-n keresztül futtathatóvá tenni. Ehhez szükség volt a teszt során csak Keras rétegeket használó neurális hálózatreprezentációmat QKeras alapúvá alakítani, és a hls4ml-ben elérhető hálózatritkító eljárásokat bevetni. A hálózat szerkezetén is módosítottam, és a következő struktúrát hoztam létre:

ábra 12
A kezdeti irányítási rendszer neurális hálózatának a felépítése



Mielőtt továbbléptem volna, végrehajtottam egy fontos ellenőrzési lépést. Meg kell vizsgálni, hogy a Vivado HLS által megkövetelt, 4096-nál kisebb loop unroll limitbe (ami a parallelizálhatóság limitjét jelenti) belefférnek-e a hálónak a rétegei, mert ha nem, akkor mivel nem ellenőrzi le automatikusan se a hls4ml, sem a HLS átalakítás előtt ezen feltétel teljesülését, így menet közben fog hibaüzenettel elbukni az átalakítás. Tekintve hogy csak egy szálon fut a HLS átalakítás, így komoly időbeli visszaesést jelenthet ezen limit ellenőrzésének a hiánya. A hálózat "metszését" (pruning) végeztem el ezután, és megnéztem mekkora mértékben váltak kinullázottá a súlyok. Ezek után a modell átalakítását végeztem el a hls4ml segítségével, az adott rétegeknek a paramétereiknek a száma alapján állítottam be a 'ReuseFactor' paramétert, ami jelenti a rétegben történő szorzásokhoz a DSP-k újrahasználhatóságát, azaz hány szorzáshoz lesz újrahasználva egy-egy DSP. A 'backend' paramétert itt is 'VivadoAccelerator'-ra állítottam, a könnyű, gyors és kompatibilis Overlay exportálhatóság végett. A fent említett tutorialhoz képest módosított hálózatstruktúrát jóval lassabban volt képes átalakítani a hls4ml, 4 órára volt szüksége a toolnak ennek végrehajtásához. Az alábbi képeken látható, hogy ebben az esetben is megfigyelhető a HLS túlbecslése a foglalt erőforrások tekintetében, érdekes módon a DSP-k számát alulbecsülte, ez azonban szerencsére az implementációban nem okozott gondot.

ábra 13

A kezdeti irányítási rendszer neurális hálózat átalakítása után a HLS és a Vivado által adott erőforrásigény

Utilization Estimates

- **Summary**

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	2	-
FIFO	1	-	86	272	-
Instance	174	137	60981	102421	0
Memory	-	-	-	-	-
Multiplexer	-	-	-	-	-
Register	-	-	-	-	-
Total	175	137	61067	102695	0
Available	280	220	106400	53200	0
Utilization (%)	62	62	57	193	0

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	31591	53200	59.38
LUTRAM	1425	17400	8.19
FF	43863	106400	41.22
BRAM	91	140	65.00
DSP	179	220	81.36
BUFG	1	32	3.13

A sikeres átalakítás után elkezdtem a PYNQ-Z2-n lehetővé tenni az inferenciát, ehhez Vivadobol exportáltam a hardvert, és a bitfile-t és a .hwh kiterjesztésű file-t ami tartalmazza a block design-ban található információkat, áthelyeztem a PYNQ-Z2-re. A neurális hálózat tanításához és kvantálásához szükséges kódon, ami megtalálható az oktatási anyagban, és amit a fent említett PhD. kutató kreált a képfelismerő hálózat PYNQ-n használhatóságának érdekében, nem változtattam, csak az epochok számát emeltem meg, mivel bármely más paraméter értékét módosítottam, csak rosszabb eredményeket kaptam mint az eredetiekkel. Hasonlóképp jártam el a későbbiekben is.

Az OpenBCI EEG eszközből érkező adatok feldolgozásáért felelős pipeline-t próbáltam meg ezután a PYNQ-Z2-n működőképes állapotba hozni, azonban kombatibilitási hibák miatt nem voltam képes ezt összehozni, a feldolgozási pipelineért felelős szükséges könyvtárakat, azaz a scikit-learn-t és a brainflow-t nem voltam képes működőképessé tenni az eszközön a próbálkozásaim alatt, így mint proof-of-work teszt, végrehajtottam a teszt adatokon a szükséges előfeldolgozást a PC-men, hagyományos CPU segítségével, majd az adatokat .npy kiterjesztésben elmentve áthelyeztem az eszközre. Ezek után ugyanúgy jártam el mint a fenti teszt esetben, példányosítottam a DMA-t, betöltöttem np.load segítségével az előfeldolgozáson átment teszt adatokat, és változókban elhelyeztem egyes elemeit az adatoknak, majd a példányosított DMA sendchannel és recvchannel objektumainak transfer(), wait(), _active_buffer.invalidate(), és _clear_interrupt() metódusaival lefuttattam az inferenciát. A teszt lezajlott, azonban a hálózat pontossága még a korábbi teszthez képest is komoly kívánnivalót hagyott maga után, az átalakítástól mentes, azelőtt csak CPU-n futtatott, tisztán keras alapú megoldásoktól messze elmarad a precizitása.

A lehetségességről megbizonyosodva elkezdtem ugyanezen fenti lépéseket elvégezni a tudatállapotvizsgáló esetben is, azonban a bemeneti adatok méretbeli különbségei miatt a

használt neurális hálózat paramétereit módosítanom kellett, nem használhattam akkora számú, és akkora méretű 2D-s konvolúciós filtereket mint az irányítási esetben, ezen hibát még a loop unroll limitet ellenőrző kódrészlet jelezte. További hibák jelentkeztek a Vivado projekt létrehozása során, a Vivado implementáció, ahogy iteráltam, azaz csökkentettem a filterek méretét és számát, növeltem a pooling layerek számát, elbukott minden alkalommal, és egy ponton végül, hogy lefusson hiba nélkül, a teljes hálózat precízitását csökkentettem le 2 bittel. Ezt egy paraméterrel lehetséges megtenni a *hls_config['Model']['Precision'] = 'ap_fixed<16,6>'* -t állítottam át *hls_config['Model']['Precision'] = 'ap_fixed<14,6>'*-ra Ekkor az átalakítás végbement hibaüzenet nélkül. Ezen rendszert a fenti, irányítás esetben használt módszerrel teszteltem, .npy kiterjesztésben áthelyeztem filterezett teszt adatokat a kártyára, majd lefuttattam helyben az inferenciát. A rendszer működőképesnek bizonyult, azonban ebben az esetben is látszott, hogy a rendszer pontossága elmarad a kívánatostól, de mégsem akkora mértékben mint az irányítási esetben.

3.5.5. Konklúzió

Mint a fentiekben leírtak alapján kiderült, hogy nem lesz lehetséges az adatok előfeldolgozását elvégezni a kártyán, így a tervezett dataflow a következő lett: az OpenBCI EEG Cap által gyűjtött adatot a számítógépemen előfeldolgozom, ezen előfeldolgozott adatokat eljuttatom a PYNQ-Z2-re, ott végrehajtom az inferenciát, majd a neurális hálózat kimenetét visszajuttatom a számítógépre. Az irányítási opció esetén ezen visszajuttatott információt továbbítom soros porton keresztül az RC autót irányító ESP32-nek, míg a tudatállapotokat vizsgáló opció esetén egy tömbbe elmentem a kapott adatokat, mivel ezen utóbbi esetben hosszabb ideig tart egy-egy osztályozás, így amint az adott tudatállapot vizsgálására szánt módszernek (zenehallgatás, mantra meditáció, előadás/leckerészlet megtekintés) vége, megvizsgálom hogy az adott időtartam alatt mekkora százalékban adott helyes predikciót a neurális hálózat, és a parancssoros PC-oldali applikációban kiíratom ezt.

A pontosság növelésének érdekében egyrészt elkezdtem további adatokat felvenni a korábban felvett training setek méretének növelése érdekében. Immáron kizárólag magamról gyűjtöttem adatokat, otthonomban, ahol az elektromágneses interferencia mértéke kismértékű, a korábban leírt módokon végeztem az adatok gyűjtését.

3.6. A teljes rendszer elkészítésének folyamata

3.6.1. Elsődleges vizsgálatok, adatmozgatási módszertanok ellenőrzése

Eelkezdtem kialakítani a fenti flow-ban szereplő, valós idejű inferenciához szükséges összeköttetési rendszert a számítógép és a PYNQ-Z2 eszköz között. A legoptimálisabb, leggyorsabban és könnyedebben implementálható megoldásnak a websocket alapú kommunikációt találtam, így a Python nyelv websocket könyvtárának a segítségével egy tesztet hajtottam végre, ahol megmértem az adatátvitel sebességét először pár byte-nyi szöveges adat segítségével, a miliszekundumos nagyságrendű kapott érték bizakodásra adott okot, így kipróbáltam az inferencia során alkalmazandó egy-egy adategységgel. Az adatokat pickle segítségével szerializáltam, mivel a websockets könyvtár nem tartalmaz alapértelmezetten numpy array-ek küldésére alkalmas opciót, így a számítógép oldalon a loads() segítségével szerializáltam a küldendő adatokat, a PYNQ oldalon dumps() segítségével deszerializáltam az érkező adatokat, és összehasonlítottam a numpy array_equal-jának segítségével az eredeti adatokkal, amit eltároltam változóként korábban. A deszerializált adatokat a loads segítségével a PYNQ oldalon újra szerializáltam, majd visszaküldtem a számítógépnek, ahol a dumps segítségével deszerializáltam és ott is végrehajtottam az előbbi ellenőrzést. Ezen teszt is sikeres volt, a kapott érték még mindig miliszekundumokban volt mérhető, ami bőven megfelelő még akkor is, ha 8, vagy 16 méretű ablak van használva, így elkezdtem megírni azon kódokat, amik a teljes rendszer tesztelését lehetővé tették.

3.6.2. A számítógépes oldal működésének leírása

3.6.2.1. Irányítási eset

A számítógépes oldalon egy korábban a távirányítható autóval inferencia tesztelésére használt kódomat kombináltam a websocketes megoldással, a websocket async meghívandó függvényébe helyeztem el azon részeit a kódnak amik nem a BrainFlow könyvtár, vagy a Cyton beállításához szükségesek. Ezen részek állnak az OpenBCI EEG Cap-ből a Cyton Daisy-n keresztül érkező stream elindításából, az adatstream leállításából egy adott időtartam után, az adatok filterezéséből, az adatoknak a neurális hálózatnak megfelelő formátumába alakítása, elküldésük a fejlesztőkártyára, és a visszatérített prediktált osztály alapján soros porton elküldése egy irányítási információnak a távirányítható autó felé.

A websocket segítségével történő elküldése az adatoknak, a megfelelő formátumba alakítás utáni, pickle segítségével történő szerializációs lépés után történik meg.

3.6.2.2. Tudatállapotokat vizsgáló eset

Ebben az esetben is egy korábban inferenciára használt kódot vettem elő, és alakítottam át websockettel működőképessé, a fő különbségek a fenti esettel szemben hogy itt nincs szükség az információt soros porton továbbítani, így az inferencia kimenetét ami visszatér a számítógépre, elég eltárolni egy tömbben. Ezen kívül a mechanizmus ugyanaz, az adatok elkapása után végrehajtódik a filterezés, az adatok átkerülnek a PYNQ-Z2-re, visszatérnek miután végrehajtódott az inferencia, és ismertté válik a prediktált osztály.

3.6.3. A PYNQ-Z2 oldal működése

Ekkor az adatok átkerülnek a PYNQ-Z2-re, ahol szintén egy korábban inferenciára használt kódot alakítottam át websocketes működéssel kooperatívvá. A korábbiakban már többször leírt módon az xlnk és dma segítségével végrehajtom az inferenciát, a kimeneti array-en np.argmax segítségével megkapom a predikciót, majd ezen predikciót immáron már nem szerializálva, hanem egy számot a Python str() függvényével átalakítva visszaküldöm a számítógépnek, ahol a fentieknek megfelelően ezt az információt felhasználva generálódik egy irányítási parancs ami az RC autó aktuátorait fogja vezérelni. Ebben az esetben az egyetlen különbség az irányítási és a tudatállapotvizsgáló esetek között a lefoglalt tömb mérete a be és a kimeneti puffernek. Az előbbi esetén [64 * 16], míg az utóbbi esetén [512 * 16] a bemeneti, a kimeneti pedig 5, és 3, az osztályok számának megfelelően. A típusuk ugyanaz minden esetben, np.float32.

3.6.4. Reprodukálhatóság

Elkezdtem összeállítani a GitHub oldalon elérhető Tutorial részhez elérhető Jupyter Notebookokat, amelyek segítségével reprodukálhatóak a szakdolgozatomban létrejött eredmények. A korábban említett CERN PhD hallgató (Thea Klæboe Årrestad, GitHub:thaarres) repositoryját forkoltam, majd egy ígéretes branchének klónozásával egy új branchet hoztam létre openbci-on-pynq néven. Ezen branch tartalmáról, használatáról ismeretek alább, a Tutorial részben találhatóak.

3.6.5. A végleges rendszer elkészülte

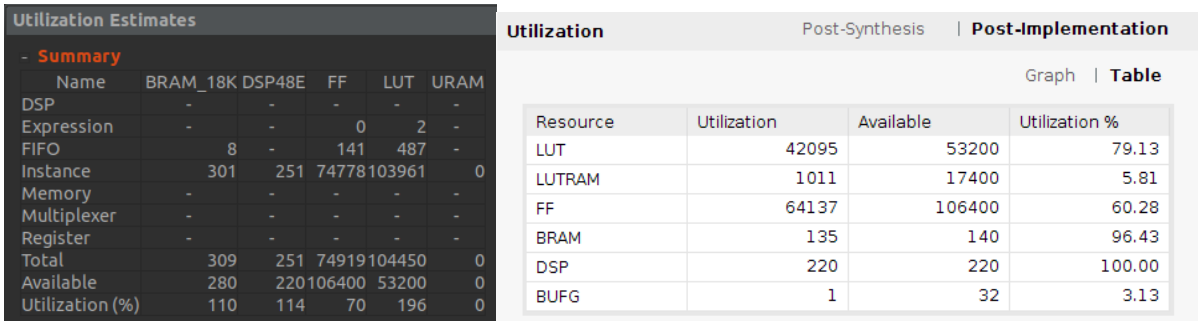
Az élő adatokon, valós időben FPGA-n történő inferenciának a validálására létrehozott rendszer tesztje sikeresen lezajlott, a PYNQ-Z2 oldalon egy adatpont inferenciája, ami a teszt során 64 x 16 méretű, 35ms-ot vett igénybe szerializációval együtt általában. A számítógép oldalon, ahol

zajlik az előfeldolgozás és továbbítódik a visszaérkezett predikció, ott 40ms-volt általában a szükséges időtartam, a teljes pipeline teljesítményét a számítógépen mért adat reprezentálja megfelelően, mivel a PYNQ-Z2-n történő előfeldolgozás megvalósíthatósága nem kétséges kellő mennyiségű hardveres erőforrás szabadon maradása esetén, azonban realizálása a szakdolgozat pillanatában meglévő tudásomat, kapacitásomat és időmet meghaladta.
Az irányításhoz használt neurális hálózat training adatainak a bővítése és kiegyenlítése után körülbelül 50%-os pontosságot kaptam a validation seten, ami messze áll az optimális eredménytől, élő tesztek során azonban 20% alatti pontosságot mutatott a rendszer, ami az 5 lehetséges osztály miatt jelzi hogy véletlenszerűen prediktál a rendszer, ezáltal teljességgel használhatatlanná vált.
A pontosság növelésének érdekében a neurális hálózatnak a konvolúciós filtereinek a számát növeltem, ezzel párhuzamosan a használt bitek számát is lecsökkentettem 10-re hogy beleférjen a PYNQ-Z2-be, ennek okán igaz hogy teljesen pontatlanok az eredmények, de nagyon gyorsak. A PYNQ-Z2-n az inferencia sebessége 600-700 μs környékén mozog. Annak fényében, hogy a tudatállapotokat vizsgáló esetben használható eredményeket értem el, az irányítási eset további fejlesztésével egy ponton felhagytam, hogy a tudatállapotokat vizsgáló esettel tudjak megfelelő mennyiségben foglalkozni.
A fő probléma az, hogy az OpenBCI Electrode Cap Kit eredendően egy EEG eszköz és nem EMG jelek felvételére van fizikailag kialakítva. Az OpenBCI-nak létezik EMG eszköze, gél alapú, egyszer használatos korong formájú elektróda, mely az online boltjukban EMG/ECG Foam Solid Gel Electrodes néven található meg, és elektromiográfiás, homloklebenyi EEG, és elektrokardiográfiás vizsgálatokat lehetséges vele végezni. Ezen eszköztípus az Informatikai Karon megtalálható, de a szakdolgozat írásának egy már olyan késői szakaszában szereztem erről tudomást, hogy idő hiányában nem kezdtem el foglalkozni vele.

Ezek után kettő dologgal foglalkoztam, az egyik a training set kiegészítéseként felvett többletadatok segítségével a neurális hálózat tovább tanítása, amely segítségével sikerült a pontosságot növelni nagymértékben, a tudatállapotokat vizsgáló esetben a neurális hálózatot körülbelül 3300 adatponton tanítottam be, minden osztálynak körülbelül 1100 adatpontot szánva. A rendszer minden osztály klasszifikálásának az esetén, a PYNQ-Z2-n futtatva élő tesztadatokon általában 75-80 százalékos pontosságot ért el. Ebben az esetben is megfigyelhető

volt a korábban már látott jelenség, miszerint a HLS jóval túlbecsüli az erőforrások kihasználtsági szintjét. Az alábbi képeken látható ez:

ábra 14
A végleges rendszer a HLS és a Vivado által megadott erőforrásigényei



Utilization Estimates

- **Summary**

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	2	-
FIFO	8	-	141	487	-
Instance	301	251	74778	103961	0
Memory	-	-	-	-	-
Multiplexer	-	-	-	-	-
Register	-	-	-	-	-
Total	309	251	74919	104450	0
Available	280	220	106400	53200	0
Utilization (%)	110	114	70	196	0

Utilization Post-Synthesis | **Post-Implementation**

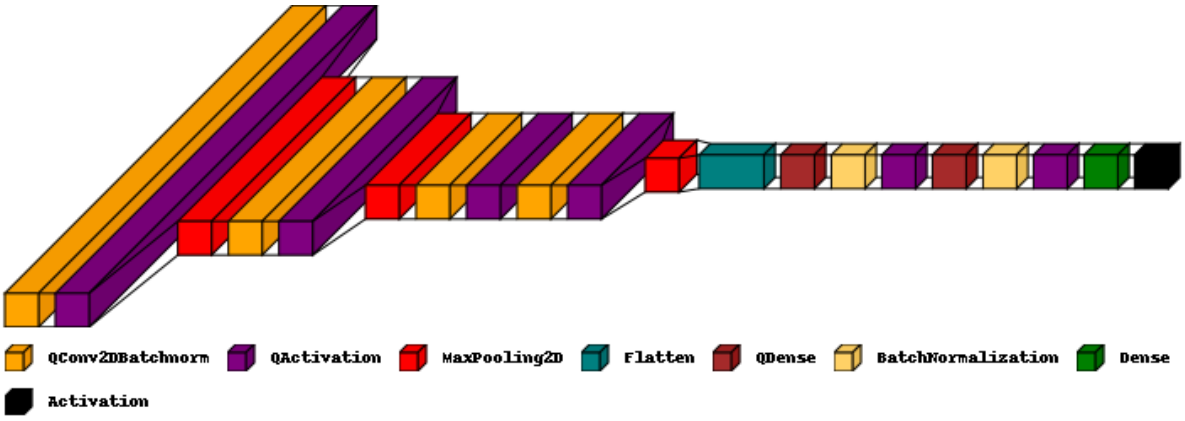
Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	42095	53200	79.13
LUTRAM	1011	17400	5.81
FF	64137	106400	60.28
BRAM	135	140	96.43
DSP	220	220	100.00
BUFG	1	32	3.13

A bal oldali a HLS, míg a jobb oldali a Vivado által megadott, implementáció utáni kihasználtsági szint. Mint látható a jobb oldali képen a DSP-k kihasználtsága 100%, ezt a ReuseFactor nevű változó rétegemlámto beállításával értem el, több implementációs iteráción keresztül jutottam el erre a pontra. A többi paraméter, például a BRAM vagy a LUT-ok kihasználtsága a használt neurális hálózat mélységétől, rétegeinek típusától, és a kvantálás során meghatározott bitszélességtől függ. Itt is a VivadoAccelerator opciót használtam az átalakítás során, azaz a Block Design a Vivadoban ugyanúgy néz ki mint a korábbi esetekben, a HLS IP tartalma változik csupán, ez azonban absztrahálva van, és csak a HLS IP C++ reprezentációjából, vagy a Vivado Design Suite-ban a használt erőforrások mennyiségéből látszik hogy egy másik designról van szó.

A végleges neurális hálózat struktúra:

ábra 15
A végleges, tudatállapotokat vizsgáló esetben a neurális hálózat struktúrája



A másik rész pedig egy összehasonlítási kísérlet megalapozása, ahol megvizsgáltam, hogy a neurális hálózatnak az inferenciaideje mekkora különböző platformokon, hardvereken futtatva.

3.7. Teljesítményösszehasonlítás

A következő opciókat vizsgáltam:

CPU – Ryzen 5 5600h, Ryzen 5 3600x

Beágyazott rendszer: PYNQ-Z2

A teszt kivitelezésének érdekében PC-n a korábban felvett, a test setből származó, a neurális hálózat által még nem látott példányokon futtattam le a neurális hálózatot reprezentáló Keras model predict() metódusát. A PYNQ eszközön a DMA IP be és kimenetének sendchannel.transfer metódusainak a meghívása között eltelt időtartamot figyeltem. Mind a két esetben azt néztem hogy az adatfeldolgozási, és adatmozgatási utasításokat leszámítva mennyi időt vett igénybe egy inferencia végrehajtása. Minden eszközön 100 darab inferencia végrehajtása után, a szükséges időtartamok átlagát kiszámolva kaptam meg az eszközre jellemző időtartamot. Az alábbi táblázatban elrendezve láthatóak az eredmények.

táblázat 1
Teljesítményösszehasonlítás

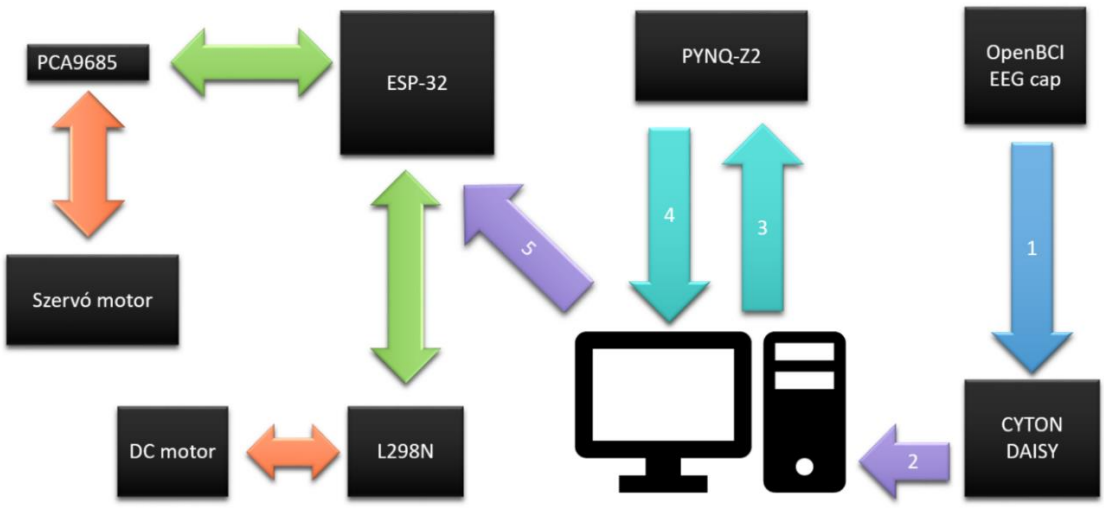
Használt eszköz	PYNQ-Z2	Ryzen 5 3600x	Ryzen 5 5600h
Inferenciaidő (s)	0,001862073	0.019515088	0.033071331

Az eredményekből látható, hogy az FPGA implementáció minimum 10-szer gyorsabb inferenciát tesz lehetővé, mint egy modern AMD processzor. Videokártyán, más beágyazott rendszeren vagy fejlesztőkártyán nem tudtam kipróbálni a rendszert.

3.8. A kész rendszer működési diagramjai irányítási, és tudatállapotfelismerési esetben

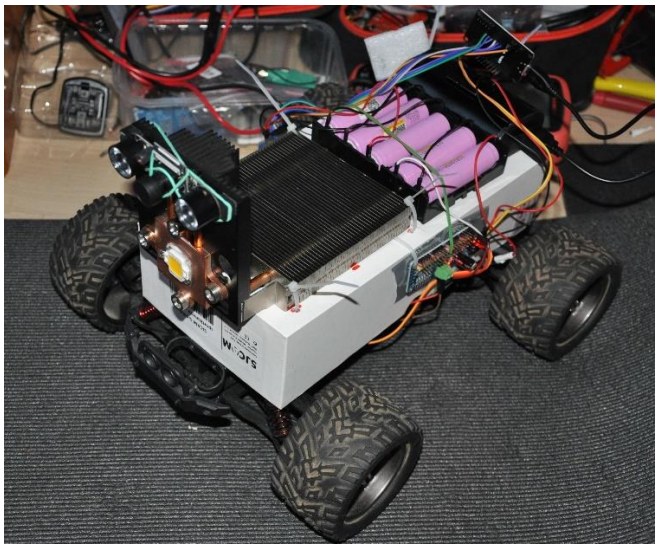
Az irányítási rendszer blokkdiagramja a következő:

ábra 16
A rendszer blokkdiagramja az irányítási esetben



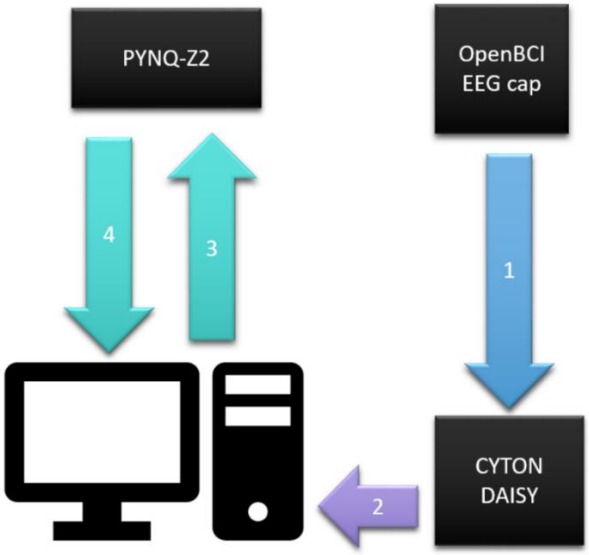
A fenti diagram elemeinek nagy részéről már volt szó, amit még nem fejtettem ki, az az RC autót irányító rendszer elemei, és egymáshoz viszonyulásuk. Amikor a számítógép továbbítja a predikciót az RC autót kontrolláló rendszernek, azt egy ESP32-es mikrokontroller dolgozza fel. A kanyarodásért egy darab szervómotor a felelős, ezt a gyári megoldás helyére szereltem be. A szervómotort egy PCA9685 16 csatornás szervómotorkontroller irányítja. A haladásért előre/hátra, és a fékezésért a gyárilag a modellben lévő DC motor a felelős, amelynek az irányítását egy L298N kétcsatornás DC motorkontrollerrel oldom meg. Az 5-ös számú nyíllal jelzett kommunikáció soros porton, egy USB kábelen keresztül zajlik, az ESP32 modul a tápfeszültséget is ezen keresztül kapja. A használt átalakított RC autóról egy kép:

ábra 17
Az átalakított RC autó



A tudatállapotvizsgáló rendszer blokkdiagramja:

ábra 18
A rendszer blokkdiagramja a tudatállapotokat vizsgáló esetben



Itt az irányítási részre szükség nem lévén a számítógépen gyűlik az információ, a futás végén egy százalékos értékkel lehet ellenőrizni, hogy mekkora mértékben adott helyes predikciót a neurális hálózat, és az oktatási anyagban szereplő rendszer úgy van kialakítva, hogy minden inferencia session egyben adatfelvételi is. Emiatt szükséges a futtatásnál megadni egy „–in_filename” paramétert, amely a kreálandó file nevét alapozza meg, ezen megadott névhez egy

időbélyeg és a file kiterjesztése jön hozzá. A szöveges file-ként, .txt formátumban tárolja el a program az inferencia/adatfelvétel során generálódott adatokat.

4. Oktatási anyag

Témavezető oktatómmal arra jutottunk, hogy érdemes lenne egy kezdetleges, tutorial jellegű, oktatási célokra később alkalmazható anyagot létrehozni a szakdolgozatom anyagából, hogy a projekt esetleges folytatása esetén könnyebben kezdjenek hozzá a hallgatók. Az EEG adatoknak az adatbiztonságilag kényes helyzete miatt a saját magamról felvett adataimat amit a neurális hálózat betanítására használtam nem teszem közzé, azonban a .h5 kiterjesztésű neurális hálózatot reprezentáló, Keras segítségével beolvasható és használható file-t igen, ahogyan a .bit és .hwh kiterjesztésű fileokat is amik a PYNQ-Z2-n futtatáshoz szükségesek, emellett a kész, átalakított HLS, és Vivado projekt is megtalálható a GitHub repositoryban. Ezen döntésem egyik foganatja, hogy a tutorial notebookoknak bizonyos részét (a hálózat betanítása) csak akkor lehetséges futtatni, ha a személy aki dolgozik a notebookon biztosít adatokat a dataset felépítéséhez.

Lépésekben írom le a teendőket amit követni kell a reprodukálhatóság érdekében.

Első lépés ha nincsen telepítve, egy Ubuntu 20.04 LTS telepítése a céleszközre, a háttértárolón ajánlott legalább 150 – 200 GB szabad hely hogy legyen, és a PYNQ-Z2 eszközre a PYNQ Image 2.5-ös verzióját szükséges feltelepíteni, ezt a telepítőfile a PYNQ-Z2 eszközhöz mellékelt SD-kártyára írásával oldhatjuk meg, ehhez én a Balena Etcher nevű programot használtam, de sok különböző alternatíva létezik. Az eszközspecifikus telepítendő file itt található meg: https://github.com/Xilinx/PYNQ/releases/tag/v2.5.

Második lépés feltelepíteni a Vivado HLS 2019.1-et, és a Vivado Design Suite 2019.1-et. Ehhez el kell navigálni a szakdolgozat írásának pillanatában a Xilinx cég oldalára, és a következő linken a 2019.1-es fület lenyitva, a “Vivado Design Suite - HLx Editions - 2019.1 Full Product Installation” alatt található Linux Self Extracting Web Installert le kell tölteni.

Letöltés regisztráció után a U.S. Government Export Approval form kitöltése után lehetséges letölteni a telepítőt, szükséges egy Xilinx fiókot létrehozni ha még nincsen. A letöltött file-t megkeressük például az Ubuntu Nautilus fájlböngészőjével, jobb klikket kell nyomni a fileon, Properties-re kattolva a Permissions fül alatt be kell pipálni az “Allow executing file as program” mellett található négyzetbe. Ezek után egy terminált kell nyitni a fájlt tartalmazó mappában, és a “*sudo ./fájl_neve*” paranccsal el kell indítani a telepítőt. A User Authentication alatt a bejelentkezéshez szükséges adatokat kell megadni. A “Select Edition to Install” alatt a Vivado HL WebPACK-ot kell kiválasztani. A következő ablakban a “Devices” pont alatt, az “SoC” pont alatt csak a “Zynq-7000" kell hogy bepipálva maradjon, ezáltal is kevesebb

tárhelyre lesz szükség. Minden további maradhat alapértelmezett. A telepítés rendszertől, internetelérétől függően órákat is igénybevehet.
A telepítés után szükséges kiadni egy könyvtártelepítési parancsot:
sudo apt-get install libtinfo5
Majd létre kell hozni a Vivado Design Suite-nak, és a Vivado HLS-nek egy-egy .sh kiterjesztésű filet, csak ezeknek a segítségével sikerült bugmentes működést elérnem. A fileok tartalma a következő Vivado Design Suite esetén:
cd /tools/Xilinx/Vivado/2019.1/bin
export LANG="en_US.UTF-8"
./vivado

És a következő Vivado HLS esetén:
cd /tools/Xilinx/Vivado/2019.1/bin
export LANG="en_US.UTF-8"
./vivado_hls

Célszerű a fileokat ésszerűen elnevezni, például Vivado_2019_1.sh, és Vivado_HLS_2019_1.sh, a fileokat bárhová el lehet helyezni, és ahol elhelyeztük, ott a terminálban a *bash Vivado_2019_1.sh*, vagy a *bash Vivado_HLS_2019_1.sh* parancs segítségével el tudjunk indítani a programokat.
Ezután el kell helyezni a .bashrc fileban egy sort, ehhez ki kell adni a *gedit ~/.bashrc* parancsot, majd a file tartalmához nem nyúlva, az alján egy új sorba a következőt kell elhelyezni:
export PATH="/tools/Xilinx/Vivado/2019.1/bin:$PATH"
Ctrl+S segítségével el kell menteni a változtatásokat, majd be kell zárni a gedit ablakát.
A saját teszt környezetemben, a laptopomon *sudo apt update*, majd *sudo apt upgrade* parancsok kiadása után a python 3.8 automatikusan feltelepítésre került. Ezek után le kell az Anaconda telepítőt tölteni a következő oldalról: https://repo.anaconda.com/archive/, és a következő nevűre van szükség: “Anaconda3-2020.11-Linux-x86_64.sh”. Ezen file letöltése után *bash Anaconda3-2020.11-Linux-x86_64.sh* parancs kiadásának segítségével el kell indítani a telepítőt.
A telepítés befejeztével a fentiekhez hasonlóan a .bashrc fájl végére kell írni egy újabb sort, ami a következő:

export PATH=”/home/username/anaconda3/bin:$PATH”

És a “username” helyett az Ubuntu rendszer felhasználónevét kell írni.

Ezek után a *sudo apt install git* paranccsal fel kell telepíteni a gitet, majd a GitHubon hosztolt, tutorial kódokat tartalmazó repom lokális klónját kell létrehozni a következő parancsokkal:

git clone https://github.com/kendermag/hls4ml-tutorial.git

cd hls4ml-tutorial/

git checkout openbci-on-pynq

Ezek után a következő parancsokat szükséges futtatni:

conda env create –f environment.yml

conda init - ezután szükséges egy új terminált indítani

conda activate hls4ml-pynq-tutorial

jupyter notebook

A következő lépés a part8_openbci_control_pynq.ipynb nevű Jupyter Notebookot megnyitni, amelynek a lépéseit követve reprodukálható az általam készített flow. A shift+enter gomb megnyomásával az adott cellát lehetséges futtatni. Enterrel lehetséges a kijelölt cellán szerkesztő módba lépni, Esc gombbal kilépni a szerkesztő módból. Ha az importokat tartalmazó cella futtatásakor "ImportError: No module named" hibaüzenettel találkozunk, akkor egy olyan terminálban, ahol az anaconda környezet aktiválva van, ott az adott modult *pip3 install modulnév* paranccsal feltelepíthetjük.

Amint lefutottak sorrendben fentről lefelé a cellák, el kell indítani a Vivado Design Suite-t, és a megjelenő ablakban a Quick Start-nál az Open Project-re kell kattintani. El kell navigálni a GitHubról klónozott repository fájlrendszerben elfoglalt helyére, és a TUDATALLAPOT_QUANTIZED_VER0_312_FULL mappán belül a myproject_vivado_accelerator mappában található project_1.xpr file-t kell megnyitni. Ha megnyílt, akkor a baloldali menüsáv IP INTEGRATOR lenyitható fülnél található Open Block Design-ra kattintva vizuálisan szemrevételezhetjük a hls4ml által készített designt.

A PYNQ-Z2-n futtathatáshoz szükséges exportálnunk a hardvert, amit a következőképpen tehetünk meg:

A legfelső vízszintes menüsáv “File”-pontjára kattintva, az “Export” almenüpont “Export Hardware” opciójára kattintunk. Kipipáljuk az “Include Bitstream” opciót, és az “Export to” legördíthető menüből a choose location segítségével kiválasztunk egy helyet neki. Ezek után az “OK” gombra kattintva végbemegy a .hdf formátumú file generálása.

A generált file-t fájlkezelőben kikeresve, azon duplán kattintva megnyílik archive managerben, két filera lesz innen szükség, a design_1_wrapper.bit-re, és a design_1.hwh-ra. Ezeket egérrel áthúzhatjuk a Nautilus fájlkezelő ablakába, ezáltal kicsomagolva létrehozunk másolatokat a fileokról. A design_1.hwh-t át kell nevezni design_1_wrapper.hwh-ra, mivel a PYNQ-Z2 Overlay kezelő rendszere akkor lesz képes a hardveres erőforrásokat megfelelően tájolni a bitfilehoz, ha a kiterjesztésen kívüli név megegyezik. Ezen .bit, és .hwh kiterjesztésű fileokat immáron átmásolhatjuk a PYNQ-Z2 eszközre, célszerű vagy több mappát létrehozni a különböző bitfileoknak, vagy magukat a bitfileokat az alkalmazásnak megfelelően elnevezni. Én az előbbit preferálom, mert így könnyedebben lehetséges járulékos fileokat (például képek, chartok) elhelyezni projekthez kötődően. A PYNQ-Z2 eszközön futtatandó Jupyter notebook a GitHubról klónozott repositoryban PYNQ_openbci_inference.ipynb néven szerepel, ezt a filet fel kell tölteni a PYNQ-Z2-re.

Ezek után fel kell helyezni az OpenBCI eszközt, érdemes mindig teljesen, vagy majdnem teljesen feltöltött akkumulátorral használni, nálam gyakran előjött a probléma hogy eldobált csomagokat, elvesztette gyakran a jelet ha alacsony volt az akkumulátor töltöttsége. Először kell a számítógépbe/laptopba csatlakoztatni az USB dongle-t, és ezután kell áram alá helyezni a Cyton Daisy boardot.

A PYNQ-Z2-n található notebookot szükséges először futtatni.

Egy bug miatt egy trükkös megoldást kell alkalmazni. Bizonyos cellák meg vannak jelölve, és amely cella felé oda van írva hogy ezen cella futtatása közben kell interruptolni a kernelt, ezen cella futtatásakor a Jupyter Notebook menüsávjában felül, a Run gomb mellett jobbra található Stop (fekete kitöltött négyzet) gombot szükséges megnyomni. Ezek után lehet haladni a következő cellákkal sorrendben lefelé. A cellák futtatása után el kell indítani a számítógépen található, a PYNQ-Z2-vel Python websocket segítségével kommunikáló Python programot, ami az adatok az OpenBCI capből való elkapását, az adatok előfeldolgozását, a neurális hálónak megfelelő formátumba való átalakítását, és a PYNQ-Z2 felé továbbítását végzi. Ezen filenak a neve tudatallapot_inferencia.py.

Ha ezen file futtatása során az UNABLE_TO_OPEN_PORT_ERROR:2 jelenne meg, ekkor ki kell adni a következő parancsot:

sudo chmod 777 /dev/ttyUSB0

Ez az oktatási anyag a szakdolgozatomnak a tudatállapotokat vizsgáló részének a reprodukálhatóságát biztosítja, a PYNQ-Z2-től visszakapott predikciót és a predikciók

pontosságát a számítógépen terminálablakba kiíratva. A GitHub repositoryban a DATA_ACQUISITION mappában találhatóak az irányítási adatok felvételére használható fileok.

Az esetlegesen felmerülő problémákra 2022 októberétől tudok reagálni, pull requesteket, megjegyzéseket akkor tudok majd ellenőrizni.

5. Összefoglalás

A szakdolgozatom során megkíséreltem összeállítani egy rendszert, ami a legoptimálisabb esetben egy OpenBCI EEG Electrode Cap Kitből, az egyszerű agy-számítógép interface eszközből, egy PYNQ-Z2 eszközből, és a predikciót felhasználó eszközből állt volna. A kivitelezés során kiderült, hogy a fennálló időkeret szűkössége, a téma sokrétűsége és komplexitása miatt kompromisszumokat kell kötnöm, és ki kellett egészítenem egy általános célú számítógéppel a pipeline-t. Az adatok előfeldolgozását bízom rá erre a bónusz komponensre.

A kitűzött célok közül véleményem szerint a tudatállapotokat vizsgáló eset kitűzött célját teljesítettem, sikerült kiválasztani egy megfelelő módszert neurális hálózatoknak az átalakítására, amely által újrakonfigurálható hardveren lesz az képes futni.

Az irányítási eset célkitűzéseit nem tudtam teljesíteni, mivel mint időközben kiderült, nem erre lett kitalálva az OpenBCI EEG Cap eszköz.

A neurális hálózatok átalakításához használt framework alacsony belépési tudásszintet követel meg, ha az összes hozzá szükséges egyéb eszközzel már volt dolga a felhasználónak, könnyedén működésbe tudja hozni, azonban még egy, a területtel csak ismerkedő személy számára sem jelenthet komoly akadályt a fent leírt tutorialnak a lépéseinek a lekövetése.

Ezen, hls4ml nevű könyvtár képességei túlmutatnak ezen szakdolgozaton, nagyobb méretű FPGA-k esetén mélyebb, és az erőforrások számának függvényében akár gyorsabban futó neurális hálózatok hozhatóak létre, az implementációs, tanítási és átalakítási fázisokat leszámítva gyorsan iterálva a dizájnok verzióin. Az átalakítás egyszerűsége és a fejlesztés sebessége a kézzel írt RTL-hez, vagy a kézzel írt C++ kódhoz nagyságrendekkel gyorsabb, az automatikus ellenőrző lépések hiánya okozhat problémát, de még ezzel együtt is megéri használni véleményem szerint. A generált C++ kód testreszabásával pedig lehetséges volna az általam prezentáltnál jóval nagyobb mértékben a hardverspecifikációkhoz, és az adott rendszer követelményeihez igazítani a dizájnt, a fejlesztési idő növekedésének az árán. A felvetett elvet, azaz

Az átalakított neurális hálózat inferenciasebessége, és a teljes rendszer sebessége elfogadható, processzoron futtatott inferenciánál minimum 10x gyorsabb a működőképes hálózat esetében, 14 bit használata esetén és több mint 20x-os sebességnövekedés volt tapasztalható 10 bit használatával, habár ezen rendszer nem adott értékelhető predikciókat. A használt OpenBCI

eszközzel csak alacsony akkumulátorszint, és magas elektromágneses térben történő közben akadtak problémák.

A PYNQ keretrendszerrel teljesen elégedett vagyok, a Jupyteres magas szintű környezet könnyen kezelhető, gyors prototípustervezést tesz lehetővé.

Az irányítási eset továbbfejlesztésére egy lehetőség lehetne a bőrre ragasztható EMG felvételre optimalizált elektródák használata az arc különböző pontjai felragasztva. Véleményem szerint működőképes rendszereket lenne lehetséges összeállítani, mivel ezen elektródák az EEG sisakkal ellentétben célspecifikusan EMG jelek felvételére lettek kifejlesztve, a tanszéken elérhetőek ezen eszközök, és a tanszék tulajdonában lévő Cyton Daisy támogatja a használatát ezen típusú elektródáknak.

A tudatállapotokat vizsgáló eset továbbfejlesztésére opció lehet vagy többfajta tudatállapot vizsgálata, vagy több emberről történő adatfelvétellel a neurális hálózat nagyobb generalizáló képességének elérése.

6. Referenciák

[1] Krucoff, M. O., Rahimpour, S., Slutzky, M. W., Edgerton, V. R., & Turner, D. A. (2016). Enhancing Nervous System Recovery through Neurobiologics, Neural Interface Training, and Neurorehabilitation. *Frontiers in neuroscience*, *10*, 584. https://doi.org/10.3389/fnins.2016.00584 Megtekintve 2021 09 10

[2] OpenBCI Shop EEG Electrode Cap Kit - https://shop.openbci.com/collections/frontpage/products/openbci-eeg-electrocap Megtekintve 2021 08 27

[3] GitHub – hands on PYNQ workshop training material https://github.com/Xilinx/PYNQ_Workshop Megtekintve 2021 09 14

[4] GitHub Hello World type application for Xilinx PYNQ framework https://github.com/Xilinx/PYNQ-HelloWorld/tree/v2.5 Megtekintve 2021 09 10

[5] PYNQ Read the Docs PYNQ-Z2 Setup Guide https://pynq.readthedocs.io/en/v2.5/getting_started/pynq_z2_setup.html Megtekintve 2021 09 03

[6] hls4ml Read the Docs - Concepts https://fastmachinelearning.org/hls4ml/concepts.html Megtekintve 2021 09 09

[7] GitHub hls4ml package repository https://github.com/fastmachinelearning/hls4ml Megtekintve 2021 09 07

[8] Pynq Read the Docs – PYNQ Introduction https://pynq.readthedocs.io/en/v2.5/ Megtekintve 2021 09 10

[9] Aurélien Géron Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition, Second Release, O'Reilly Media, Inc. ISBN: 9781492032649, Chapter 1: The Machine Learning Landscape 2. oldal

[10] Aurélien Géron Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition, Second Release, O'Reilly Media, Inc. ISBN: 9781492032649, Chapter 1: The Machine Learning Landscape 6-10. oldal

[11] Xcell - The Quarterly Journal for Programmable Logic Users Issue 32 Second Quarter 1999 https://www.xilinx.com/publications/archives/xcell/Xcell32.pdf Letöltve 2021 09 07

[12] Xilinc.com - Devices – FPGAs & 3D ICs – What is an FPGA? https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html Megtekintve 2021 09 10

[13] All About Circuits – What Is an FPGA? An Introduction to Programmable Logic https://www.allaboutcircuits.com/technical-articles/what-is-an-fpga-introduction-to-programmable-logic-fpga-vs-microcontroller/ Megtekintve 2021 09 06

[14] Xilinx Products - ZYNQ https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html https://www.xilinx.com/content/dam/xilinx/support/documentation/product-briefs/zynq-7000-product-brief.pdf Megtekintve 2021 09 10

[15] Xilinx – Zynq-7000 SoC Product Selection Guide https://www.xilinx.com/support/documentation/selection-guides/zynq-7000-product-selection-guide.pdf 2. oldal Megtekintve 2021 09 04

[16] 15

[17] Xilinx Products - ZYNQ – Product Advantages - Block Diagram https://www.xilinx.com/content/dam/xilinx/imgs/block-diagrams/zynq-mp-core-dual.png Letöltve 2021 09 01

[18] Pynq.io – What is PYNQ http://www.pynq.io/ Megtekintve 2021 09 07

[19] Pynq Read the Docs - PYNQ Introduction https://pynq.readthedocs.io/en/v2.5/ Megtekintve 2021 09 07

[20] Tul PYNQ-Z2 Product Specification https://www.tul.com.tw/images/PYNQ-Z2_PA_v2_pp_20201209_STD.pdf
1., 2. oldalak. Megtekintve 2021 09 04

[21] Tul Products – FPGA – PYNQ-Z2 https://www.tul.com.tw/images/01_PYNQ-Z2.jpg Letöltve 2021 09 04

[22] PYNQ Read the docs PYNQ-Z2 Block Diagram
https://pynq.readthedocs.io/en/v2.5/_images/pynqz2_base_overlay.png Letöltve 2021 10 07

[23] Fastmachinelearning.org/hls4ml autogenerated read the docs - Concepts -
https://fastmachinelearning.org/hls4ml/concepts.html Megtekintve 2021 09 10

[24] Sioni Summers – hls4ml tutorial 4th IML Workshop, 19th October 2020
https://indico.cern.ch/event/852553/sessions/370141/attachments/2125512/3578488/hls4ml_tutorial_iml.pdf
Megtekintve 2021 09 06

[26] Sioni Summers – hls4ml tutorial 4th IML Workshop, 19th October 2020
https://indico.cern.ch/event/852553/sessions/370141/attachments/2125512/3578488/hls4ml_tutorial_iml.pdf 25. oldal, Part 2: Advanced Configuration – Efficient NN design Megtekintve 2021 09 07

[27] Sioni Summers – hls4ml tutorial 4th IML Workshop, 19th October 2020
https://indico.cern.ch/event/852553/sessions/370141/attachments/2125512/3578488/hls4ml_tutorial_iml.pdf 26, 27. oldal, Part 2: Advanced Configuration – Efficient NN design Megtekintve 2021 09 07

[28] Sioni Summers – hls4ml tutorial 4th IML Workshop, 19th October 2020
https://indico.cern.ch/event/852553/sessions/370141/attachments/2125512/3578488/hls4ml_tutorial_iml.pdf 32, 33, 34. oldal, Part 3: Compression Megtekintve 2021 09 07

[29] Sioni Summers – hls4ml tutorial 4th IML Workshop, 19th October 2020
https://indico.cern.ch/event/852553/sessions/370141/attachments/2125512/3578488/hls4ml_tutorial_iml.pdf 36, 37. oldal, Part 4: Quantization Megtekintve 2021 09 07

[30] Aurélien Géron Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition, Second Release, O'Reilly Media, Inc. ISBN: 9781492032649, Chapter 10: Introduction to Artificial Neural Networks with Keras 283, 284, 285, 286. oldalak

[31] Vincent Dumoulin, Francesco Visin - A guide to convolution arithmetic for deep learning (BibTeX)
Megtekintve 2021 09 05

[32] Aurélien Géron Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition, Second Release, O'Reilly Media, Inc. ISBN: 9781492032649, Chapter 14: Deep Computer Vision Using Convolutional Neural Networks 448. oldal

[33] Large Scale Visual Recognition Challenge 2012 (ILSVRC2012) https://image-net.org/challenges/LSVRC/2012/results.html Megtekintve 2021 09 01

[34] Google Research – Publications: TensorFlow A system for large-scale machine learning 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), USENIX Association (2016), pp. 265-283, https://research.google/pubs/pub45381/ Megtekintve 2021 09 02

[35] TensorFlow Core v2.7.0 API Documentation TensorFlow C++ API Reference
https://www.tensorflow.org/api_docs/cc/ Megtekintve 2021 09 01

[36] Keras – About Keras https://keras.io/about/ Megtekintve 2021 09 02

[37] Britton JW, Frey LC, Hopp JLet al., Electroencephalography (EEG): An Introductory Text and Atlas of

Normal and Abnormal Findings in Adults, Children, and Infants https://www.ncbi.nlm.nih.gov/books/NBK390346/ Megtekintve 2021 09 03

[38] University of Pennsylvania – Perelman School of Medicine - Electroencephalogram (EEG)/Electromyography (EMG) https://www.med.upenn.edu/brainstimcenter/electroencephalogram-eeg/electromyography-emg.html Megtekintve 2019 09 03

[39] Journal of Neurology, Neurosurgery, and Psychiatry 2005;76(Suppl II):ii32–ii35. doi: 10.1136/jnnp.2005.069211 https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1765694/pdf/v076p0ii32.pdf Megtekintve 2021 09 03

[40] Shih, J. J., Krusienski, D. J., & Wolpaw, J. R. (2012). Brain-computer interfaces in medicine. *Mayo Clinic proceedings, 87*(3), 268–279. https://doi.org/10.1016/j.mayocp.2011.12.008 Megtekintve 2021 09 04

[41] Xilinx Inc. Vivado Design Suite Tutorial – High-Level Synthesis UG871 (v2019.1) May 22, 2019 https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug871-vivado-high-level-synthesis-tutorial.pdf - Letöltve 2021 08 20

[42] Mentor Graphics Corporation – High-Level Synthesis Blue Book – https://www.eet.bme.hu/~timar/data/hls_bluebook_uv.pdf Megtekintve 2021 09 10

[43] Semiconductor Engineering – Dave Pursley – What's The Real Benefit of High Level Synthesis https://semiengineering.com/whats-the-real-benefit-of-high-level-synthesis/ - Megtekintve 2021 09 11

[44] MathWorks HDL Coder - https://www.mathworks.com/products/hdl-coder.html Megtekintve 2021 09 11

[45] Ministry of External Affairs – Government of India – Dr. Ishwar V. Basaravaddi – Yoga: Its Origin, History and Developement https://www.mea.gov.in/search-result.htm?25096 Megtekintve 2021 09 11

[46] U.S. Department of Health and Human Services – National Institutes of Health – National Center for Complementary and Integrative Health – Meditation: In Depth https://www.nccih.nih.gov/health/meditation-in-depth Megtekintve 2021 09 05

[47] U.S. Department of Health and Human Services – National Institutes of Health – National Center for Complementary and Integrative Health – Yoga: What You Need To Know https://www.nccih.nih.gov/health/yoga-what-you-need-to-know Megtekintve 2021 09 06

[48] Farkas Attila Márton és Tenigl-Takács László - Patandzsali: Az igázás szövétneke(Jóga-Szútra) A Tan Kapuja Buddhista Főiskola, 1994. https://mek.oszk.hu/00500/00586/html/ - Megtekintve 2021 09 02

[49] GitHub - thaarres hls4ml fork https://github.com/thaarres/hls4ml-tutorial/tree/master Klónozva 2021 10 10