UNIVERSITATEA „SAPIENTIA” DIN CLUJ-NAPOCA

FACULTATEA DE ȘTIINȚE TEHNICE ȘI UMANISTE,

TÎRGU-MUREȘ

SPECIALIZAREA CALCULATOARE

Framework pentru implementarea rețelelor neuronale artificiale pe circuite digitale reconfigurabile.

PROIECT DE DIPLOMĂ

Coordonator științific:
Conf. dr. ing. Brassai Sándor Tihamér

Absolvent:
Hammas Attila

2021

UNIVERSITATEA "SAPIENTIA" din CLUJ-NAPOCA **Viza facultății:**
Facultatea de Științe Tehnice și Umaniste din Târgu Mureș
Specializarea: **Calculatoare**

LUCRARE DE DIPLOMĂ

Coordonator științific:	Candidat: Hammas Attila
Conf. dr. ing. Brassai Sándor Tihamér	Anul absolvirii: **2021**

a) **Tema lucrării de licență:**

-Framework pentru implementarea rețelelor neuronale artificiale pe circuite digitale reconfigurabile

b) Problemele principale tratate:

-Studiu bibliografic privind framework-uri pentru implementarea rețelelor neuronale artificiale-
Implementarea funcțiilor de activare în hardware

c) Desene obligatorii:

- Arhitectura sistemului pentru generarea modelului hardware
- Schema bloc a procesului de generare a modelului în hardware
- Diagrame UML privind software-ul realizat.
- Măsurători realizate pentru validarea rețelelor neuronale implementate în hardware și compararea diferitelor configurații

d) Softuri obligatorii:

- Soft pentru implementarea rețelelor neuronale într-un program de nivel înalt (Python)
- Soft pentru implementarea rețelelor neuronale în C/C++
- Soft pentru testarea rețelelor neuronale implementate în circuitul FPGA

Bibliografia recomandată:

[1] I. Tsmots, O. Skorokhoda and V. Rabyk, "Hardware implementation of sigmoid activation functions using FPGA," in 2019 IEEE 15th International Conference on the Experience of Designing and Application of CAD Systems (CADSM), IEEEE, 2019, pp. 34-38.

[2] D. H. Noronha, B. Salehpour and S. J. Wilton, "LeFlow: Enabling flexible FPGA high-level synthesis of tensorflow deep neural networks," ArXiv 1807.05317, 2018.

[3] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang and J. Cong, "FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates," in 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), IEEE, 2017, pp. 152-159.

f) Termene obligatorii de consultații: săptămânal

g) Locul și durata practicii: Universitatea Sapientia, Facultatea de Științe Tehnice și Umaniste din Târgu Mureș

Primit tema la data de: 31.03.2020
Termen de predare: 06.07.2021

Semnătura Director Departament **Semnătura coordonatorului**

Semnătura responsabilului programului de studiu **Semnătura candidatului**

Model tip a.

Declarație

Subsemnatul **Hammas Attila**, absolvent al specializării **Calculatoare**, promoția **2021** cunoscând prevederile Legii Educației Naționale 1/2011 și a Codului de etică și deontologie profesională a Universității Sapientia cu privire la furt intelectual declar pe propria răspundere că prezenta lucrare de licență/proiect de diplomă/disertație se bazează pe activitatea personală, cercetarea/proiectarea este efectuată de mine, informațiile și datele preluate din literatura de specialitate sunt citate în mod corespunzător.

Localitatea, Târgu Mureș

Data: 05.07.2021

Absolvent

Semnătura……

Framework pentru implementarea rețelelor neuronale artificiale pe circuite digitale reconfigurabile

Extras

Rețelele neuronale artificiale (ANN) sunt utilizate pe scară largă în rezolvarea problemelor, cum ar fi: prelucrarea imaginilor, extragerea datelor, sarcini de clasificare. Pentru a reduce latența rețelelor neuronale, este recomandabil să se utilizeze acceleratoare hardware. Au existat multe astfel de abordări hardware în ultimul deceniu. O opțiune este proiectarea unui accelerator hardware bazată pe circuite digitale reconfigurabile (FPGA). Principalele avantaje ale sistemelor bazate pe FPGA sunt paralelismul, consumul redus de energie și flexibilitatea. Cu toate acestea, proiectarea și implementarea rețelelor neuronale pe sisteme bazate pe FPGA este un proces dificil și de lungă durată care poate fi realizată doar de profesioniști cu cunoștințe și experiență potrivite în proiectare hardware. O soluție la această problemă poate fi găsită în cadrul prezentat în lucrarea de diplomă, care este capabilă să ușureze procesul de implementare a rețelelor neuronale pe circuite digitale reconfigurabile. Scopul cadrului este de a accelera și a facilita proiectarea și implementarea rețelelor neuronale care urmează să fie construite pe FPGA, pentru a accelera calculul rețelelor neuronale și pentru a studia posibilitățile de optimizare în ceea ce privește resursele hardware utilizate și performanța. Cadrul implementat în Python generează cod C ++ pentru rețelele neuronale pe baza unui șablon. Acest cod HLS este optimizat folosind directive. Printre altele, lucrarea de diplomă prezintă o comparație a diferitelor aproximări ale funcțiilor de activare din următoarele puncte de vedere: acuratețe, resurse necesare și timpul de procesare. În cele din urmă, este prezentat un exemplu de utilizare a modulului de Rețea neuronală artificială generată de framework. Rețeaua neuronală care rulează pe hardware a fost integrată într-un sistem pe care noi l-am dezvoltat, testat și comparat cu rezultatele date de modelele implementate în Python.

Cuvinte cheie: Framework pentru implementarea rețelele neuronale artificiale, HLS, MLP, RBF, FPGA

SAPIENTIA ERDÉLYI MAGYAR TUDOMÁNYEGYETEM
MAROSVÁSÁRHELYI KAR
SZÁMÍTÁSTECHNIKA SZAK

Keretrendszer neuronhálók FPGA alapú megvalósítására.

DIPLOMADOLGOZAT

Témavezető:
dr. Brassai Sándor Tihamér,
egyetemi előadótanár

Végzős hallgató:
Hammas Attila

2021

Kivonat

A neuronhálók széles körben alkalmazhatók, mint például: képfeldolgozás, képalkotás, osztályozási feladatok. A neuronhálók számítási késleltetésének csökkentése érdekében érdemes hardveres gyorsítókat alkalmazni. Az egyik lehetőség hardveres gyorsító kialakítására az újrakonfigurálható digitális áramkörök (FPGA). Az FPGA alapú rendszerek legfőbb előnyei, hogy könnyedén lehet rajtuk párhuzamosítani, alacsony az energiafogyasztásuk és a rugalmasság, mert új modell esetén újra lehet konfigurálni. Azonban a neuronhálók tervezése és megvalósítása FPGA alapú rendszereken egy nehéz és hosszadalmas folyamat, amit csak megfelelő tudással és tapasztalattal rendelkező szakemberek tudnak elvégezni. Erre a problémára nyújthat megoldást a dolgozatban bemutatott keretrendszer, amely képes neuronhálók újrakonfigurálható digitális áramkörökön való megvalósítását megkönnyíteni. A keretrendszer célja az FPGA-n elkészítendő neuronhálók tervezési és megvalósítási folyamatának felgyorsítása, megkönnyítése. a neuronhálók számításának gyorsítása, optimalizálási lehetőségek tanulmányozása méret és teljesítmény szempontjából. A Pythonban megvalósított keretrendszer neuronhálók C++ kódját generálja egy sablon alapján. Ez a HLS kód direktívák alkalmazásával van optimalizálva. A dolgozat többek között még bemutatja a nemlineáris függvények (bázis- és aktivációs függvények) különböző approximációjának az összehasonlítását pontosság, erőforrás szükséglett és késleltetés szempontjából. Végül a kigenerált hálómodul felhasználása van bemutatva. A hardveren futó neuronhálót egy általunk kialakított rendszerbe integrálva teszteltük és összehasonlítottuk a Pythonban megvalósított modellekkel.

Kulcsszavak: keretrendszer neuronhálókra, HLS, MLP, RBF, FPGA

Abstract

Artificial neural networks (ANN) are widely used in solving problems like image processing, data mining, or classification. Hardware accelerators are used for increasing the performance and efficiency of neural networks. An option for implementing such an accelerator is an FPGA-based system, although developing neural networks for FPGAs is very time-consuming and requires professionals to do it. In this article, we try to tackle this problem by creating a framework that should speed up the process. At the same time, we will take a look at some efficiency optimization and speed-up options as well. The framework is written in Python and generates a C++ code whit HLS directives. This code can be compiled by Vivado HLS into a hardware descriptive language and packaged as an IP. The Vivado tool can generate a bit file that can be uploaded onto the FPGA device.

Among other things, the dissertation presents a comparison of different approximations of nonlinear transformations (base functions and activation functions) in terms of accuracy, required resource, and delay needed for evaluating the transformation. The generated neural network module was integrated into a system, that we developed. Using that system, we tested the neural network module and compared it with the models implemented in Python.

Keywords: framework for neural networks, HLS, MLP, RBF, FPGA

Tartalomjegyzék

Ábrák jegyzéke

Táblázatok jegyzéke

1. Bevezető

Az emberiség számára már rég óta cél, hogy megértse, hogy hogyan is működünk, mint intelligens és kognitív lények. Napjainkba is számos sci-fi-könyv és film említ vagy mutat be olyan gépeket, amelyek rendkívüli intelligenciával bírnak. Ez is arra utal, hogy az emberben megvan az a törekvés, hogy egy olyan rendszert alkoson, ami rendelkezik egy általános intelligenciával. Ennek a vágynak a köszönhetően tudunk beszélni a mesterséges intelligenciáról, mint fogalom.

Meg lehet próbálni kvantitatív módon megfogalmazni azt, hogy a gerincesek agya és a digitális gépek menyire jók vagy rosszak a számítási problémák megoldásában. Viszont az elött, hogy egy ilyen összehasonlításra képesek lennénk először kell definiáljuk, hogy mit értünk belső sebeségként egy agy, illetve egy számítógép esetében. Figyelemben kell vegyük, hogy egy processzor belső órajele 10^5 nagyságrendekkel gyorsabb, azaz nagyságrendekkel több alapműveletet végez el, mint a mi ideg hálózatunk hoz egy elemi döntést [1]. Ha a kérdés megfelelése érdekében azt a szempontot vesszük figyelembe, hogy melyik rendszer képes több aritmetikai műveletet elvégezni, akkor arra a következtetésre jutunk, hogy a számítógép az sokkal hatékonyabb. Másrészt, ha komplexebb feladatok megoldásban mérjük össze a két rendszert, mint például arcfelismerés vagy egy nyelv dekódolása, akkor arra a következtetésre jutunk, hogy az agy az 10^8 nagyságrenddel hatékonyabb, mint egy számítógép [1]. Napjainkban viszont a számítógépet egyre többször van rászorulva arra, hogy olyan feladatokra keressen megoldást, amelyeket korában csak a az emberek voltak képesek megoldani. Ekkor felmerül a kérdés, hogy tudunk-e olyan elektronikus architektúrákat készíteni, amelyek jobban hasonlítanak a neurobiológiai architektúrákra, és amelyek jobbak, mint a hagyományos architektúrák a homályos problémák kezelésére, mint például az arcfelismerés vagy a természetes nyelv dekódolása. A legtöbb típusú mesterséges neuronháló szerkezete nagyban hasonlít a gerincesek idegrendszeréhez és azok több tulajdonságát is modellezik. Például a neuronok szinapszisokon keresztül kommunikálnak egymással.

A mesterséges intelligencia (angolul: Artificial Intelligence, rövidítése: AI) területén, új előre haladások lehetővé teszik, hogy a mesterséges neuronhálók (angolul: Artificial Neural Network, rövidítése: ANN) megtanuljanak komplex problémákra megoldást találni elfogadható idő alatt [2]. A számítógépes látás, beszédfelismerés és sok más területen is forradalmi változásokat hoztak a mély tanulású hálózatok. Eleinte legfőképpen adatközpontokban alkalmazták viszont a későbbiekben telefonokban, robotikában és külön féle beágyazott rendszerekben is erőszeretettel alkalmazzák. A mély tanulású hálózatoknak magas a számítás és memória igénye, valamint sok műveletet lehet párhuzamosan is végrehajtani így hatékonyan lehet őket grafikus gyorsitokon (GPU) fútatni. Ugyanakkor elkezdtek dedikált hardveres gyorsítókat kifejleszteni. Grafikus

gyorsítók körében vannak olyanok, amik főként adatfeldolgozásra és mesterséges intelligencia alkalmazásokra voltak tervezve. Egy másik lehetőség az FPGA-s gyorsítok, melyek nagy népszerűséget szereztek rugalmasságuk, teljesítményük és kis energiafogyasztásuk miatt. Azonban az FPGA alapú gyorsítok programozása sok tapasztalatot és szakértelmet igényel és nagyon időigényes folyamat. Ezt a folyamatot szeretnénk leegyszerűsíteni egy keretrendszer segítségével. A projektet társammal, Bustya Balázzsal, közösen kezdtük el az Accenture cég által meghirdetett Accenture Student Research Scholarship 2020 pályázat keretén belül. A pályázat során többrétegű Perceptron neuronhálókat implementáltunk FPGA áramkörökön. Két technológiával is próbálkoztunk: VHDL (angolul: Very High Speed Integrated Circuit Hardware Description Language) és HLS (angolul: High-Level Synthesis). Azt tapasztaltuk, hogy a VHDL-ben való fejlesztés az sokkal nehezebb, viszont több beleszólásunk van a hardveren kialakított mikró architektúrába, így egy minimálisan hatékonyabb rendszert lehet így kialakítani. Mivel C++-ban könnyebb volt megvalósítani a neuronháló modelleket, ezért úgy döntöttünk, hogy HLS köré építjük a keretrendszert. Ennek az volt az előnye, hogy gyorsabban haladtunk a fejlesztéssel és sokkal könnyebb volt bizonyos almodulokat általánosítani. Az elért eredményeket bemutattuk **XX. Online Kari Tudományos Diákköri Konferencián**, ahol 1. helyezést értünk el. Az MLP modell mellet elkezdtük fejleszteni az RBF modellt is. A továbbfejlesztett rendszert bemutattuk **XXII. Műszaki Tudományos Diákköri Konferencián**, ahol 2. helyezést értünk el és OTDK jelölést kaptunk.

2. Elméleti megalapozás és bibliográfiai tanulmány

2.1. Már létező magas szintű keretrendszerek

A mélytanuló algoritmusok és részei, mint például aktivációs függvények, tanítási módszerek, leírhatóak matematikai kifejezésekkel, valós alkalmazások esetén szükség van a programkódbeli leírásukra. E célból már létezik több nyílt forráskódú és kereskedelmi neurális háló könyvtár és keretrendszer.

2.1. táblázat: A már létező keretrendszerek funkcionalitásának az összehasonlítása [3, 4, 5].

Tulajdonság	CNTK	TensorFlow	Theano	Caffe	Keras	PyTorch
Implementáció	C++	C++	Python	C++	Python	C++
CPU támogatás	✓	✓	✓	✓	✓	ü
Több száll támogatása	✓	Eigen	Blas, conv2D, OpenMP	Blas	✓	✓
GPU támogatás	✓	✓	✓	✓	✓	✓
Több GPU támogatása	✓	✓	✓	✓	✓	✓

2.1.1. PyTorch

Az egyik legrégebbi a Torch[1] [4], 2002-ben jelent meg. Lua[2] szkript nyelven keresztül biztosít interfészt C-ben és CUDA-ban implementált matematikai modellekre. PyTorch[3] [4] ugyan arra Lua szkript alatt rejlő könyvtár alapra épít, viszont kiegészíti a rendszert egy Python alapú keretrendszerrel.

A fejlesztés során igyekeztek arra, hogy a rendszer az legyen egyszerű és könnyéden bővíthető. A tenzorokat, ha lehet akkor cache-elik, ezáltal gyorsítják a rendszert, mivel nem kell mindig új helyet foglalni a mátrixoknak. Ugyanakkor a paramétereket osztott memórián keresztül osszák meg a párhuzamosan futó folyamatok között, így nincs szükség sok folyamatközötti kommunikációra.

2.1.2. Theano

Az egyik népszerű Python könyvtár az a Theano[4], 2008-óta fejlesztik, a NumPy[5] könyvtáron alapszik. Erőssége, hogy nagyon jól kezeli a több dimenziós tárolókat, és hatékonyan tud velük műveleteket végezni. Több keretrendszer is épült erre a könyvtárra, főként azért, mert a könyvtár modelljei C-ben, illetve CUDA-ban vannak megvalósítva, ami gyors futást biztosít CPU-n és GPU-n egyaránt [6]. Ha teheti, akkor a matematikai modellek kiszámítására a grafikus gyorsítót alkalmazza, ezért sok esetben nem használja ki eléggé a CPU erőforrásokat [3].

[1] Torch: http://torch.ch/
[2] Lua: https://www.lua.org/
[3] PyThorch: https://pytorch.org/
[4] Theano: https://pypi.org/project/Theano/
[5] NumPy: https://numpy.org/

2.1.3. Caffe

A Caffe[6] [5] egy nyílt forráskódú mélytanuló könyvtár, 2014-ben jelent meg. C++ -ban van implementálva és neuronháló rétegek az alap számítási építőelemei. Moduláris, így könnyedén kiterjeszthető más modellekkel vagy költségfüggvényekkel. Ezek az almodulok automatizáltan vannak tesztelve mielőtt frissítik őket, így gyorsan tudnak iterálni a modulokon. Jól alkalmazható képfelismerésnél használt konvolúciós neurális hálózatok fejlesztésére és tanítására.

A keretrendszer biztosít Python és MATLAB interfészt is egyaránt, mivel mindkét nyelv előszeretettel van használva hálózatok felépítésére és különböző minták osztályozására. Python interfészen keresztül biztosítanak hozzáférést lassabb modellekhez is, amikkel lehet új prototípusokkal és új tanítóeljárásokkal kísérletezni.

2.1.4. TensorFlow

TensorFlow[7] 2015-ben volt publikálva a Google Brain csapat által. TensorFlow-ban a mélytanuló algoritmusok adat folyamként vannak felépítve. Ez felrajzolható mint egy irányított gráf. Egy számítás gráfban a csúcsok jelentik a műveleteket és az irányított élek az adat (tenzor[8]) áramlásának útját mutatják. Nem TensorFlow volt az első rendszer, ami gráfként modellezte a hálókat, viszont eltér az elődjeitől olyan téren, hogy a csúcsok komplex réteg műveletek helyett primitív matematikai operátorokat (mátrix szorzás, konvolúció stb.) alkalmaznak [7]. Ez az architektúra nagyon flexibilis és könnyeden skálázható [3].

2.1.5. Microsoft Cognitive Toolkit

CNTK[9] a Microsoft Research által volt fejlesztve és publikálva 2016-ban. TensorFlow-hoz hasonlóan egy irányított gráf mentén hajtotta végre a műveleteket. A csúcsok azok vagy paramétert, vagy egy műveletet jelképeztek, míg az élek ezeknek az útját mutatják [3]. Microsoft 2019-óta nem támogatja többet ezt a projektet.

2.2. Keretrendszerek neuronhálók FPGA alapú megvalósításra

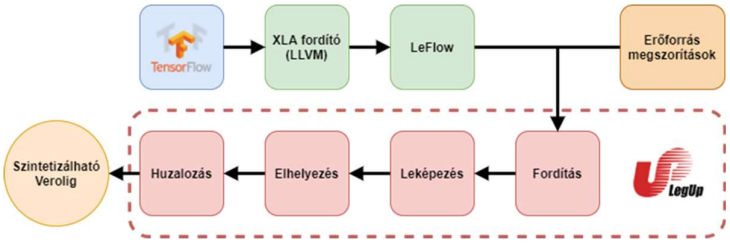
Az FP-DNN a szoftver alapú leírásból generálja az FPGA alapú megvalósítást, mindezt teljesen automatikusan. A TensorFlow modellből a saját szimbolikus fordítója generálja a C++ programot, amiből a HLS eszköz generálja a hardvert [8].

A Vitis AI a Xilinx fejlesztői platformja, neurális hálózatok Xilinx hardver-platformok számára. Tartalmaz optimalizált IP-ket, eszközöket, könyvtárakat, modelleket és teljes példákat. Használja

[6] Caffe: https://caffe.berkeleyvision.org/
[7] TensorFlow: https://www.tensorflow.org/
[8] Tenzor: Egy vagy több dimenziós adat tároló, mint például egy vektor vagy egy mátrix.
[9] Microsoft Cognitive Toolkit: https://github.com/microsoft/CNTK

a népszerű keretrendszereket (Caffe, TensorFlow PyTorch). Rétegenkénti elemzést végez, hogy segítsen a szűk keresztmetszetek megtalálásában. A hatékony és méretezhető IP magok testre szabhatóak.

LeFlow[10] [9] egy nyílt forráskódú könyvtár, ami TensorFlow által definiált numerikus modelleket Google által készített XLA fordító segítségével lefordítja egy LLVM formátumba, amiből egy HLS fordító, mint a LegUp[11] le tud fordítani regiszter szintű logikára (RTL). LeFlow ezt a fordítási folyamatot próbálja teljesen automatizálni, és optimalizálni HLS direktívák segítségével.



2.1. ábra: LeFlow keretrendszer folyamat ábrája [9].

3. Célkitűzések

A projekt célja egy keretrendszer megvalósítása, amely megkönnyíti a neuronháló modellek megvalósítását FPGA alapú rendszereken. A keretrendszer lehetővé kellene tegye különböző típusú és méretű neurális hálók FPGA alapú rendszereken történő modellezését. A keretrendszer segítségével a felhasználó képes kell legyen különböző párhuzamosítási szinttel rendelkező neuronhálót generálni, hogy alkalmazható legyen nagyobb és kisebb FPGA lapokon is egyaránt.

A tanulmány céljai közé tartozik a neurális hálózatok optimalizálása úgy teljesítmény növelés érdekében, mint a szükséges erőforrások csökkentés szempontjából is egyaránt.

4. A rendszer specifikációi és architektúrája

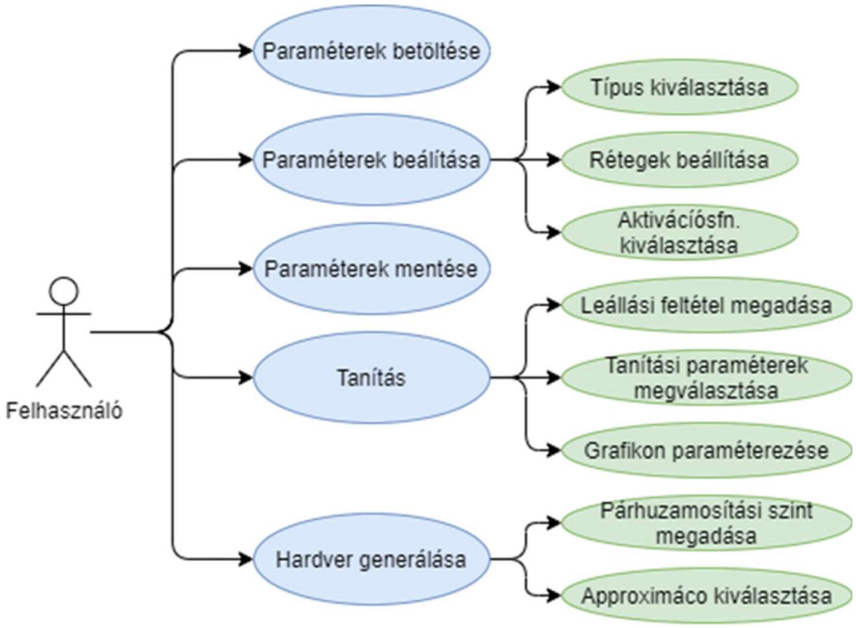
4.1. Felhasználói követelmények

A keretrendszer fő célja, hogy megkönnyítse a neuronháló modellek FPGA rendszereken való alkalmazását. A rendszer főbb funkcionalitásai (4.1. ábra) a következők:

- A neuronháló paramétereinek a beállítása: A felhasználó több típusú (MLP, RBF), méretű (rétegek száma, rétegenkénti neuronok száma) hálók közül tud választani, valamint ezek paramétereit szabadon tudja változtatni (aktivációs függvény típusa, és ennek paraméterei).
- Neuron háló tanítása: A felhasználó a felparaméterezet neuronhálókat tudja tanítani, valamint a tanítás folyamatnak a paramétereit be tudja állítani.

[10] LeFlow: https://github.com/danielholanda/LeFlow
[11] LegUp: https://www.legupcomputing.com/

- A háló paramétereinek a mentése: A betanított neuronháló paramétereit a felhasználó ki tudja menteni a későbbi felhasználásra.
- A hardver generálása: A felhasználó miután felparaméterezte és betanította a neuronhálót ki tudja generáltani ennek a direktívákkal ellátott HLS kódját.



4.1. ábra: A felhasználói követelmények.

4.2. Rendszer követelmények

4.2.1. Funkcionális követelmények

A keretrendszer kipróbálásához szükség van egy digitális eszközre, ami lehet asztaligép vagy laptop is egyaránt. A keretrendszer operációs rendszer független (lehet Windows, Linux vagy MacOS). A digitális eszköz képes kell legyen Python szkripteket futtatni és Python könyvtárcsomagokat telepíteni.

Ha a felhasználó új konfigurációt szeretne létrehozni vagy a példákat szeretné módosítani, akkor szükség lesz egy szövegszerkesztőre (mint például: Notepad, Notepad++, Visual Studio Code, PyCharm, stb.)

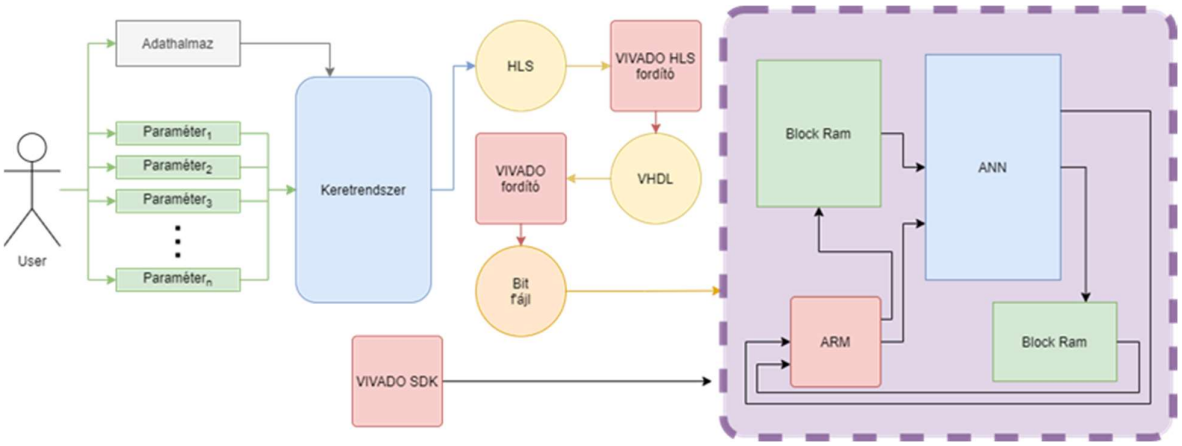
Amennyiben a felhasználó a generált HLS kódokat szeretné szintetizálni, lefordítani (egy hardverleíró nyelvre, mint VHDL vagy Verilog), vagy tesztelni, akkor szükség van a VIVADO HLS eszközre (vagy más HLS fordítóprogramra, mint például LegUp). A kigenerált hardver szintetizálását, leképezését, elhelyezését, halálozását és feltöltését VIVADO eszköz oldja meg. A VIVADO tervező eszköz csak Windows és Linux operációs rendszereket támogatja.

4.2.2. Nem funkcionális követelmények

A rendszer használatához a felhasználóknak szüksége egy digitális eszközre, amely képes Python szkripteket futtatni és Python könyvtárcsomagokat telepíteni. Az eszköz az lehet egy asztali számítógép, laptop, vagy akár egy szervergép is. Az operációs rendszer az lehet Windows vagy Linux.

4.3. A javasolt rendszer elvi architektúrája

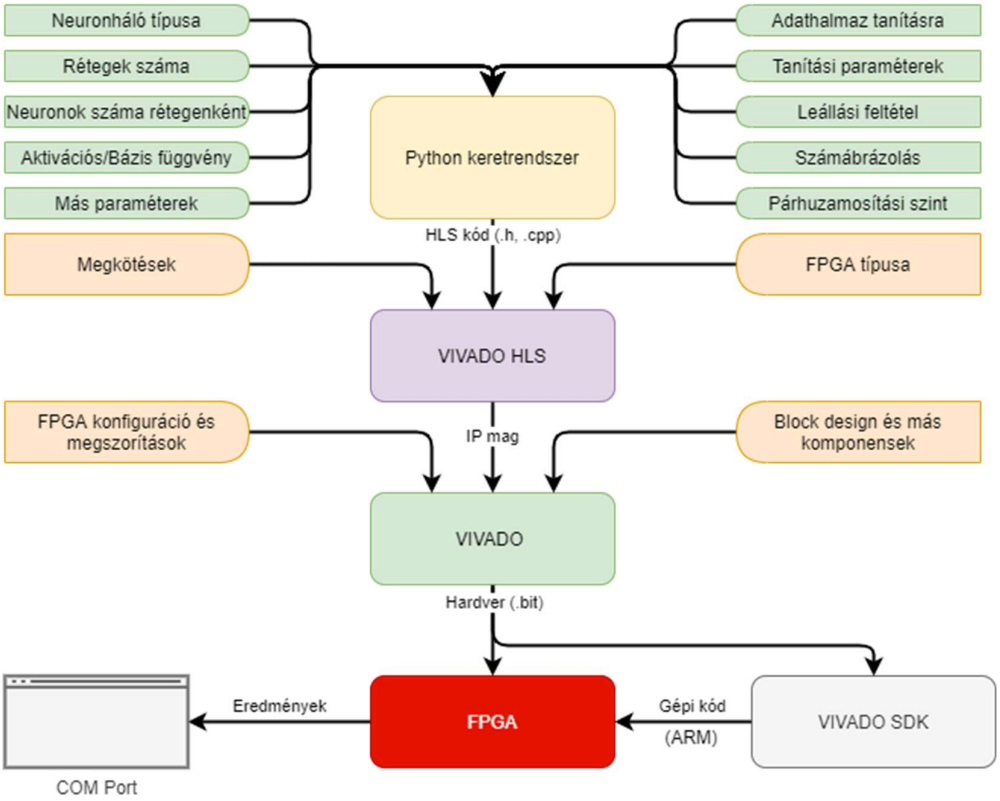
A keretrendszer, a felhasználó által megadott paraméterek alapján generálja a neurális C++ kódját. Ezt a kódot a Vivado vagy HLS eszköz szintetizálja és a Vivado elvégzi az implementációt (fordítás, leképezés, elhelyezés és huzalozás) és a programozást FPGA-ra. Az általunk javasolt rendszer tömbvázlatát a 2.1. ábra mutatja.



4.2. ábra: A javasolt rendszer tömbvázlata.

Az interfész megvalósítására egy potenciális megoldás lenne a Xilinx által készített ChipScopy használata, amin keresztül csatlakozva az FPGA virtuális ki-bemeneteire adagolni lehet a neuronháló bemeneteit. Sajnos ez a rendszer még nem teljesen elérhető, ezért a generált hardver tesztelésére a fejlesztőlapon található ARM processzort fogjuk használni.

A keretrendszer a direktívákkal ellátott HLS kódot fogja kigenerálni, amit IP (angolul: Intellectual Property) magként belehet majd illeszteni egy VIVADO-ban megvalósított projektbe. A VIVADO által generált hardver alapján a VIVADO SDK fejlesztői környezet segítségével, a fejlesztőlapon található 2 ARM (angolul: Advanced RISC Machines) processzort tudjuk konfigurálni és felprogramozni. Ezt a folyamatot a 4.3. ábra szemlélteti.



4.3. ábra: Hardver megvalósításának és tesztelésének a folyamata..

5. A megvalósítás során használt technológiák

5.1. Python

Python egy magas szintű, általános célú, interpretált nyelv [10], amelynek nagyon beszédes szintakszisa van. Sokan azt mondják rá, hogy egy „futtatható pszeudokód” [11]. Több programozási paradigmát követ, mint például: funkcionális, objektumorientált, imperatív. Magas szintű programozás révén a memóriakezelése automatikus (angolul: garbage collected) és dinamikus típusokat használ. Eredetileg Guido van Rossum fejlesztette, és 1991-ben tette nyilvánossá [10]. Python 2.0 2000-ben volt kiadva, amit a 3.0 követett 2008-ban. Az utóbbi az nem teljesen visszafele kompatibilis. A fejlesztés során a Python 3.7-t fogjuk használni.

Mivel a Python egy interpretált nyelv, ezért nem a leg hatékonyabb, viszont rengeteg ingyenesen elérhető könyvtárcsomaggal rendelkezik, ami megkönnyítik a fejlesztést. Ugyanakkor sok könyvtár egy alacsonyabb szintű programozási nyelvben van implementálva (mint például C++). A fejlesztés során több ilyen könyvtárat fogunk alkalmazni:

- NumPy: C++-ban megvalósított könyvtár numerikus számításokra
- Numba: Python függvényeket képes C/C++-ra fordítani, ezáltal növelve a teljesítményt
- MatplotLib (PyPlot): Leginkább grafikonok rajzolására van alkalmazva

5.1.1. Numpy

NumPy egy ingyenesen letölthető numerikus nyílt forráskódú könyvtárcsomag. Eredetileg 1995-ben Numeric néven fejlesztették, viszont sokan nem voltak megelégedve vele ezért elkezdtek fejleszteni egy vele kompatibilis könyvtárat, amit Numarray-nek neveztek. Ez azonban függőségi problémákhoz vezetett, mert egyes könyvtárka az egyiket használták, míg mások a másikat. Így 2005-ben elkezdték egyesíteni a két könyvtárat és 2006-ban adták ki NumPy néven. [11]
Erőssége a, hogy nagyon jól kezeli több dimenziós tárolókat (tömbök, mátrixok). Valamint rengeteg magas szintű matematikai műveletéhez ad hozzáférést.

5.1.2. Numba

Numba egy nyílt forráskódú futás idejű (angolul: Just In Time, rövidítése: JIT) fordító. Képes bizonyos Python és Numpy utasításokat egy gyors gépi kódra fordítani. Ugyanakkor segítségével nem csak felgyorsítani tudunk bizonyos kódrészleteket, hanem párhuzamosítani is. Mind ezek mellett még támogatja az NVIDIA CUDA technológiát is.

5.2. Xilinx Vivado Design Suite

A Vivado Design Suite egy olyan szoftvercsomag, amelyet a Xilinx készített a HDL-tervek szintézisére és elemzésére. Ez a szoftvercsomag rendelkezik magasszintű szintézis funkcióval is.

5.2.1. VIVADO HLS

Napjainkban a magas szintű szintézis (angolul: High-Level Synthesis, rövidítése: HLS) nagy érdeklődést élvez [12]. Főként azért, mert lényegesen könnyebben és gyorsabban lehet egy magasabb szintű programozási nyelvben fejleszteni (mint például: C/C++ vagy SystemC) mint a regiszter szinten való fejlesztés (angolul: Register Transfer Level, rövidítése: RTL).
Az RTL szinten való tervezéshez szakképzett és tapasztalt mérnökökre van szükség. A VHDL és a Verilog alacsonyabb szintű programozási nyelvek, amelyek jelentősen hosszú forráskódokhoz vezetnek, ami növeli a kódolási hibák valószínűségét, és időigényes folyamattá teszi a benne való tervezést. A HLS eszközök, mint a VIVADO HLS, egy magasabb absztrakciós szintben megvalósított modell alapján kigenerálják az RTL szintű megvalósítást, amit majd alkalmazhatunk FPGA alapú rendszereken.

5.2.2. VIVADO SDK

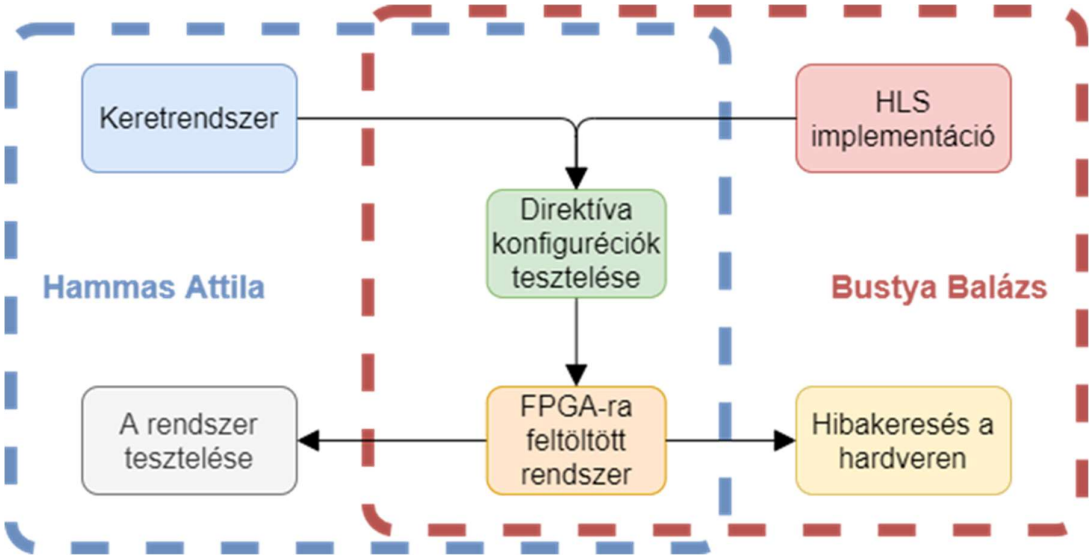
Xilinx SDK egy integrált tervezői környezet beágyazót alkalmazások megvalósítására. Az SDK az Eclipse integrált fejlesztői környezeten alapszik és lehetőséget biztosít tervezésre, hibakeresésre és teljesítmény elemzésre. Támogatja a Xilinx által tervezett Zynq UltraScale+ MPSoC, Zynq-7000 SoC és más szoft proceszorokat (MicroBlaze) is egyaránt [13].

6. Metodológia

A tervezés az fentről lefele haladva történt. Azaz legelőször a legfelső absztrakciós szint feladatait határoztuk meg, majd azokat alfeladatokra és almodulokra bontottuk. A megvalósítás során viszont a legkisebb feladatok és modulok implementációjával kezdtük. Majd ezeket egymással való összekötésével építettük fel a nagyobb modulokat és valósítottuk meg a magasabb absztrakciós szintű funkcionalitásokat. A kiindulási pont az az alapműveletek megvalósítása volt, mint a mátrix szorzás vagy egy aktivációs függvény alkalmazása. Ezeknek a modulok kombinációjával már ki tudtunk alakítani egy neuront vagy a mátrix méretének a növelésével egy egész neuronháló réteget. Ezt követően a réteg paramétereinek a tanításával foglalkoztunk. A rétegek egymásután helyezésével egy többrétegű neuronhálót tudtunk kialakítani. Amint kialakítottuk az első többrétegű neurális hálót, kellet implementálnunk a hiba visszaterjesztést rétegek között. Ezen a szinten már az általánosítást kell megoldani, hogy a neuronháló típusa, topológiája pár paraméter változtatásával lehessen könnyeden beállítani. Gondoltunk arra is, hogy a későbbiekben a rendszert könnyedén lehessen bővíteni is újabb almodulokkal (aktivációs függvényekkel, bázisfüggvényekkel), ezért mindent egy minimális főosztályból volt származtatva. Az előbb bemutatott lépéseket az implementált háló típusokon egyesével végeztük, így jobb rálátást szereztünk abból a szempontból, hogy különböző műveleteknek milyen erőforrás igényük van. Ugyanakkor bizonyos műveleteket különböző approximációs módszerekkel próbáltunk helyettesíteni annak érdekében, hogy kevesebb erőforrásra legyen szükség, illetve növeljük a teljesítményt vagy a pontosságot..

7. Tervezés és megvalósítás

A rendszer megvalósításán társammal, Bustya Balázzsal, dolgoztam. A feladat leosztást a 7.1. ábra szemlélteti. Én főként a Pythonban megírt keretrendszerrel foglalkoztam, míg a társam a HLS modulok implementálásával foglalkozott. Így a következőkben főként a Python modell lesz bemutatva és a HLS-ben megvalósított modulok nem lesznek részletezve. A méréseket közösen végeztük: a Pythonban megírt rendszer által volt paraméterezve a HLS-ben megvalósított modulokat, és a szimulációk és a mérések az így kialakult hardver architektúrán voltak elvégezve.



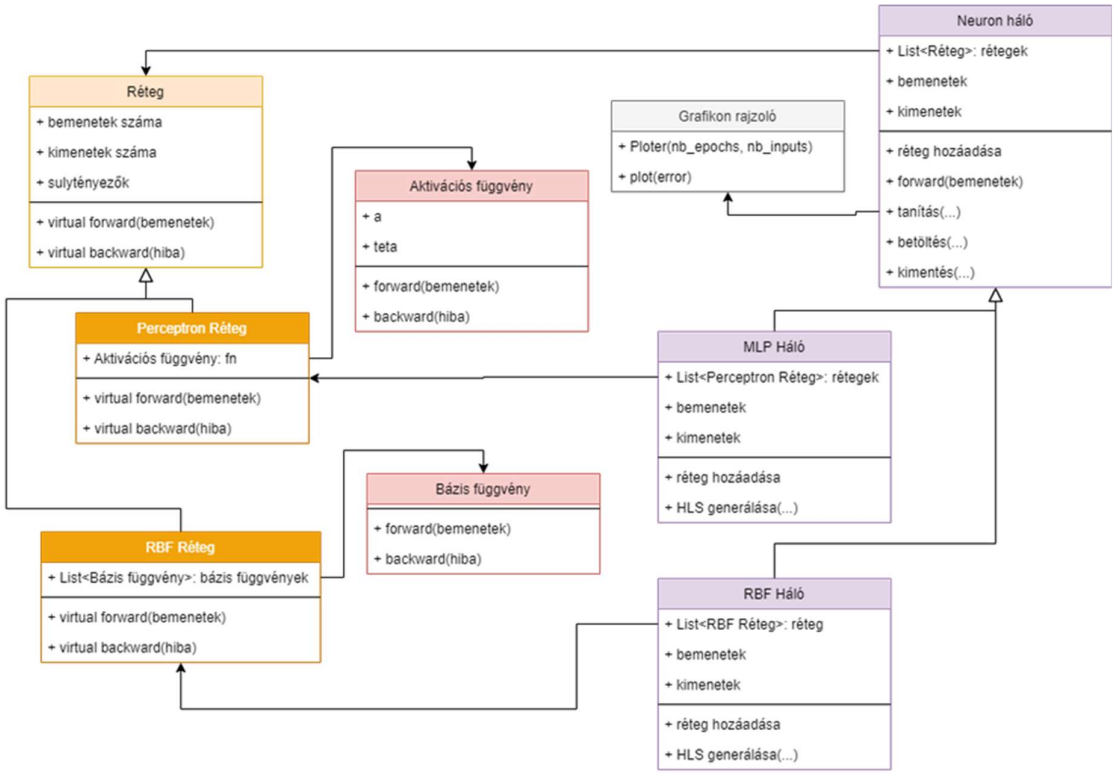
7.1. ábra: A feladatok leosztása.

7.1. A keretrendszer alkomponensei

A tervezett keretrendszert azt Python programozási nyelv segítségével készült el. A rendszer tervezése során fontos volt az, hogy a kész rendszer átlátható és könnyeden tovább fejleszthető legyen, ezért az Objektum Orientált Programozási (OOP) elveket követi. Ez enyhént csökkentheti a teljesítményt, viszont a kód az könnyebben értelmezhető, valamint a rendszer későbbi továbbfejlesztése is hatékonyabb.

A rendszer 3 fő absztrakciós szintből épül fel:

1. **Neurális háló**: ez a legkülső absztrakciós szint, tartalmazza a réteg modulokat, és biztosítja a közöttük való kommunikációt. Ez felel a teljes hálóval végzendő műveletekért, mint például architektúra kialakítása, háló tanítása, háló kimenetének kiszámítása vagy a modell és a paraméterek alapján a HLS-re való leképezés. A neuronhálóban a réteg modulok egy egyszerű listában vannak eltárolva. A specifikus neuronhálótípusok ebből a neuronháló osztályból vannak örökítve.
2. **Réteg**: A rétegek tartalmazzák a számukra szükséges almodulokat, mint például bázisfüggvény, vagy aktivációsfüggvény. Ezek az alkomponensek megfelelő alkalmazásáért ők felelnek.
3. **Különböző alkomponensek**: ide tartoznak például az aktivációs függvények vagy bázisfüggvények.



7.2. ábra: A rendszer osztály diagramja.

7.1.1. Aktivációs függvény osztály

A neurális hálókban többféle aktivációs függvény modellt alkalmaznak: küszöb függvény, lineáris függvény, logisztikus függvény, szakaszonként lineáris függvény, és még sok más.

Előre csatolt neurális halókban egy csomópont aktiválási funkciója határozza meg annak a csomópontnak a kimenetét, amely adott bemenetet kapott. Annak érdekében, hogy nem triviális problémák kiszámítását minél kevesebb neuron használatával oldjuk meg érdemes nem lineáris aktivációs függvény használni.

A hiba visszafejtése során az aktivációs függvény első rendű deriváltját megszorozzuk a hiba mértékével:

$$error' = error \cdot f'(s), \quad ahol\ s\ az\ inger\ amely\ az\ adott\ hibához\ vezetett. \tag{7.1}$$

Ezt a műveletet minden egyes neuron esetében el kell végzeni így a következőképpen fog kinézni:

$$\begin{bmatrix} error'_0 \\ error'_1 \\ \vdots \\ error'_n \end{bmatrix} = \begin{bmatrix} error_0 \\ error_1 \\ \vdots \\ error_n \end{bmatrix} \odot \begin{bmatrix} f'(s_0) \\ f'(s_1) \\ \vdots \\ f'(s_n) \end{bmatrix}, \quad ahol \odot elemenkénti\ szorzást\ jelent \tag{7.2}$$

A fent bemutatott műveletek numpy és numba könyvtárcsomagok segítségével vannak megvalósítva.

```
    @staticmethod
    @vectorize(['f8(f8,f8,f8)'], nopython=True)
    def _forward(x, a, teta):
        return 1 / (1 + math.exp(-(a * (x - teta))))

    def forward(self, x):
        self.x = x
        self.y = self._forward(x, self.a, self.teta)
        return self.y

    @staticmethod
    @vectorize(['f8(f8,f8,f8)'], nopython=True)
    def _vdiff(x, a, teta):
        exp = math.exp(-(a * (x - teta)))
        return a * exp / (1 + exp)**2

    def backward(self, error):
        dfpdx = SigmoidFunction._vdiff(self.x, self.a, self.teta)
        error = np.multiply(error, dfpdx)
        return error
```

7.3. ábra: Kódrészlet egy aktivációs függvény implementálásáról.

A fenti ábrában látható, hogy az alul vonassál („_”) kezdődő függvények a numba könyvtárcsomag @vectorize annotációval vannak ellátva, emiatt futás időben át lelesznek fordítva C-re, ami felgyorsítja a futásidőt, valamint a függvény úgy alkalmazható lesz egy elemre, mint egy tömbre is egyaránt. A másik két függvény definíciójában csak az előbb említett függvények meghívása történik illetve a numpy könyvtárcsomag np.multiply függvénye van meghívva.
A fontosabb aktivációs függvények amelyek implementálva vannak:

1. Identitás függvény és egyszerű lineáris függvény:

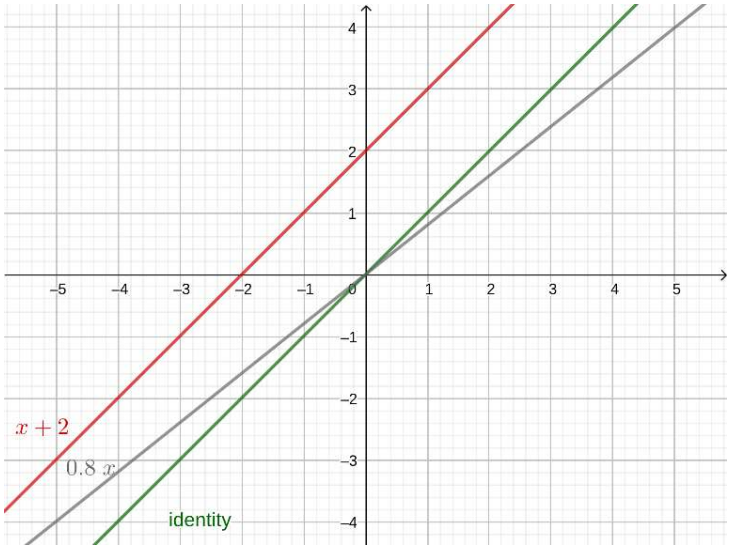
   Az identitás függvény a legegyszerűbb lineáris függvény, a kimenet megegyezik a bemenettel és a következő képen írható fel (7.3):$f(x) = x$.

   $$f(x) = x \tag{7.3}$$

   Ritkán van alkalmazva napjainkban. Lehet paraméterezni: $f(x) = a(x - \theta)$

   $$f(x) = a(x - \theta) \tag{7.4}$$

   A (7.4) egyenletben az a paraméter a meredekségét és a θ paraméter az eltolást alítja.

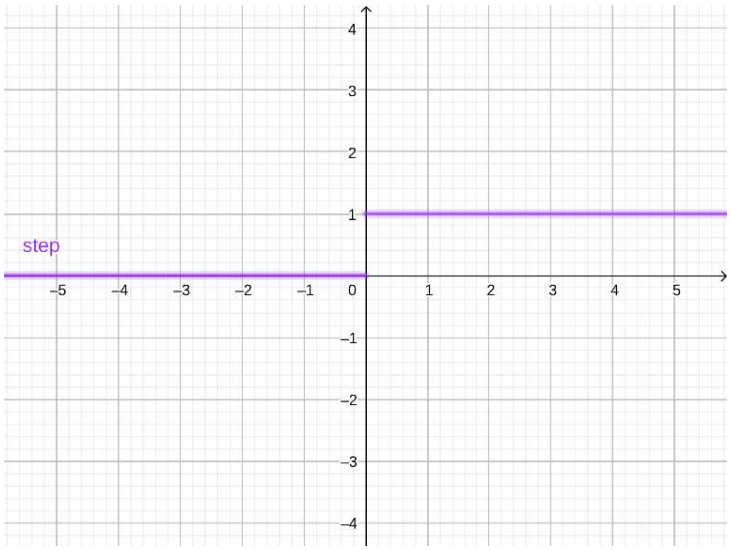


7.4. ábra: Identitás- és lineáris aktivációs függvény grafikus képe.

A hiba visszafejtésénél a lineáris függvény deriváltja a következő képen néz ki (7.5):

$$f'(x) = a \tag{7.5}$$

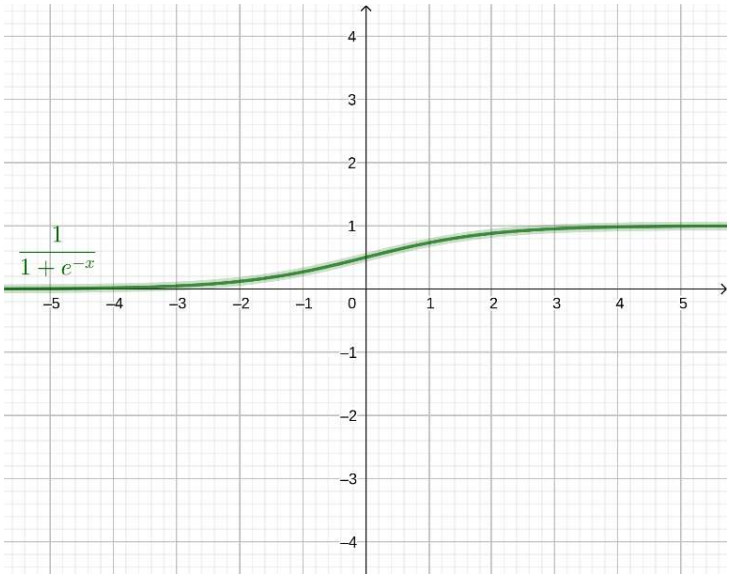
2. Lépcső függvény:



7.5. ábra: Lépcső aktivációsfüggvént grafikus képe.

A lépcső függvény az egy egyszerű két állapotú függvény, ami vagy 1, vagy 0. A lépcső függvény grafikus képét a 7.5. ábra mutatja. Ha a kimenet értéke az -1 és 1 között kell legyen akkor hasonló képen lehet használni az előjel függvényt is.

3. Logisztikus függvény (más néven Szigmoid függvény):



7.6. ábra: Szigmoid aktivációsfüggvény grafikus képe.

A Szigmoid aktivációs függvény (7.6) az adott ingert az beszorítja 0 és 1 közé. A függvény grafikus képét a 7.6. ábra mutatja.

$$f(x) = \frac{1}{1 + e^{-a(x-\theta)}} \tag{7.6}$$

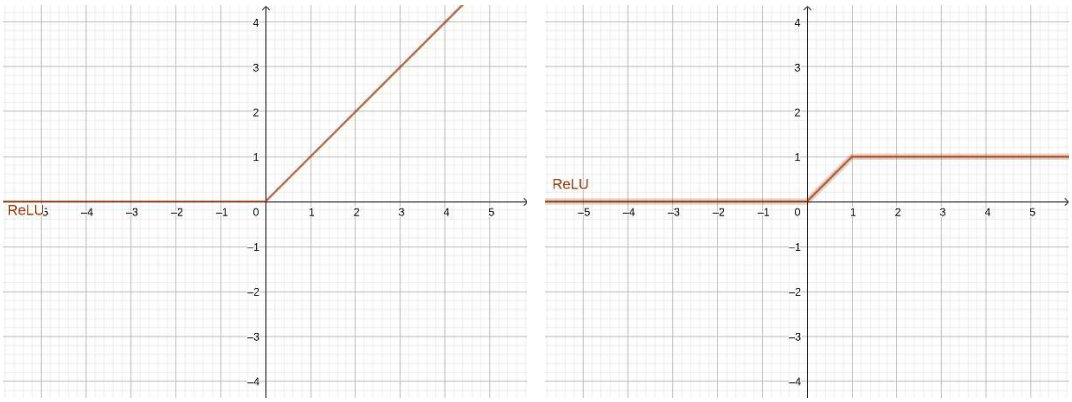
A hiba visszaterjesztése során a deriváltja:

$$f'(x) = f(x) \cdot \big(1 - f(x)\big) = \frac{a \cdot e^{-a(x-\theta)}}{(1 + e^{-a(x-\theta)})^2} \tag{7.7}$$

4. Rectified linear unit (ReLU)

A ReLU (7.8) az előbb bemutatott Szigmoid függvényhez hasonlóan az ingert azt 0 és 1 közé szorítja vagy a negatív értékek helyet 0-t ad. A függvény grafikus képe az alábbi ábrán (7.7. ábra) látható.

Előszeretettel, van alkalmazva, mert kevesebb neuronra van szükség, mint ha mondjuk egy egyszerű lineáris vagy identitás aktivációs függvényt alkalmaznánk, és gyakorlatban kevesebb számítási kapacitást igényel, mint a Szigmoid vagy a Softplus függvény. A deriváltját a (7.9) egyenlet mutatja.

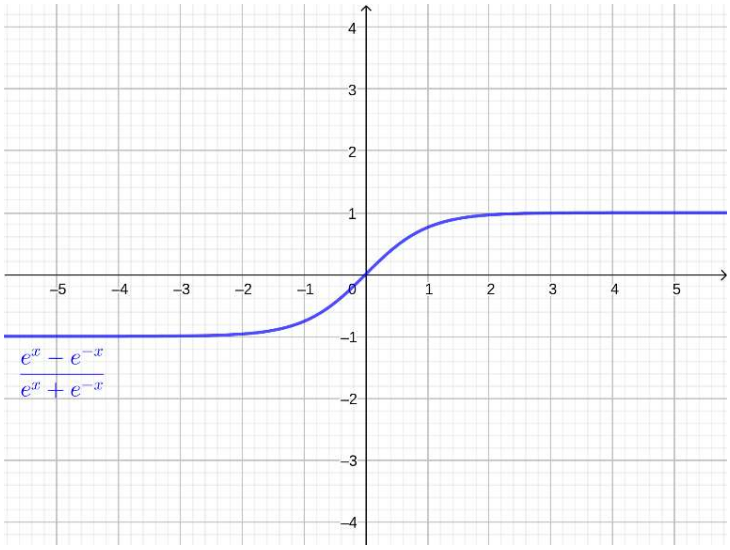


7.7. ábra: ReLU aktivációs függvény grafikus képe.

$$f(x) = \begin{cases} ax, & x > 0 \\ 0, x \leq 0 \end{cases} \text{ vagy } f(x) = \begin{cases} 1, x > 1/a \\ ax, & 0 \leq x \leq 1/a \\ 0, 0 < 0 \end{cases} \tag{7.8}$$

$$f'(x) = \begin{cases} a, & x > 0 \\ 0, x \leq 0 \end{cases} \text{ vagy } f'(x) = \begin{cases} 0, x > 1/a \\ a, & 0 \leq x \leq 1/a \\ 0, 0 < 0 \end{cases} \tag{7.9}$$

5. Tangens hiperbolikus



7.8. ábra: Tangenshiperbolikus aktivációsfüggvény grafikus képe.

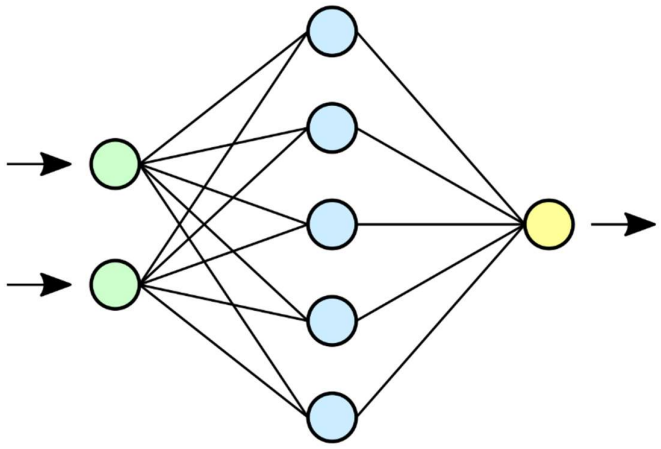
Hasonló tulajdonságokkal rendelkezik, mint a Szigmoid aktivációs függvény, annyiban különbözik, hogy az értékkészlete az $(-1, 1)$ intervallumban van.

Abban azesetben, ha a keresett függvény nincs definiálva, akkor a felhasználó létrehozhat egy új függvény modellt. Annak kell teljesítenie az „AktivationFunction” interfészt, azaz rendelkeznie kell a következő metódusokkal:

- **forward metódus** egy adott ingerre kell egy kimenetet megfeleltessen

- **backward metódus** a hibának a függvényen keresztüli visszafejtését kell megvalósítsa.
- **to_obj metódus** (opcionális) az aktivációs függvény fő paramétereit kell visszatérítse egy objektumba, ez szükséges, ha el szeretnénk menteni a hálót, amibe ez az aktivációs függvény alkalmazva van.

7.1.2. Perceptron réteg



7.9. ábra: Egy többrétegü perceptron neruron háló [14].

A neurális hálók strukturális és funkcionális szempontból 3 részből tevődnek össze: egy bemeneti réteg, rejtett rétegek (0 vagy több) és egy kimeneti réteg.

A bemeneti réteg feladata, hogy a bemeneti adatot módosítatlanul átadja a következő rétegnek (ez lehet egy rejtett réteg vagy egyből a kimeneti réteg) így ez a réteg nem rendelkezik a szorzó modulokkal, az összeadókkal és az aktivációs függvényekkel. Ez a réteg a „numpy" könyvtárcsomagban definiált tömbbel van modellezve.

A rejtett rétegek a bemeneti réteg és a kimeneti réteg között, egymás után helyezkednek el. Itt már jelen vannak a szorzó, összeadó és aktivációs függvény alegységek. A szorzó és összeadó alegységeket egy mátrix szorzással modelleztük. Ha a neuronokat oszloponként helyezük el, azaz minden oszlop egy neuron súly tényezőit tartalmazza, akkor az ingerkiszámítása során egyel kevesebb transzformációra van szükség [15].

$$[b \quad x_0 \quad x_1 \quad \cdots \quad x_m] * \begin{bmatrix} w_{b,0} & w_{b,1} & \cdots & w_{b,n} \\ w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,0} & w_{m,2} & \cdots & w_{m,n} \end{bmatrix} = [b \quad s_0 \quad s_1 \quad \cdots \quad s_m] \qquad (7.10)$$

Amint a (7.10 egyenletben látható egy réteg ingereinek kiszámítása felírható egy mátrix szorzásként. Egy inger kiszámításához szükséges képlet alább látható (7.11).

$$s_i = b * w_{b,i} + \sum_{j=0}^{m} x_j * w_{j,i} \tag{7.11}$$

A kapott inger tömb (S) elemeit majd beadjuk az aktivációs függvénynek.

A hiba visszaterjesztése az alábbi képlet szerint történik, ahol az $error'$ az aktivációs függvényen visszaterjesztetett hiba:

$$Error_{visszafejtett} = \begin{bmatrix} error'_0 \\ error'_1 \\ \vdots \\ error'_n \end{bmatrix} \cdot \begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,0} & w_{m,2} & \cdots & w_{m,n} \end{bmatrix}^T =$$

$$= \begin{bmatrix} error'_0 \\ error'_1 \\ \vdots \\ error'_n \end{bmatrix} \cdot \begin{bmatrix} w_{0,0} & w_{1,0} & \cdots & w_{m,0} \\ w_{0,1} & w_{1,1} & \cdots & w_{m,1} \\ \vdots & \vdots & \ddots & \vdots \\ w_{0,n} & w_{1,n} & \cdots & w_{m,n} \end{bmatrix}, \tag{7.12}$$

ahol error' az aktivációs függvényen visszaterjesztett hiba.

A mátrix szorzás műveletek a numpy könyvtárcsomagban definiált np.dot vagy np.matmul metódusok segítségével van elvégezve. A réteg rendelkezik egy „use_bias” értékkel, ami lehetővé teszi egy tanítható bias alkalmazását. Ha ez az érték igaz, akkor a forward metódusban a bemenet kap egy 1 értéket a tömb elejére, valamint a réteg létrehozása során a súly mátrix az eggyel több sort fog tartalmazni. A hiba visszaterjesztése során az első sor (bias értékek) mellőzve lesznek. A paraméterek tanítása ki van emelve egy különálló függvénybe.

```
def forward(self, inputs):
    x = np.array(inputs, ndmin=2)
    if(self.uses_bias):
        bias = np.ones((x.shape[0] if x.ndim != 1 else 1, 1))
        x = np.hstack((bias, x))
    self.x = x
    self.s = np.dot(x, self.weights)
    self.y = self.fn.forward(self.s)
    return self.y

def _adjust(self, error, u, m):
    momentum = self.weights_prev - self.weights
    self.weights_prev = self.weights.copy()
    self.weights = \
        self.weights + u * np.dot(error.T, self.x).T + m * momentum

def backward(self, error, u=.1, m=.01):
    error=self.fn.backward(error)
    self._adjust(error=error, u=u, m=m)
    error = np.dot(error, self.weights.T)
    if (self.uses_bias):
        error = np.delete(error, 0, axis=1)
    return error
```

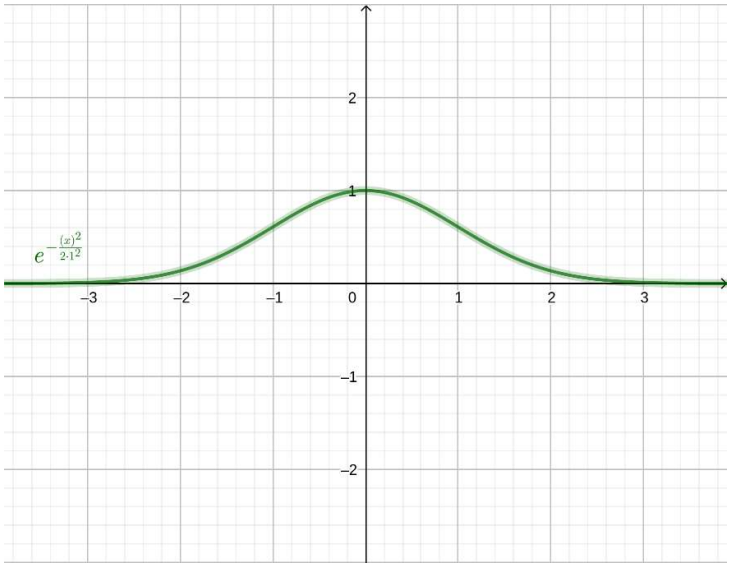
7.10. ábra: Perceptron réteg forward és backward metódusainak definiciója.

7.1.3. Bázis függvény

A bázis függvény egy nem lineáris függvény. Az aktivációs függvénnyel ellentétben ennek lehet több mit egy bemenete. Az egyik legalkalmazottabb az a Gauss függvény:

$$f(x) = e^{-\frac{(\|x-c\|)^2}{2\cdot\sigma^2}}, ahol\ c\ a\ görbe\ középpontja\ és\ \sigma\ a\ szorás. \tag{7.13}$$

Az megfigyelések azt mutatják, hogy a bázis függvény kiválasztásának feladata nem kulcsfontosságú az RBF hálók hatékonysága szempontjából. Így a legtöbb esetben a Gauss bázisfüggvényt alkalmazzák [16], ennek a grafikus képét a 7.11. ábra mutatja.



7.11. ábra: A Gauss függvény grafikus képe.

A Gauss függvény implementálása során numpy könyvtárcsomagban definiált 2-es normát (7.14) alkalmaztuk:

$$xmc = \sqrt[2]{\sum_{i=0}^{n}(x_i - c_i)^2} = \|\underline{X} - \underline{C}\|, \tag{7.14}$$

$$xmc\ a\ távolság\ a\ középponttól, n\ a\ dimenziók\ száma,$$
$$\underline{X}\ a\ bemenet\ vektor\ és\ \underline{C}\ a\ bázisfüggvény\ középpontja.$$

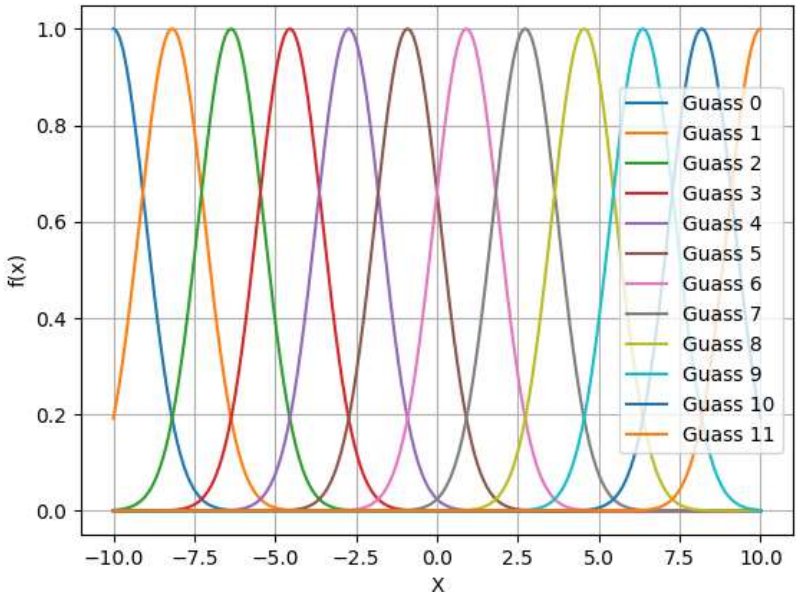
Sajnos a numba könyvtár nem támogatja ezt a függvényt ezért a bázisfüggvény implementálása során nem volt alkalmazva (7.12. ábra).

```
    @staticmethod
    def _forward(x: float, c: float, s: float) -> float:
        xmc = np.linalg.norm(x - c)
        return math.exp(-.5 * (xmc / s)**2)

    def forward(self, input: float) -> float:
        input = np.squeeze(input)
        return self._forward(input, self.c, self.sigma)

    @staticmethod
    def _diff(x: float, c: float, s:float):
        xmc = np.linalg.norm(x - c)
        ss = pow(s, 2)
        return -xmc * math.exp(-.5 * math.pow(xmc, 2)/ss) / ss

    def backward(self, x, error):
        diff = self._diff(x, self.c, self.sigma)
        error = np.multiply(error, diff)

        return error
```

7.12. ábra: A Gauss függvény forward és backward metódusainak definiciója.

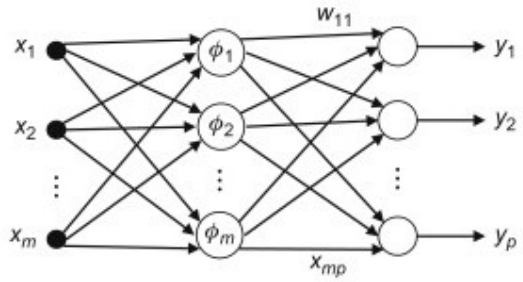
A Gauss függvény osztály rendelkezik segéd statikus függvényekkel, amelyek egyenletesen el tudják osztani a bázis függvények középpontjait egy egyenesen vagy egy síkban.

A következő ábrán (7.13. ábra) látható 12 Gauss függvény, amiknek a középpontjai egyenletesen vannak szétosztva a [-10, 10] intervallumon.



7.13. ábra: Gauss függvények egyenletes elosztása egy egyenes mentén.

7.1.4. RBF réteg



7.14. ábra: Egy RBF háló [16].

Egy RBF háló 2 fő részből tevődik össze: egy bázisfüggvényeket tartalmazó rétegből és egy neuronokat tartalmazó rétegből. A bázisfüggvényeket tartalmazó réteg kimenetének kiszámítása lényegében a bázisfüggvényeknek a kiértékelése A neuronokat tartalmazó réteg az nagyban hasonlít egy Perceptron réteghez, az eltérés csupán annyi, hogy nem rendelkezik aktivációs függvénnyel. Így ennek a rétegnek a kiszámítása megegyezik a 7.1.2 fejezetben bemutatott (7.10) és (7.11) lépésekkel.

A hiba visszalejtés során is a Perceptron réteghez hasonló an járunk el (7.12). A különbség az, hogy ebben az esetben mivel nincs aktivációs függvény az $error'$ az megegyezik a kimeneten levő költségfüggvény értékével.

7.1.5. Neurális háló

A neurális háló osztály magába foglalja a rétegeket és ő felelős a neuronháló adott bementére a kimenet kiszámításáért, a háló tanításáért és a HLS modell sablon paraméterezéséért. Ugyanakkor ez az osztály biztosít egy interfészt, amin keresztül tudunk műveleteket végezni a hallóval.

7.2. Tanítás

A háló paramétereinek tanítására egy gradiens alapú módszer van alkalmazva a hibát egyszer visszaterjesztjük az aktivációs függvényen (ha az adott rétegtípus rendelkezik ilyennel), azaz hibát megszorozzuk az aktivációs függvény deriváltjával, ahogy a (7.1) és (7.2) egyenlet mutatja. Az így kapott hibát majd alkalmazzuk a súly paramétereknek a javítására, optimalizálására:

$$W_{<t+1>} = W_{<t>} + \mathrm{u} * \mathrm{Error}^{\mathrm{T}} * \mathrm{X} + \mathrm{m} * \mathrm{M} \qquad (7.15)$$

A súlyok tanítását több paraméter befolyásolja:

- **Tanítási együttható (u)**: ez meghatározza, hogy mekkora lépésekkel haladunk a költség függvény optimuma felé
- **Momentum mátrix (M)**: ez a mátrix jelöli, hogy az előző iterációban alkalmazott lépést
- **Momentum együttható (m)**: mekkora hatása legyen a momentumnak a tanítás során

Ha a hálónak van egy vagy több rejtett rétegje, majd ugyan ezt a tanítási folyamatot elvégezzük az összes rétegen egymás után. A rétegek között a hibát a (7.12) képlet szerint fejtsük vissza:

7.3. A HLS kód generálása

A HLS kód az egy előre elkészített C++ sablon alapján történik. Egy sablon az áll egy fejléc állományból (.h vagy .hpp) és egy forráskódból (.cpp). Az utóbbi az tartalmazza a háló és annak a komponenseinek és függvényeinek a definícióit. A header állományban jelen vannak a hálót meghatározó paraméterek (mint, az architektúra, súly paraméterek, aktivációs függvény stb.), az fordítás során alkalmazandó direktívák, a használt adattípus és a használt függvények és almodulok deklarációja. A generálás során csupán csak a header állomány van dinamikusan létrehozva a sablon alapján, míg a C++ állomány az csupán át van másolva.

7.3.1. Számábrázolás

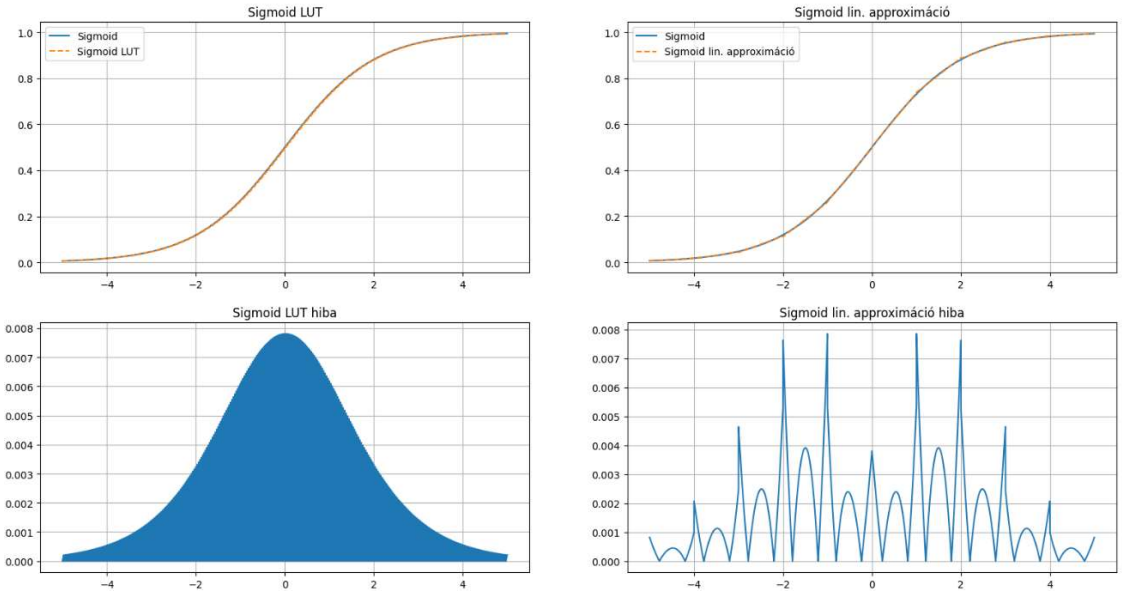
Számábrázolás szempontjából három lehetőséget volt vizsgálva: lebegőpontos, fixpontos és egész számon való ábrázolás. Mivel a lebegő pontos ábrázolás és az azzal végzett aritmetika az több erőforrást igényelt és nem föltétlen javított a pontosságon [17, 18] , ezért arra nem fektettünk nagy hangsúlyt, hanem a fixpontos és egész számábrázolást használtuk inkább. Pontosabban 16 bites fixpontos számábrázolást, ahol 1 bit az előjel 4 bit az egész rész és 11 bit a tört rész. Más kutatások is hasonló konfigurációkat javasoltak annak érdekében, hogy a számítási pontosságvesztés az minimális legyen [19].

7.3.2. Aktivációs függvények és bázis függvények megvalósítása

Az aktivációs függvények értékeit, ha nem DSP-n akarjuk kiszámolni, akkor kézenfekvő megoldás lehet, hogy a függvény értékei egy keresőtáblába (angolul: LookUp Table, röviden: LUT) tároljuk [20, 19]. Ez által sok számítási kapacitást lehet spórolni, valamint lényegesen le lehet csökkenteni az alkalmazandó DSP -k számát.

7.1. táblázat: Különböző aktivácós függvény implementációk erőforrás szükségletének összehasonlítása [21].

Architektúra	BlockRAM	DSP	Flip-Flop	LUT	Késleltetés
35-10 (DSP)	0	33	7986	13755	80,85 μs
35-10 (LUT)	0	4	1505	19951	73,29 μs
35-10 (LUT lin. approximációval)	2	5	1512	3332	39,45 μs



7.15. ábra: Különböző aktivácós függvény implementációk összehasonlítása [21].

Ha az aktivációsfüggvény értékeit LUT-okban tároljuk akkor sok erőforrásra van szükség. Az erőforrás igényt le lehet csökkenteni, ha az aktivációs függvényt szegmensekre bontsuk és egy (7.16) alakú lineáris függvénnyel megközelítsük [22, 23, 24]. Igy 320 érték helyet csak 20 értéket kell LUT-okba tárolni Viszont szükség lesz egy DSP-re, amivel kiszámoljuk a lineáris függvény értékét. Meglepő módon a késleltetés is sokkal kisebb lett, ez valószínűleg annak tudható be, hogy kisebb LUT-ból kell kikeresni a megfelelő értéket.

$$f(x) = ax + b \tag{7.16}$$

7.3.3. MLP háló sablonja

Első sorban definiáljuk a számábrázolást, amit alkalmazni akarunk a háló létrehozása során. Ezt egy egyszerű „typedef” kulcsszóval meg lehet tenni. Például az alábbi ábrán (7.16. ábra) egy 16 bites számábrázolás konfigurációja látható, ahol 5 bit az egész rész, amiből 1 bit az előjel és a maradék 11 bit a törtrész.

```
/** DATA TYPE */
#define DATA_WIDTH 16
#define INT_WIDTH 5
#define FRACTION_WIDTH 11
typedef ap_fixed<DATA_WIDTH, INT_WIDTH, AP_RND, AP_SAT> MY_FIXED;
```

7.16. ábra: Példa használt típus beállítására.

Egy több rétegű Perceptron háló topológiáját 5 konstans beállításával lehet meghatározni (7.17. ábra):

- perceptron_input: ez a paraméter segítségével határozzuk meg a háló bemeneti rétegének a méretét.

- layer_number: ezzel tudjuk meghatározni a háló rétegeinek a számát; mivel a bemeneti réteg módosítatlanul továbbítja a bemenetként átadott adatot a hálózat többi részének ezért azt nem számoljuk, így ez a paraméter a rétegek száma - 1 értékkel lesz egyenlő.
- output_number: ez a paraméter a kimeneti réteg méretét adja meg.
- neurons: ez a paraméter egy tömb, ami meghatározza a háló rétegéiben a neuronok számát.
- greatest_layer_neurons: Ez a legnagyobb réteg + 1 értékkel lesz egyenlő; ennek az oka, hogy helyfoglalás során szükség van plusz egy tárolóra, ami a bias-t fogja tartalmazni.

```
/** NETWORK */
const unsigned perceptron_input = 35; /// Number of inputs.
const unsigned layer_number = 2;      /// Number of layers - 1 .
const unsigned output_number = 10;    /// Number of outputs.
const unsigned neurons[layer_number + 1] = {35, 16, 10};
const unsigned greatest_layer_neurons = 36;
const MY_FIXED W_glob[layer_number][greatest_layer_neurons][greatest_layer_neurons] = {
    {…
    {…
};
```

7.17. ábra: Példa MLP háló topológia beállítására.

Ezen kívül szükség van egy 3 dimenziós tömbre, ami tartalmazza minden réteg súly mátrixát és a és az eltolási értéket (angolul: bias) értékeket. A fenti ábrán (7.17. ábra) ez össze van csukva, mivel nagyon sok elemet tartalmaz.

Az alkalmazott aktivációs függvényt egy „ACTIVATION_FUNCTION” makró beállításával tudjuk megoldani. Ezen kívül meg kell határozzuk a függvény által alkalmazott keresőtáblákat. A következő képen (7.18. ábra) egy szigmoid aktivációs függvény van alkalmazva, aminek az értékei egyszerűen a „sigmoid_LUT” változóban vannak tárolva.

```
/** ACTIVATION FUNCTION */
#define ACTIVATION_FUNCTION sigmoid
const unsigned sigmoid_LUT_size = 640;
const MY_FIXED sigmoid_LUT[sigmoid_LUT_size] = {0.5, 0.5019531150659532, 0.5039061705290805, ...
```

7.18. ábra: Példa MLP háló aktivációs függvényének beállítására.

Végül pedig beállítjuk a direktívákat. A fenti ábrán (7.19. ábra) látható egy direktíva konfiguráció, ahol egy réteg neuronjait párhuzamosan hajtsuk végre, valamint egy neuron műveletei csővezetékesítve vannak.

```
/** DIRECTIVES */
#define VECTOR_MULT_DIRECTIVE #pragma HLS pipeline
#define NEURON_DIRECTIVE #pragma HLS unroll
```

7.19. ábra: Példa MLP háló direktíva konfigurációjának a beállítására.

7.3.4. RBF háló sablonja

A számábrázolást beállítása ugyanúgy történik, mint a 7.3.3 pontban bemutatott MLP sablon esetében. Ezt követően hasonló képen be kell állítani a topológiát meghatározó 3 konstanst (7.20. ábra). Az alkalmazott bázisfüggvényt egy „BASE_FUNCTION” makró beállításával tudjuk megoldani. Ezen kívül meg kell határozzuk a függvény által alkalmazott keresőtáblákat. A 7.21. ábra egy szegmensenkénti lineáris approximációval megközelített példát mutat be, így az értékek egy 2 dimenziós keresőtáblában vannak eltárolva. A tömbnek 2 sora van az első sor az (7.16) egyenletben bemutatott függvény a paramétereit tárolja minden egyes szegmensre, míg a második sor a b paramétereket):

- rbf_input: ez a paraméter segítségével határozzuk meg a háló bemeneti rétegének a méretét.
- rbf_number: ezzel az értékkel határozzuk meg hogy hány bázisfüggvénnyel rendelkezik a neurális háló.
- rbf_output: ez a paraméter a kimenetek számát adja meg.

A három konstanst egy 2 dimenziós tömb követi a súlytényezőkkel. A 7.20. ábra csak az első 3 súlytényezőt mutassa, a többi az „…” van jelölve.

```
/** NETWORK */
const unsigned rbf_input = 2;    /// Number of inputs.
const unsigned rbf_number = 100;   // Number of base functions.
const unsigned rbf_output = 1; /// Number of outputs.
const MY_FIXED W_rbf[rbf_output][rbf_number] = {
    {0.41792689692586543, 0.2923732896245581, 0.45377059993667834, ...
};
```

7.20. ábra: Példa RBF háló topológia beállítására.

Az alkalmazott bázisfüggvényt egy „BASE_FUNCTION” makró beállításával tudjuk megoldani. Ezen kívül meg kell határozzuk a függvény által alkalmazott keresőtáblákat. A 7.21. ábra egy szegmensenkénti lineáris approximációval megközelített példát mutat be, így az értékek egy 2 dimenziós keresőtáblában vannak eltárolva. A tömbnek 2 sora van az első sor az (7.16) egyenletben bemutatott függvény a paramétereit tárolja minden egyes szegmensre, míg a második sor a b paramétereket. A „segments” konstans az meghatározza, hogy az adott aktivációs függvény hány szegmensre volt bontva. A bázisfüggvények középpontjait az „rbf_center” tömb tartalmazza, minden sor egy középpontnak felel meg. Mivel e2 bemenetellel rendelkező hálóról van szó ezért a középpontok is 2 dimenzióba vannak meghatározva, azaz 2 koordináta ponttal vannak leírva.

```
/** BASE FUNCTION */
const MY_FIXED rbf_center[rbf_number][rbf_input] = {
    {-3.0, -3.0},
    {-3.0, -2.3333333333333335},
    ...
};
#define BASE_FUNCTION gauss
const unsigned segments = 4;
const MY_FIXED gauss_lut[2][segments] = {
    {-0.41196085515265213, -0.4776893415300143, -0.11861486988468443, -0.009505468935339526},
    {1.0616002928877966, 1.057196054957734, 0.3501809808210514, 0.0365737560390326}
};
```

7.21. ábra: Példa RBF háló aktivációs függvényének beállítására.

Végül pedig a direktíva konfiguráció van meghatározva. Az alábbi ábrán (7.22. ábra) látható, hogy 4 direktíva makró van kiemelve. Az alábbi konfigurációban egy neuron kiszámítsa az csővezetékezve van, míg a neuronok párhuzamosan vannak kiszámítva. Amint a 7.20. ábra mutatta, ennek a neurális hálónak csupán 1 kimenete van, így az az „unroll” párhuzamosítás az elhanyagolható, mivel nem fog se gyorsítást és se plusz erőforrás szükségletet behozni.

```
/** DIRECTIVES */
#define GAUSS_DIRECTIVE
#define VECTOR_MULT_DIRECTIVE _Pragma("HLS pipelin");
#define BASE_FUNCTIONS_DIRECTIVE VECTOR_MULT_DIRECTIVE
#define NEURON_DIRECTIVE _Pragma("HLS unroll");
```

7.22. ábra: Példa RBF háló direktíva konfigurációjának a beállítására.

8. Üzembe helyezés és kísérleti eredmények

8.1. Üzembe helyezés

8.1.1. Python környezet

A Python környezet felkonfigurálása az nagyon egyszerű. Javasolt egy virtuális Python környezet létrehozása, ezáltal a telepítendő csomagok verziója nem fogja befolyásolni a teljes rendszert, csak a keretrendszert. A keretrendszer mappájában található egy „requirements.txt” nevű állomány, ami tartalmazza a szükséges könyvtárak nevét és verzióját, így a könyvtárak telepítése egy parancssorral megoldható, mint a 8.1. ábra mutatja.

A konfigurálási folyamatot nagyon meg tudja könnyíteni egy integrált fejlesztői környezet (angolul: Integrated Development Enviroment, rövidítése: IDE), mint például Visual Studio Code vagy PyCharm. Az ehhez hasonló IDE-k automatikusan elvégzik az előbb leírt lépéseket.

```
# Creats a virtual python3 enviroment
python3 -m venv venv
# Activate the virtual enviroment
source tutorial-env/bin/activate
# Install packages.
pip install -r requirements.txt
```

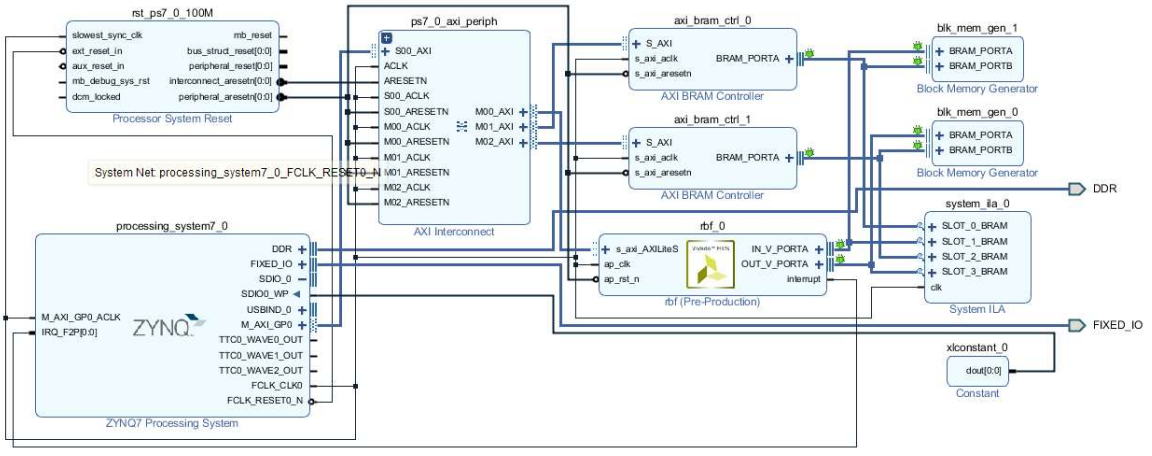
8.1. ábra: Python környezet konfigurálása.

VIVADO HLS

A VIVADO HLS fejlesztői környezetben létre kell hozni egy projektet. A projekt létrehozása során be kell állítani a céleszköz típusát, és hozzá kell csatolni a kigenerált .h és .cpp állományokat. Ezt követően a kigenerált forráskódot szabadon lehet módosítani, vagy akár tesztelni a rendszert funkcionalitás szempontjából egy tesztpad segítségével. Ha ebben a stádiumban teszteljük a neuronháló modult, akkor nem kell szintetizálni, majd leképezni, elhelyezni FPGA-n, ezzel időt lehet spórolni, valamint a hibakeresés is könnyebb, mint valós hardveren. Végül pedig lehet szintetizálni, és kilehet generálni az IP magot.

8.1.2. VIVADO

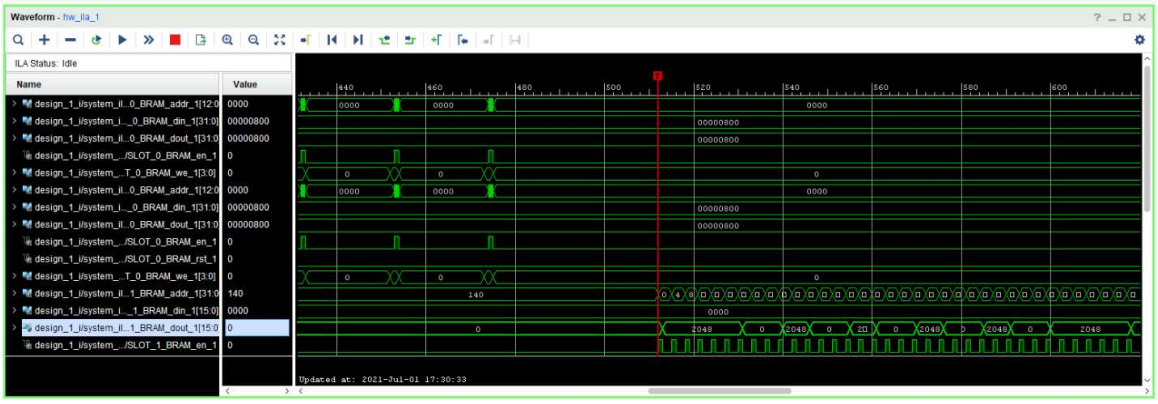
A VIVADO HLS által kigenerált IP magot behelyeztük egy VIVADO-ban megvalósított rendszerbe, Ez a rendszer az IP magon kívül még tartalmaz egy ki- és egy bemenetre alkalmazott BlockRAM-ot, és egy AXI sínrendszert, amin keresztül a komponenseket összekapcsoltuk a fejlesztő lapon található ARM processzorral.



8.2. ábra: A kigenerált rendszer tömbvázlata.

Mivel a neuronháló modul az 16 bitszélességű számokat alkalmaz, és az ARM processzoron és a BlockRAM-ban 32 bites számokat alkalmaztunk, ezért üzembehelyezéskór egy olyan problémában ütköztünk, hogy a neuronháló modul az rossz értékeket olvasott be, és térített vissza eredményül. Az előbbi ábrán (8.2. ábra) látható, hogy a rendszerben be van építve egy integrált

logikai analizátor (angolul: System Integrated Logic Analyzer, rövidítése: ILA) modul. Az előbb említett hardveres hibát az ILA modul segítségével tudtuk megvizsgálni és kijavítani. A következő ábrán (8.3. ábra) láható az ILA modul használat közben. Az analizátort élesítését (angolul: trigger) a bemeneti Dual Port BlockRAM-nak a PortA sínrendszer engedélyező jelére volt kötve. Ez azt jelenti, hogy az ILA modul rögzíteni fogja a neuronháló modul beolvasási folyamatát, amit a 8.3. ábra mutat. A legutolsó sor az engedélyező jel, amire volt élesítve az analizátor, a fölötte levő jel az érték amit kapott a neuronháló modul a BlockRAM-tól, lentről a negyedik jel az a cím ahonnan a neuronháló akarja olvasni a bemeneti értékeket.

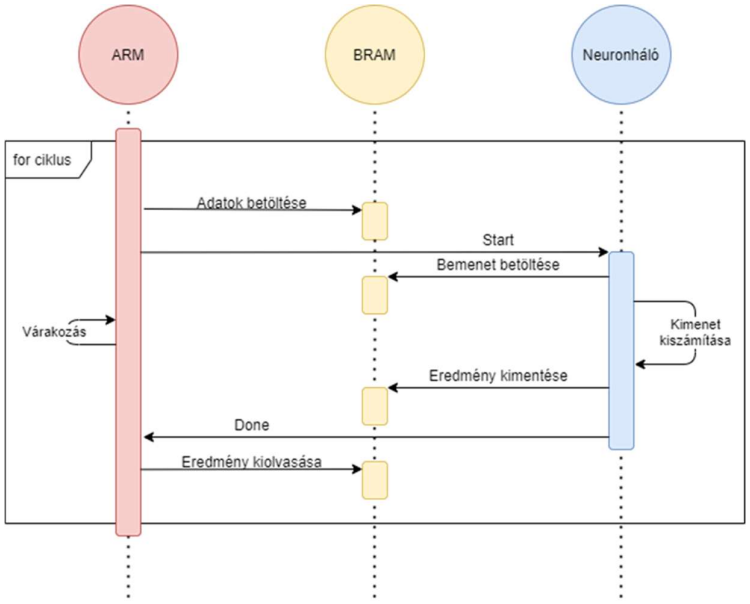


8.3. ábra: Hiba keresés az integrált logikai analizátorral.

8.1.3. VIVADO SDK

Az FPGA-n megvalósított rendszer tesztelésére a Zybo Zynq-7000 FPGA fejlesztőlapon lapon található ARM mikroprocesszort alkalmaztuk. A beágyazott alkalmazás tervezése VIVADO SDK-val készült.

Az ARM magon futtatott alkalmazás az előre definiált bemeneteket egy AXI sínrendszerre kötött BlockRAM vezérlőn keresztül betöltötte egy BlockRAM-ba, majd ugyanezen a sínrendszeren keresztül elküldte neurális háló modulnak a „start” jelzést. Az alkalmazás figyeli a neurális háló modul állapotát és tevékeny várakozással várja míg neuronháló modul elküldi a „ap_done” jelzést. Közben a neurális háló modul betölti a BlockRAM-ból a bemeneti értékeket, kiszámolja a kimenetet, és kiírja egy másik BlockRAM-ba az eredményt. Végül a „done” státusz regiszteren és megszakításon keresztül jelzi, hogy befejezte a műveletet.



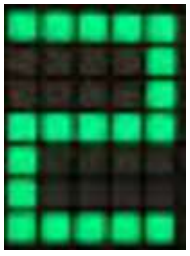
8.4. ábra: Tesztelő algoritmus szekvencia diagramja.

Az alkalmazás segítségével kapott eredményeket össze tudjuk hasonlítani a Python modell által kapott eredményeket, így meg tudjuk határozni a HLS modellben alkalmazott 16 bitszélességű fixpontos számábrázolás és a különböző approximációs módszerek által behozott pontosságvesztést a Python modellben alkalmazott 64 bites lebegőpontos számábrázolás és numerikus módszerekhez képest.

8.2. Hálók tanítása

8.2.1. 7×5-es számjegyek felismerése

A feladat egy egyszerű osztályozási feladat. A háló bemenetet egy 7×5 mátrix képezi, ami egy számjegyet reprezentál. A kimenet az „one hot” kódolást követi, ami annyit jelent, hogy annak a neuronnak a kimenete lesz 1-es, amelyik osztályba tartozik a bemenet, valamint a többi neuron a kimenete az 0 lesz. Erre egy egyszerű példa, a tekintsük a 2-es számjegyet, ami a következő ábrán látható (8.5. ábra):



$$= \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \textit{bemenetre az elvárt kimenet az} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

8.5. ábra: Példa bemenetre és az elvárt eredményre.

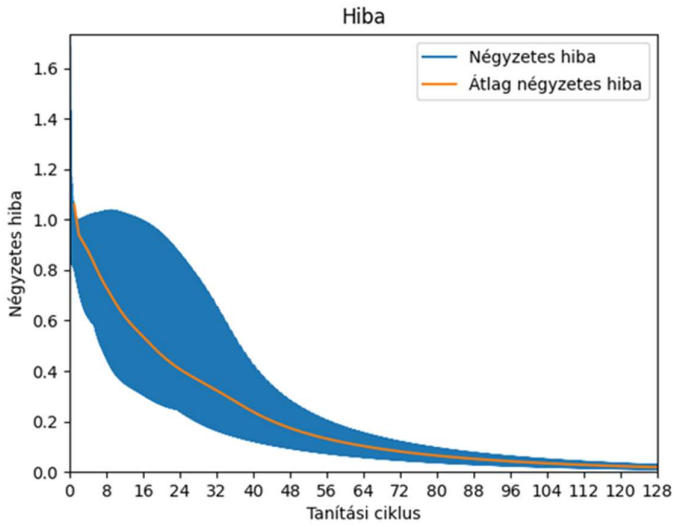
A feladat megoldására 3 topológia volt készítve, mindhárom topológiának 35 bemenete és 10 kimenete volt:

- Egy 2 rétegűt.
- Egy 3 rétegűt: tartalmaz egy 20 neuronból álló rejtett réteget.
- Egy 4 rétegűt: tartalmaz egy 32 és egy 20 neuronból álló rejtett réteget.

Mindhárom topológiát ugyanazokkal a tanítási paraméterekkel tanítottuk:

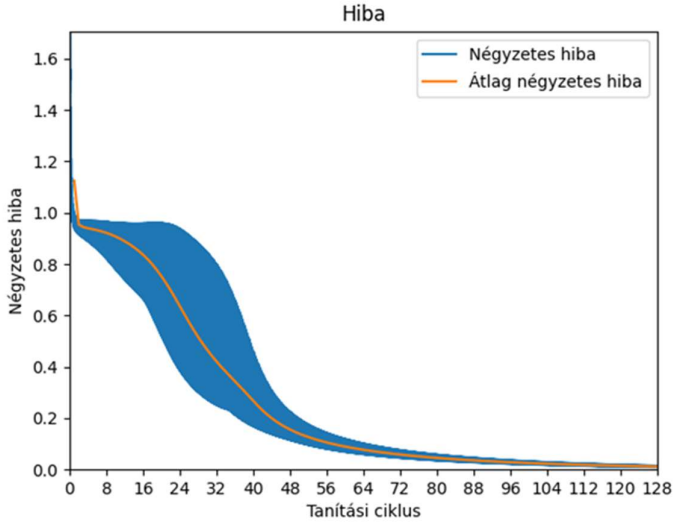
- Tanítási ciklusok száma: 128.
- Kiinduló tanítási együttható (u): 0,08.
- Kiinduló momentum együttható (m): 0,005.
- Batch mérete: 5

A 2 rétegű háló 385 paraméterének a tanítása megközelítőleg 20 másodpercbe telt. A betanított háló átlag négyzetes hibája 0,018720. A költségfüggvény alakulását a 8.6. ábra mutatja.



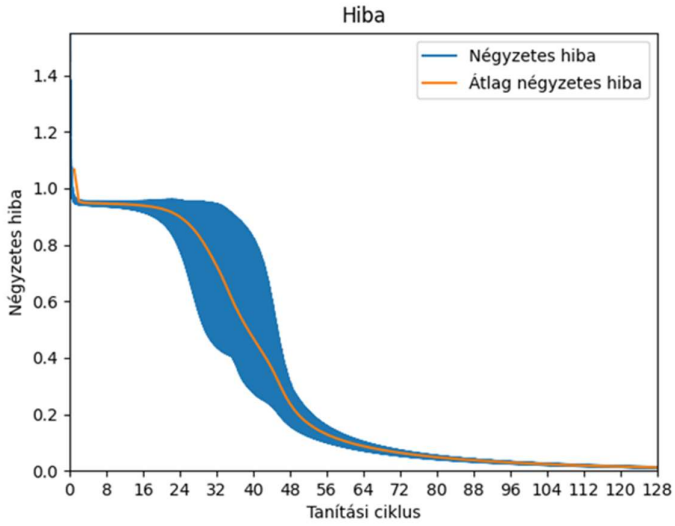
8.6. ábra: Tanítás (35-10).

A 3 rétegű háló 955 paraméterének a tanítása megközelítőleg 21 másodpercbe telt. A betanított háló átlag négyzetes hibája 0,011832. A költségfüggvény alakulását a 8.7. ábra mutatja.



8.7. ábra: Tanitás (35-20-10).

A 4 rétegű háló 2047 paraméterének a tanítása megközelítőleg 23 másodpercbe telt. A betanított háló átlag négyzetes hibája 0,011862. A költségfüggvény alakulását a 8.8. ábra mutatja.



8.8. ábra: Tanítás (35-32-20-10).

8.2.2. MNIST adathalmaz

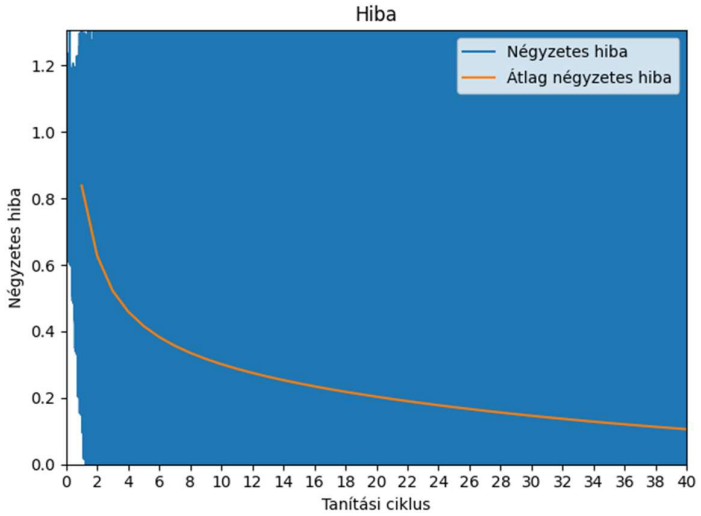
A keretrendszert ki volt próbálva a nyilvánosan elérhető MNIST számjegy adathalmazon. Ez az adathalmaz 28×28 pixelből álló számjegyeket tartalmaz. Az adathalmazt azt 6:1 arányba alkalmaztam, azaz 60000 mintát használtam tanításra és a maradék 10000 elemet a tesztelésre. A kipróbált architektúra az egy 4 rétegű perceptron háló volt:

- 784 elemből álló bemenet (28×28)
- 512 neuront tartalmazó rejtett réteg egyszerű ReLU aktivációs függvénnyel

- 64 neuront tartalmazó rejtett réteg egyszerű ReLU aktivációs függvénnyel
- 10 neuront tartalmazó kimeneti réteg ReLU aktivációs függvénnyel

Az MNIST adathalmaz 2 részre van bontva: 60000 elemet tartalmazó halmaz a tanításra és 10000 elemből álló halmaz a tesztelésre. Az adathalmazok normalizálva voltak, azaz a $[0, 255]$ intervallumból $(0, 1]$ intervallumra volt leképezve, a következő képlet alapján:

$$x = x \cdot \frac{0.9}{255} + 0.1 \tag{8.1}$$



8.9. ábra: MNIST adathalmaz tanítása.

A tanítás során a következő paraméterek voltak használva:

- Tanítási ciklusok száma: 40
- Kezdeti tanítási együttható: 0,0001
- Kezdeti momentum együttható: 0,01
- Egy batch mérete: 50

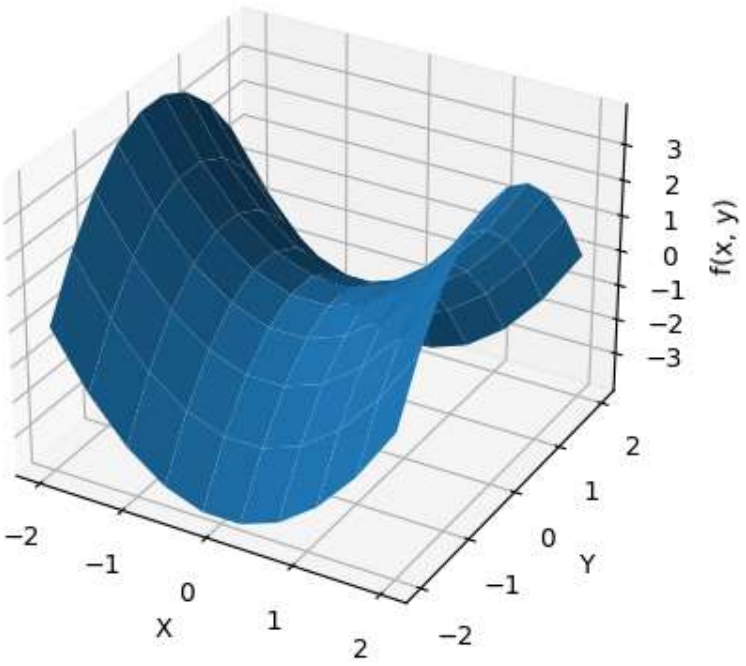
A háló tanítás a fenti paraméterekkel 1015.42 másodpercbe telt, azaz megközelítőleg 17 percre volt szükség. A betanított háló pontossága a teszt halmazon 96.65% volt.

8.2.3. Két változós függvény interpolálása

Adott a következő két változós függvény:

$$f(x, y) = x^2 - y^2 \tag{8.2}$$

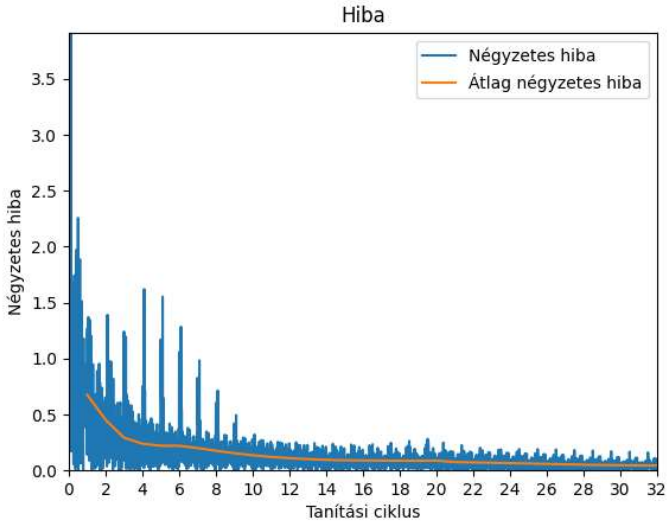
A tanítóhalmazt 100 mintavétel (8.10. ábra) képezi a (8.2 egyenletet x, y ∈ [-2, 2] intervallumából. A teszt halmazt (8.12. ábra) más 289 pont jelképezi ugyan azon az intervallumon.



8.10. ábra: A 2 változós függvény grafikus képe (tanító halmaz).

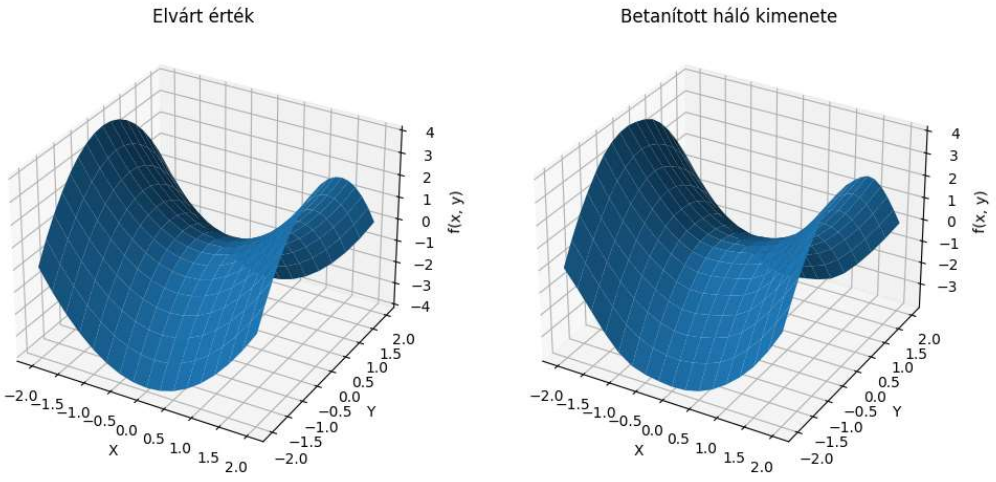
Az RBF háló rejtett rétegét 100 bázisfüggvény alkotja. A tanítás során a következő paraméterek voltak alkalmazva:

- Tanítási ciklusok száma: 32.
- Kiinduló tanítási együttható (u): 0,1.
- Kiinduló momentum együttható (m): 0,1.



8.11. ábra: RBF háló költségfüggvényének fejlődése.

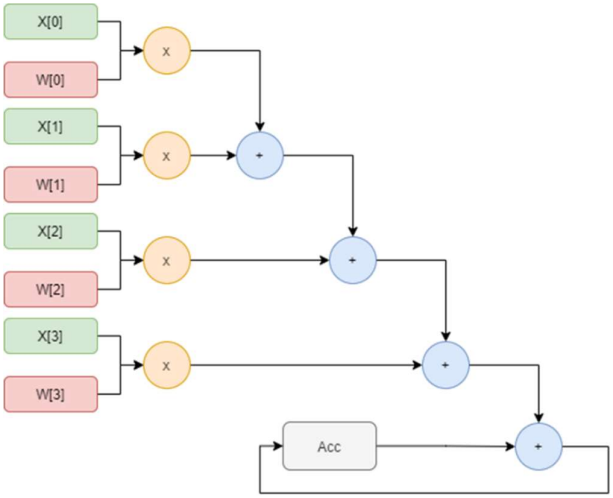
A tanítás folyamata megközelítőleg 31 másodpercbe telt. A betanított háló átlag négyzetes hibája a tanítóhalmazon 0,05798. A költségfüggvény fejlődését a 8.11. ábra mutatja.
A alábbi ábrán (8.12. ábra) látható a teszthalmaz elvárt értéke és a betanított RBF háló felelete a teszthalmazban levő pontokra.



8.12. ábra: Elvárt kimnenet (baloldali) és a betanított háló kimenete (jóbboldali)

8.3. HLS generálása

A 7×5 számjegyek osztályozására betanított hálókat különböző direktíva kombinációra generáltam ki. Az alábbi táblázatban (8.1. táblázat) egy minimálisan párhuzamosított konfiguráció erőforrás becslése látható. Egy neuronnak 4 bemenetének az súlyzóval való összeszorzása párhuzamosan végződik el (8.13. ábra).



8.13. ábra: Egy neuron 4-es faktorral való párhuzamosítása.

Látható, hogy ha az aktivációs függvény szegmensenként lineáris függvénnyel közelítjük meg, akkor eggyel több DSP-re van szükség.

8.1. táblázat: Erőforrás becslés egy neuron 4-es faktorral való párhuzamosításánál. [21]

Architektúra	BlockRAM	DSP	Flip-Flop	LUT	Késleltetés
35-10 (LUT)	0	4	1505	19951	73,29 µs
35-20-10 (LUT)	4	4	3145	21264	145,34 µs
35-32-20-10 (LUT)	6	4	2902	22251	217,37 µs
35-10 (LUT lin. approximációval)	2	5	1512	3332	39,45 µs
35-20-10 (LUT lin. approx.)	4	5	2431	4345	77,66 µs
35-32-20-10 (LUT lin. approx.)	6	5	2432	4447	115,85 µs

Ha a rendszert jobban szeretnénk párhuzamosítani akkor van lehetőség egy neuron teljes kiforgatására, azaz szorzások azok teljesen párhuzamosan lesznek végrehajtva. Ebben az esetben annyi DSP-re lesz szükség amennyi elem van a legnagyobb rétegben. A párhuzamosításnak köszönhetően a késleltetés az felére csökkent viszont jóval megnőtt az erőforrás igény (4 DSP helyet 35 DSP-re lesz szükség).

8. . táblázat: Erőforrás becslés egy neuron teljes kicsavarással való párhuzamosításánál [21].

Architektúra	BlockRAM	DSP	Flip-Flop	LUT	Késleltetés
35-10 (LUT)	0	35	4293	36838	39,45 µs
35-20-10 (LUT)	0	35	5933	28757	82,70 µs
35-32-20-10 (LUT)	6	35	5424	30324	129,89 µs
35-10 (LUT lin. approximációval)	0	36	4319	28757	25,41 µs
35-20-10 (LUT lin. approx.)	0	36	5566	12210	53,90 µs
35-20-16-10 (LUT lin. approx.)	6	36	5057	13780	86,69 µs

A rendszer sebességén még lehet javítani, ha a végrehajtást csővezetékesítjük. Ez viszont még több erőforrást igényel (majdnem 3-szor több Flip-Flop ra lesz szükség). Sajnos ebben a konfigurációban a három rétegű architektúrának nincs elegendő erőforrás az alkalmazott Zybo Zynq-7000 FPGA fejlesztőlapon.

8.2. táblázat: Erőforrás becslés egy neuron teljes párhuzamosításával és csövezetékesítésével való párhuzamosításánál [21].

Architektúra	BlockRAM	DSP	Flip-Flop	LUT	Késleltetés
35-10 (LUT)	0	35	8854	28829	2,63 μs
35-20-10 (LUT)	0	35	10085	30333	4,04 μs
35-32-20-10 (LUT)	108	35	9427	31346	5,48 μs
35-10 (LUT lin. approximációval)	0	36	7774	12262	2,23 μs
35-20-10 (LUT lin. approx.)	0	36	9109	13765	3,24 μs
35-32-20-10 (LUT lin. approx.)	108	36	8451	14776	4,28 μs

8.3. táblázat: RBF háló erőforrás becslése különböző párhuzamosításokra. [21].

Párhuzamosítás	BlockRAM	DSP	Flip-Flop	LUT	Késleletetés
neuron 4-es faktorral párhuzamosítva	1	12	2764	14890	15,33 μs
neuron csővezetékezve	1	4	2326	4259	2,19 μs

8.4. Hardver generálása

Miután a VIVADO környezet szintetizálja, leképezi, elhelyezi és huzalozza a rendszert, akkor pontosan tudni lehet, hogy mennyi erőforrást (BlockRAM, DSP, LUT, FlipFlop) alkalmaz. Ugyanakkor megbecsülhető az is, hogy menyi lesz a kialakított rendszer energia fogyasztása. Az összespárhuzamosítási konfigurációban az energia szükséglet az 1,75W és 1,85W között volt. Ez lényegesen kisebb, mint a i7-7500U processzor 15W teljesítménye. Viszont fontos megemlíteni, hogy a processzor a Pythonban megírt modell mellet még futott egy operációs rendszer és grafikus felület is.

A 8.45. táblázatban a párhuzamosítási konfigurációk számokkal vannak jelölve, amelyek a következő konfigurációkat képviselik:

1. Egy neuron 4-es faktorral való párhuzamosítása (8.13. ábra).
2. Egy neuron teljes kicsavarással való párhuzamosítása.
3. Egy neuron teljes kicsavarással való párhuzamosítása és a neuronok egymás utáni kiszámítása csővezetékezve.
4. A bázisfüggvények és neuronok kiszámítása csővezetékezve

8.45. táblázat: Errőforrásszükségletek összesítése.

Architektúra	Párhuzamosítás	BRAM	DSP	FF	LUT	LUTRAM
35-10 (LUT)	1	13,5	4	8402	6426	585
	2	12,5	35	10115	7394	585
	3	12,5	35	10115	7394	585
35-10 (LUT lin. a.)	1	13.5	5	7702	5322	585
	2	12,5	36	10129	6544	585
	3	12,5	36	10936	7149	1292
35-20-10 (LUT)	1	14,5	4	9296	7464	585
	2	12,5	35	13401	13001	585
	3	12,5	35	13401	13001	585
35-20-10 (LUT lin. a.)	1	14,5	5	8603	5872	585
	2	12,5	36	11396	7810	585
	3	12,5	36	12195	8317	1295
35-32-20-10 (LUT)	1	15,5	4	9203	7190	585
	2	15,5	35	14379	10599	585
35-32-20-10 (LUT l. a.)	1	15,5	5	8604	5878	585
	2	15,5	36	10834	7074	585
RBF	1	13	12	7374	4635	468
	4	13	4	6867	4103	474

8.5. A hardver tesztelése

A VIVADO SDK nagyon megkönnyíti a kialakított neuronháló modul megvezérlését és tesztelését. A VIVADO SDK elemzi a kigenerált hardvert és az alapján kisegítő header állományokat generál, mint például az alábbi „xmlp_hw.h” (8.14. ábra). Látható, hogy feltérképezte a neuronháló modul AXI interfészét, valamint kigenerálta a szükséges konstansokat. Ugyanakkor azt is meghatározta, hogy az adott címeken melyik bitérték milyen funkcionalitásért feleős. Például a „XMLP_AXILITES_ADDR_AP_CTRL” címen, ha a nulladik bitértéket egyesre állítjuk akkor el tudjuk indítani a neuronháló modult. Valamint az egyes biten keresztül ki tudjuk olvasni, ha a neuronháló modul befejezte a kimenet kiszámítását. A kigenerált dokumentációban az is meg van jelölve, hogy ez az „ap_done” status regiszter COR tulajdonsággal rendelkezik, azaz olvasás után automatikusan lenullázza magát.

```
// ==========================================================
// File generated by Vivado(TM) HLS - High-Level Synthesis from C, C++ and SystemC
// Version: 2017.4
// Copyright (C) 1986-2017 Xilinx, Inc. All Rights Reserved.
//
// ==========================================================

// AXILiteS
// 0x0 : Control signals
//       bit 0  - ap_start (Read/Write/COH)
//       bit 1  - ap_done (Read/COR)
//       bit 2  - ap_idle (Read)
//       bit 3  - ap_ready (Read)
//       bit 7  - auto_restart (Read/Write)
//       others - reserved
// 0x4 : Global Interrupt Enable Register
//       bit 0  - Global Interrupt Enable (Read/Write)
//       others - reserved
// 0x8 : IP Interrupt Enable Register (Read/Write)
//       bit 0  - Channel 0 (ap_done)
//       bit 1  - Channel 1 (ap_ready)
//       others - reserved
// 0xc : IP Interrupt Status Register (Read/TOW)
//       bit 0  - Channel 0 (ap_done)
//       bit 1  - Channel 1 (ap_ready)
//       others - reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)

#define XMLP_AXILITES_ADDR_AP_CTRL 0x0
#define XMLP_AXILITES_ADDR_GIE     0x4
#define XMLP_AXILITES_ADDR_IER     0x8
#define XMLP_AXILITES_ADDR_ISR     0xc
```

8.14. ábra: Xilinx VIVADO SDK által generált vezérlő jelek és dokumentáció.

A 8.15. ábra a neuronháló modul elindítását és „ap_done” jelzés várakozását szemlélteti alkalmazva az SDK által kigenerált makrókat.

```
    Xil_Out32(XPAR_MLP_0_S_AXI_AXILITES_BASEADDR + XMLP_AXILITES_ADDR_AP_CTRL, 0x01);
    while (!(Xil_In32(XPAR_MLP_0_S_AXI_AXILITES_BASEADDR + XMLP_AXILITES_ADDR_AP_CTRL) & 0x02)) {
        usleep(1000);
    }
```

8.15. ábra: A neuronháló modul elindítása és a done jelzés megvárása (kódrészlet).

Az FPGA-n futtatott háló kimeneteit összehasonlítva a Pythonban futatott háló kimenetével (8.6. táblázat) kiderül, hogy az eltérés az minimális. Egy észrevétel, hogy az MLP háló HLS modelltől kapott eredmények négyzetes hibájának átlaga egy kicsivel kisebb, mint a Pythonban megvalósított modellé. Ez annak tudható be, hogy mivel az aktivációs függvényeket megközelítettük, ezért a kereső tábla csak egy adott szakaszra vonatkozik, azaz, ha az adott bemenet egy szélsőséges értk, ami ezen a szakaszon kívül esik, akkor az aktivációs függvény kimenete az a végletnek megfelelő konstans, ebben az esetben 0 vagy 1. Így míg az FPGA-n futó

neuronháló 1-et adót egy adott kimenetre a Pythonban alkalmazott numerikus módszerek egy 1-hez közeli értéket, de nem 1-et.

8.6. táblázat: Az FPGA-n futatott hálók pontossága [21].

Architektúra	Hiba Pythonba	Hiba FPGA-n	Eltérés
35-10 (LUT)	0.0181702715228	0.0168254807722	0.006379091611
35-10 (LUT lin. approx.)		0.0135097706861	0.0151719939283
35-20-10 (LUT)	0.0115919407794	0.0078123999999	0.0085378733850
35-20-10 (LUT lin. approx.)		0.0044726970588	0.0146467258527
35-32-20-10 (LUT)	0.0116205461987	0.0079588000000	0.0085644857908
35-32-20-10 (LUT lin. a.)		0.0048744011284	0.0149282454851
RBF	0.0609929190201	0.0663894673356	0.0225502111950

Ha össze akarjuk hasonlítani a Python modell késeltetését az i7-7500U és Zybo Zynq-7000 FPGA fejlesztőlapon futatott neuronháló késeltetését, akkor, amint a 8.7. táblázat értékei mutatják, akár 20-30-szoros gyorsulást is megfigyelhetünk.
A táblázatban a 8.4 fejezetben definiált párhuzamosítási szintek vannak feltüntetve

8.7. táblázat: FPGA-n futtatott hálók késleltetése.

Architektúra	**Tanítható paraméterek száma**	**Tanítási idő (s)**	**Futás idő Pythonban (µs)**	**Futás idő FPGA-n (LUT) (µs)**			**Futás idő FPGA-n (lin. approx.) (µs)**		
Párhuzam-osítási szint	**-**			**1**	**2**	**3**	**1**	**2**	**3**
35-10	47	20	87	74	40	4	40	26	3
35-20-10	68	21	150	146	83	6	78	54	4
35-32-20-10	101	23	162	218	129	-	116	87	-
Párhuzam-osítási szint	**-**			**1**	**4**		**1**	**4**	
100-1 (RBF)	100	37	562	-	-		16	3	

9. Összefoglalás

A megvalósított keretrendszert egy sikeres projektnek tekintem. A bemutatott keretrendszer, képes neuronhálókat FPGA-ra fordíthatóvá tenni. Két modell volt alkalmazva: egy Pythonban megírt

modell, ami lebegőpontos számábrázolást alkalmazott, és egy HLS-ben megvalósított modell, ami fix pontos számábrázolást alkalmaz, valamint a nemlineáris transzformációkat LUT-okból olvassa ki az erőforrásszükséglet csökkentése és a teljesítmény növelése érdekében. Látható, hogy a két modell között az átjárás az minimális veszteséggel történik, cserébe kevesebb erőforrásra van szükség, valamint a teljesítmény is megnövekszik. Direktívák segítségével a létrehozott rendszer könnyeden testre szabható így alkalmazható egyaránt kisebb és nagyobb FPGA alapú rendszereken is.

9.1. Megvalósítások

A keretrendszer jelenleg két neuronháló modellt támogat: MLP és RBF. Ezeknek a neurális hálóknak a különböző almoduljai Pythonban vannak implementálva, illetve Numba könyvtár segítségével futásidőben gépi kódra fordítva. Az MLP háló esetén különböző aktivációs függvények voltak kialakítva. A keretrendszer biztosít átjárást a Pythonban megvalósított modell és a HLS-ben implementált modell között:

- Neuronháló topológia olyan módon van formázva, hogy a Balázs által megvalósított HLS implementáció tudja értelmezni.
- A súly tényezők azok egy nagy konstans tömbként lesznek beépítve a HLS modellbe.
- Az aktivációs függvények azok szükség esetén egy egyszerű kereső tábla vagy szegmensenkénti lineáris approximáció formájában lesznek a beillesztve a HLS-ben kialakított modellbe.

A kigenerált neuronháló modult beillesztettük egy egyszerű rendszerbe, amit ált egy bemeneti- és egy kimeneti BlockRAM-ból, majd a fejlesztőlapon található ARM processzor segítségével megvezéreltük. Így a rendszer valós hardveren is ki volt próbálva. A valós rendszeren kapott eredmények össze voltak hasonlítva a Pythonban megvalósított modell által adott eredményekkel.

9.2. Következtetések

Ha az általunk kialakított rendszert más magas szintű keretrendszerekkel szeretnénk összehasonlítani, akkor a következőket jelenthetjük ki:

- Az általunk javasolt rendszer egyaránt alkalmazható számítógépen és FPGA-s rendszereken való modellezésre is, míg TensorFlow, PyThorch, stb. csak számítógépen, illetve dedikált gyorsítókon.
- Viszont hátrányként meg kell említenünk, hogy az általunk kialakított rendszernek a számítógépen futó modellje Pythonban van megvalósítva. Annak ellenére, hogy különböző könyvtárak (Numpy és Numba) segítségével próbáltuk növelni a rendszer sebességét, nem

lesz olyan gyors, mint ha a keretrendszert egy C/C++ (vagy más alacsonyabb szintű) programozási nyelvben megvalósított könyvtárra építenénk.

- Egy másik nagy hiányosága az általunk megvalósított keretrendszernek, hogy csak két neurális háló modellt támogat (MLP és RBF), míg a többi magasszintű keretrendszer sokkal több modell kialakítására alkalmazható és sokkal nagyobb szabadságfokot biztosít új modellekkel való tesztelésre.

9.1. táblázat: Összehasonlítás más magas szintű keretrendszerrel.

	A bemutatott keretrendszer	**Más magas szintű keretrendszerek**
Előny	• Képes CPU-n és FPGA-n is modellezni a neurális hálókat	• Nagyon sok neuronháló modellt támogatnak • Általában C/C++ könyvtárra épülnek
Hátrány	• Csak az MLP és RBF modellt támogatja • Pythonban volt implementálva	• Csak CPU-n és GPU-n modellezik a neurális hálókat

Ha az általunk kialakított rendszert olyan keretrendszerekkel hasonlítjuk össze, amelyek a magas szintű modellt FPGA rendszerekre is ki tudják generálni, mint a LeFlow, akkor a következő következtetéseket vonhatjuk le (lásd 9.2. táblázat):

- Az általunk javasolt megközelítés, hogy a számítógépen alkalmazott lebegőpontos számábrázolást cseréljük le fixpontosra.
- A bemutatott keretrendszer előnye, hogy nem CPU-ra optimalizált numerikus megközelítéseket alkalmaz, hanem FPGA-ra tervezett almodulokat, melyek különböző approximációs módszerekkel csökkentik a szükséges erőforrást és növelik a teljesítményt.
- Az általunk megvalósított keretrendszer csak két modellt tud FPGA-ra fordítani, míg a LeFlow egy módosított TensorFlow keretrendszer, így a TensorFlow által támogatott modellek nagy részét képes FPGA-n modellezni.
- Egyaránt előnyként és hátrányként kell beszéljünk arról, hogy a mi keretrendszerünk két eltérő modellt alkalmaz. Előny, mert így mindkét rendszeren (CPU és FPGA) egyaránt lehet optimalizálni. Hátrány olyan szempontból, hogy fejlesztés során nem elegendő csak az egyik rendszert átlátni, hiszen a két modellt szinkronban kell tartani.

9.2. táblázat: Összehasonlítás LeFlow keretrendszerrel.

	A bemutatott keretrendszer	LeFlow
Előny	• FPGA-n támogatja a fixpontos számábrázolást is • FPGA-ra tervezett approximációs módszereket alkalmaz	• A TensorFlow által alkalalamzott modellekre épít • Sokkal több modellt támogat
Hátrány	• Csak az MLP és RBF modellt támogatja • Nem egy elterjedt keretrendszerre épül	• FPGA-n is csak a lebegő pontos számábrázolást alkalmazza

9.3. Lehetőségek a továbbfejlesztésre

- A fordítási és tesztelési folyamat teljes automatizálása.
- Több modell és neurálisháló típus implementálása, valamint más approximációs megközelítések vizsgálása.
- Dinamikusabb és nagyobb szabadságfokkal rendelkező sablonok: jelenleg a rendszer több statikus sablon alapján generálja a HLS forráskódot. Ezt e jövőre nézve lehetne egyesíteni és dinamikusabbá tenni, ezzel nagyobb szabadságfokot lehetne biztosítani a felhasználónak.
- Más keretrendszerek modelljeiből generálni a HLS-t: Jelenleg egy általunk megírt keretrendszerrel tanítjuk a hálót, vagy egy JSON állomány (ezt a rendszer generálja mentés során) segítségével tudjuk betölteni a paramétereket.

9.4. Köszönetnyilvánítás

Köszönetet szeretnék mondani vezetőtanáromnak, dr. ing. Brassai Sándor Tihamérnek, a Sapientia Erdély Magyar Tudományegyetem docensének, akinek a segítsége és útbaigazítása nélkül nem valósulhatott volna meg ez a dolgozat. Még meg szeretném köszönni a Sapientia EMTE Marosvásárhelyi Karának, a rendszergazdáknak és Székely István Zsolt labortechnikusnak, hogy annak ellenére, hogy az oktatás online történt, lehetővé tették, hogy távolról hozzáférjünk a mesterséges intelligencia laboratóriumban található számítógépekhez, mérőeszközökhöz és FPGA lapokhoz. Köszönetet szeretnék mondani az Accenture cégnek és a Magyar Külgazdasági és Külügyminisztériumnak, hogy támogatták a kutatást az „Accenture Student Research Scholarship 2020”, valamint a Balassi Bálint Ösztöndíjprogram Klebelsberg Kuno Tehetséggondozó Ösztöndíj keretén belül.

10. Irodalomjegyzék

[1] J. J. Hopfield, "Artificial neural networks," *IEEE Circuits and Devices Magazine,* vol. 4, pp. 3-10, 1988.

[2] M. Van Gerven and S. Bohte, "Artificial neural networks as models of neural information processing," *Frontiers in Computational Neuroscience,* vol. 11, p. 114, 2017.

[3] A. Shatnawi, G. Al-Bdour, R. Al-Qurran and M. Al-Ayyoub, "A comparative study of open source deep learning frameworks," in *2018 9th international conference on information and communication systems (icics)*, IEEE, 2018, pp. 72-77.

[4] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani and S. Chilamkurthy, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," *arXiv preprint arXiv:1912.01703,* 2019.

[5] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev and J. a. G. R. a. G. S. a. D. T. Long, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 675-678.

[6] Team, T. T. Development, R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau and N. B. e. al, "Theano: A Python framework for fast computation of mathematical expressions.," *arXiv preprint arXiv:1605.02688,* 2016.

[7] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard and others, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265-283.

[8] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang and J. Cong, "FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2017, pp. 152-159.

[9] D. H. Noronha, B. Salehpour and S. J. Wilton, "LeFlow: Enabling flexible FPGA high-level synthesis of tensorflow deep neural networks," *ArXiv 1807.05317,* 2018.

[10] "Wikipedia," 20 06 2021. [Online]. Available: https://en.wikipedia.org/wiki/Python_(programming_language). [Accessed 23 06 2021].

[11] T. E. Oliphant, "Python for scientific computing.," *Computing in Science & Engineering,* vol. 9, no. 3, pp. 10-20, 2007.

[12] K. Georgopoulos, G. Chrysos, P. Malakonakis, A. Nikitakis, N. Tampouratzis, A. Dollas, D. Pnevmatikatos and Y. Papaefstathiou, "An evaluation of vivado HLS for efficient system design," in *2016 International Symposium ELMAR*, IEEE, 2016, pp. 195-199.

[13] „Xilinx Software Development Kit," Xilinx, [Online]. Available: https://www.xilinx.com/products/design-tools/embedded-software/sdk.html. [Hozzáférés dátuma: 25 06 2021].

[14] „Wikipedia," 13 February 2021. [Online]. Available: https://hu.wikipedia.org/wiki/Neur%C3%A1lis_h%C3%A1l%C3%B3zat. [Hozzáférés dátuma: 25 May 2021].

[15] S. T. Brassai, Neurális hálózatok és fuzzy logika, Cluj-Napoca: Scientia, 2019.

[16] S. G. Tzafestas, „The RBF Network," in *Introduction to Mobile Robot Control*, Oxford, Elsevier, 2014, pp. 269-317.

[17] B. Bustya és A. Hammas, „Keretrendszer MLP-háló FPGA-alapú megvalósítására," in *XX. Online Kari Tudományos Diákköri Konferencia*, Marosvásárhely, 2021.

[18] B. Bustya és A. Hammas, „Keretrendszer neurális háló FPGA alapú megvalósítására," in *XXII. Műszaki Tudományos Diákköri Konferencia*, Temesvár, 2021.

[19] V. Saichand, N. Mohankumar, S. Arumugam and N. Mohankumar, "FPGA realization of activation function for artificial neural networks," in *2008 Eighth International Conference on Intelligent Systems Design and Applications*, vol. 3, IEEE, 2008, pp. 159-164.

[20] S. T. Brassai, L. Bako, G. Pana and S. Dan, "Neural control based on RBF network implemented on FPGA," in *2008 11th International Conference on Optimization of Electrical and Electronic Equipment*, IEEE, 2008, pp. 41-46.

[21] A. Hammas, „Mesterséges neurális hálók megvalósítása újrakonfigurálható digitális áramkörökön. Keretrendszer modellezése és kidolgozása.," in *Balassi Bálint − Ösztöndíjprogram − Klebelsberg Kuno Tehetséggondozó Konferencia*, Budapest, 2021.

[22] I. Tsmots, O. Skorokhoda and V. Rabyk, "Hardware implementation of sigmoid activation functions using FPGA," in *2019 IEEE 15th International Conference on the Experience of Designing and Application of CAD Systems (CADSM)*, IEEEE, 2019, pp. 34-38.

[23] M. Panicker and C. Babu, "Efficient FPGA implementation of sigmoid and bipolar sigmoid activation functions for multilayer perceptrons," *IOSR Journal of Engineering,* vol. 2, no. 6, pp. 1352-1356, 2012.

[24] V. Saichand, N. Mohankumar, S. Arumugam and N. Mohankumar, "FPGA realization of activation function for artificial neural networks," in *2008 Eighth International Conference on Intelligent Systems Design and Applications*, vol. 3, IEEE, 2008, pp. 159-164.

UNIVERSITATEA SAPIENTIA DIN CLUJ-NAPOCA
FACULTATEA DE ȘTIINȚE TEHNICE ȘI UMANISTE, TÎRGU-MUREȘ
SPECIALIZAREA CALCULATOARE

Vizat decan

Ș.l. dr. ing Kelemen András

Vizat director departament

Conf. dr. ing. Domokos József