

TDK DOLGOZAT



Sapientia

Erdélyi Magyar

Tudományegyetem

Marosvásárhelyi kar

Villamosmérnöki tanszék

TDK DOLGOZAT

KERETRENDSZER NEURÁLIS HÁLÓ FPGA ALAPÚ MEGVALÓSÍTÁSÁRA.

Szerző(k):

Bustya Balázs

Programtervező informatikus MSc. szak, I. évf.

Eötvös Loránd Tudományegyetem

Hammás Attila

Számítógépes irányítási rendszerek MSc. Szak, II. évf.

Sapientia Erdélyi Magyar Tudományegyetem

Konzulens:

dr. Brassai Sándor Tihamér

egyetemi docens

Sapientia Erdélyi Magyar Tudományegyetem

SAPIENTIA ERDÉLYI MAGYAR TUDOMÁNYEGYETEM

Keretrendszer neurális háló FPGA alapú megvalósítására.

Framework for creating neural networks for FPGA.

Bustya Balázs
Hammas Attila

Konzulens:
Dr. Brassai Sándor Tihamér

Kézirat lezárva: 2023.01.10.

Abstract

There was a surge in the popularity of artificial neural networks (ANN) because of their efficiency in solving problems like image processing, data mining, or classification. Previously the decision-making was only made by humans, but at the moment, these ANN's can make at least as good decisions or even better ones than humans. However, neural networks cannot replace real humans entirely because they are designed by people based on the problem that they need to solve. Hardware accelerators are used for increasing the performance and efficiency of neural networks. 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 with HLS directives. This code can be compiled by Vivado HLS into a hardware descriptive language and packaged as an IP. The Vivado tool then can generate a bit file which can be uploaded onto the FPGA. Based on the measurements performed in HLS and Python, with the same neural network and the same inputs, it can be observed that the acceleration achieved on the FPGA circuit is significant (42 times) in the case of smaller neural networks. For larger neural networks, the acceleration is smaller due to the limited resources of the FPGA circuits.

Keywords: ANN (neural networks), FPGA (Field Programmable Gate Logic Arrays), framework, HLS (High-Level Synthesis)

Kivonat

A neuronhálók nagy népszerűséget szereztek pontosságuknak köszönhetően képfeldolgozásban, adatbányászatban, osztályozási feladatokban, valamint olyan döntéshozó rendszerekben, amelyekben előttük csak az ember volt képes helyes döntéseket hozni. Ugyanakkor a neuronhálók még nem tudják teljesen helyettesíteni az embert, mivel az emberek tervezik meg őket az elvégzendő feladat függvényében. A neuronhálók teljesítményének növelése érdekében hardveres gyorsítókat alkalmaznak. Az utóbbi években rengetek tanulmány készült hardveres gyorsítókról. Az egyik ilyen lehetőség az FPGA áramkörök, melyek legfőbb előnyei a párhuzamosíthatóság, alacsony fogyasztás és rugalmasság. Azonban neuronhálók megvalósítása FPGA-n igencsak időigényes és nehéz feladat, melyet csak tapasztalt szakemberek tudnak elvégezni. Erre a problémára nyújt megoldást a dolgozatban bemutatott neuronháló keretrendszer. A keretrendszer célja az FPGA-n történő neuronháló megvalósításának felgyorsítása, megkönnyítése, a neuronhálók gyorsítása, optimalizálási lehetőségek tanulmányozása méret és teljesítmény szempontjából, valamint a felprogramozási folyamat automatizálása. A keretrendszer C++-ban modellezett neuronhálók kódját generálja melyek hardver megvalósítása direktívák segítségével van optimalizálva. A Vivado HLS eszköz ebből a kódból kigenerálja a neuronháló IP magját. Ez egy Vivadoban kiépített modulba lesz integrálva. A Vivado eszköz generálja az FPGA programozásához szükséges bit fájlt és felprogramozza az áramkört. A HLS-ben és Python-ban, azonos neuronhálóval és azonos bemenetekkel végzett mérések alapján megfigyelhető, hogy az FPGA áramkörön elért gyorsítás jelentős (42-szeres) a kisebb méretű neuronhálók esetében. Nagyobb méretű neuronhálóknál a gyorsítás kisebb mértékű az FPGA áramkörök erőforrásainak korlátossága miatt.

Kulcsszavak: neuronhálók, FPGA áramkör, keretrendszer, magas szintű szintézis

```

\begin{CCSXML}
<ccs2012>
  <concept>
    <concept_id>10010147.10010178</concept_id>
    <concept_desc>Computing methodologies~Artificial intelligence</concept_desc>
    <concept_significance>300</concept_significance>
  </concept>
  <concept>
    <concept_id>10011007.10011006.10011066.10011070</concept_id>
    <concept_desc>Software and its engineering~Application specific development
environments</concept_desc>
    <concept_significance>300</concept_significance>
  </concept>
</ccs2012>
\end{CCSXML}

```

```

\ccsdesc[300]{Computing methodologies~Artificial intelligence}

```

```

\ccsdesc[300]{Software and its engineering~Application specific development environments}

```

Tartalomjegyzék

1.	Bevezető.....	1
2.	Irodalmi áttekintés	2
2.1.	Magas szintű szintézis (HLS).....	2
2.1.1.	Kompilálás (Compilation)	2
2.1.2.	Erőforrás kiosztás (Allocation)	3
2.1.3.	Ütemezés.....	3
2.1.4.	Összekapcsolás (Binding).....	3
2.1.5.	RTL generálása (Generation).....	4
2.1.6.	Vivado HLS	4
2.1.7.	HLS előnyei RTL szintű tervezéshez képest	7
2.1.8.	HLS hátrányai RTL szintű tervezéshez képest	7
2.2.	Hasonló keretrendszerek	7
2.2.1.	PyTorch.....	8
2.2.2.	Theano.....	8
2.2.3.	Caffe.....	9
2.2.4.	TensorFlow	9
2.2.5.	Microsoft Cognitive Toolkit	9
2.2.6.	Keretrendszerek összehasonlítása és alkalmazásai	10
3.	Célkitűzések.....	11
3.1.	A kutatás rövid bemutatása	11
3.2.	A javasolt rendszer elvi architektúrája	12
4.	Módszerek.....	13
5.	Megvalósítás	13
5.1.	Python modell	13
5.1.1.	Aktivációs függvény	14
5.1.2.	Perceptron réteg	15
5.1.3.	Neurális háló	15
5.1.4.	Tanítás.....	16
5.2.	HLS modell	16
5.2.1.	Az inger kiszámolása	16

5.2.2.	A ciklusok kiterjesztése (Unrolling loops)	19
5.2.3.	Ciklusok csővezetékkezése (Loop pipelining).....	21
5.2.4.	Egy réteg felépítése.....	24
5.2.5.	Fix pontos számábrázolás lebegőpontos számábrázolással szemben	25
5.2.6.	Optimális modell.....	29
6.	A rendszer	29
7.	Kitűzött példa feladat.....	31
7.1.	A feladat leírása.....	31
7.2.	A feladat megoldása	32
8.	Eredmények	35
9.	Következtetések	36
10.	Irodalomjegyzék	37

Táblázatok jegyzéke

1. táblázat:	Fixpontos és lebegő pontos számábrázolás összehasonlítása.....	29
2. táblázat:	A modellek tanítási és futási ideje.....	35
3. táblázat:	A modellek négyzetes hibáinak az átlaga.....	35

Ábrajegyzék

1. ábra:	HLS folyamatok [5]	4
2. ábra:	A keretrendszerek funkcionalitásának az összehasonlítása. [12]	10
3. ábra:	LeFlow folyamat ábrája [14].	11
4. ábra:	A javasolt rendszer tömbvázlata.	12
5. ábra:	A rendszer osztály diagramja.....	14
6. ábra:	Küszöb függvény.	14
7. ábra:	ReLU függvény.....	14
8. ábra:	Logisztikus függvény.....	14
9. ábra:	Tangens hiperbolikus függvény.....	14
10. ábra:	Egy réteg ingereinek a mátrix szorzással való feliírása.	15
11. ábra:	Egy inger kiszámítási képlete.	15
12. ábra:	A hiba visszafejtése az aktivációs függvényen.	16
13. ábra:	A réteg súlymátrixának tanítása gradiens módszerrel.	16

14. ábra: A hiba visszafejtése az előző rétegre.	16
15. ábra: Vektor szorzás direktívákkal.....	17
16. ábra: Mérés direktívák használata nélkül.....	18
17. ábra: Erőforrás felhasználás statisztika, direktívák használata nélkül.	18
18. ábra: Vektor szorzás mikroarchitektúra direktívák alkalmazása nélkül	18
19. ábra: Mérés, vektor szorzás kiterjesztve 4-es faktorral.....	19
20. ábra: Erőforrás felhasználási statisztika, vektor szorzás kiterjesztve 4-es faktorral.	20
21. ábra: Mérés, tömbök particionálva.	20
22. ábra: Vektor szorzás mikroarchitektúra, ciklus kiterjesztve, tömbök particionálva.....	21
23. ábra: Mérés, vektor szorzás csővezetékhezve.....	22
24. ábra: Erőforrás felhasználási statisztika, vektor szorzás csővezetékhezve.....	22
25. ábra: Vektor szorzás, ciklus kiterjesztve, tömbök particionálva, ciklus csővezetékhezve. .	23
26. ábra: Egy neuron réteg.	24
27. ábra: Egy neuron.....	24
28. ábra: Mérés, neuronok párhuzamosítva.	25
29. ábra: Erőforrás felhasználási statisztika, neuronok párhuzamosítva.	25
30. ábra: Mérés, 4 bájtos lebegőpontos adattípust használva.	26
31. ábra: Erőforrás felhasználási statisztika, 4 bájtos lebegőpontos adattípust használva.....	26
32. ábra: Mérés, fixpontos<32,16> adattípust használva.....	27
33. ábra: Erőforrás felhasználási statisztika, fixpontos<32,16> adattípust használva.....	27
34. ábra: Mérés, fixpontos<16,8> adattípust használva.....	27
35. ábra: Erőforrás felhasználási statisztika, fixpontos<16,8> adattípust használva.....	28
36. ábra: Erőforrás felhasználási statisztika, fixpontos<12,6> adattípust használva.....	28
37. ábra: A modell korlátai.	29
38. ábra: A rendszer tömbvázlata.....	30
39. ábra: Vivado-ban felépített rendszer.	31
40. ábra: Simulink-ben System Gnerator-al felépített rendszer.....	31
41. ábra: A számjegyek ábrázolása a feladatban.	31
42. ábra: Tanítás (35->10).....	32
43. ábra: Szimuláció (35->10).	33
44. ábra: Tanítás (35->56->10).....	33
45. ábra: Szimuláció (35->56->10).....	34
46. ábra: Tanítás (35->128->64->32->10).....	34
47. ábra: Szimuláció (35->128->64->32->10).....	35

1. Bevezető

A természetes intelligencia megértésére és intelligens gépek építésére régóta megszületett a törekvés az emberben, már a görög mitológiában van szó mechanikus emberről. Ennek a törekvésnek a révén jött létre a mesterséges intelligencia. A legtöbb típusú mesterséges neuronháló szerkezete nagyban hasonlít a gerincesek idegrendszeréhez és azok több tulajdonságát is modellezzik. Például a neuronok szinapszisokon keresztül kommunikálnak egymással [1].

Mesterséges neurális hálózatok általános jellemzői:

Nagyszámú egymáshoz kötődő egyszerű feldolgozó egységek (neuronok) összessége (modularitás).

A neuronok párhuzamosan dolgozhatnak (párhuzamosság).

A neuronok egymással történő összekapcsolása által kialakíthatunk különböző topológiájú neurális hálókat. (újrakonfigurálás)

Napjainkban mesterséges neurális hálózatok nagy érdeklődésnek örvendenek a kutatások és ipar szférájában, főleg a gépi látás és mesterséges intelligencia területén mutatott pontosságuknak köszönhetően. Azonban ezek a rendszerek hatalmas számítási kapacitást igényelnek [1].

A GPU-k (grafikus feldolgozó egység) rendelkeznek a szükséges számítási kapacitással, viszont nem elég energia takarékosak a beágyazott rendszerekbe történő alkalmazáshoz. A CPU-k (központi feldolgozó egység) párhuzamosítási lehetőségei korlátozottak és szintén nem elég energia takarékosak. Az ASIC (alkalmazás specifikus integrált áramkör) alkalmas lehet energia takarékoság és teljesítmény szempontjából is, de nagy hátránya, hogy nem újrakonfigurálható [2].

Egyik alternatíva neuronhálók beágyazott rendszerekben történő megvalósításához az FPGA (programozható kapumátrix) áramkör. Logikai blokkokból épül fel, amelyeket egyszerű funkcionalitásra programozhatunk. A különböző logikai blokkokat össze lehet kapcsolni egymással az FPGA-ban található kapcsoló mátrixok segítségével, ezáltal kialakítva bonyolultabb szerkezeteket. Egy FPGA lapon több egymástól független struktúra hozható létre, amelyek párhuzamosan dolgoznak. Így alkalmas a neurális hálózatokban jelen levő modularitás és párhuzamos végrehajtás megvalósítására. Az újraprogramozhatóság következtében különböző modelleket valósíthatunk meg ugyanazon az FPGA áramkörön.

Az FPGA áramkörök egy nagyságrenddel kevesebb energiát fogyasztanak a grafikus feldolgozó egységeknél és jelentősen gyorsabbak az általános célú processzoroknál.

Rugalmasak és alkalmasak fejlesztésre újrakonfigurálhatóságuknak köszönhetően. A rendszer karbantartható, valamint bármely ötlet azonnal megvalósítható, az alkalmazás specifikus áramkörökkel szemben, ahol meg kell várunk a hosszú tervezési majd gyártási folyamatot.

Az FPGA áramkörök tartják a lépést a technikai fejlődéssel. A legújabb FPGA lapok már 7 nm-es tranzisztor mérettel készülnek, aminek köszönhetően még kisebbek és nagyobb teljesítményűek. Méretüknek és alacsony fogyasztásuknak köszönhetően jól alkalmazhatóak beágyazott rendszerekben.

Hátrányaik közé tartozik, hogy korlátozottak az erőforrásaik akárcsak a GPU-knak, és lassú a fordítás. További hátrányuk, hogy az FPGA-n történő fejlesztéshez szükséges a VHDL vagy Verilog programozási nyelvek ismerete, valamint digitális rendszer alapismeretek. Az FPGA áramköröket programozó mérnökök kell tudják használni a fejlesztési és szimulációs eszközöket. Tehát az FPGA-n történő fejlesztés sok tapasztalatot szakértelmet és időt igényel. A neuronhálók pedig gyorsan fejlődnek, különböző feladatokhoz különböző felépítésű neuronhálóra van szükség. Ezek állandó újraprogramozása FPGA-ra nagyon hatástalan lenne. Erre a problémára nyújtana megoldást az általunk kiépített keretrendszer.

2. Irodalmi áttekintés

2.1. Magas szintű szintézis (HLS)

A HLS eszköz bemenete az alkalmazás funkcionális leírása kiegészítve a kényszerfeltételekkel (direktívákkal) valamint, könyvtárelemek az elérhető komponensekkel (tetszőleges pontosságú adattípusok, DSP modulok, memória modulok). Kimenete a regiszter szintű leírás (RTL). A tervező leírja a modulok funkcionalitását és megtervezi a modulok közötti kommunikációt. A HLS eszköz kompilálja a funkcionális leírást, kiosztja a hardver erőforrásokat, ütemezi a műveleteket órajel szerint, a műveleteket funkcionális egységekhez köti, a változókat tároló elemekhez kapcsolja és generálja az RTL architektúrát [3].

2.1.1. Kompilálás (Compilation)

Első lépésként a HLS a bementként kapott kód kompilálását végzi el. Ez a folyamat magába foglalja a kód optimalizálását. Eltávolítja az úgynevezett „dead-code”-ot, vagyis azokat a részeket, amelyeknek az eredménye nem használandó fel. Eltávolítja a hamis adatfüggőségeket, kiértékeli a konstans kifejezéseket (constant folding) és ciklus átalakításokat (loop transformation) végez, hogy felfedje a párhuzamosítható részeket. A kompilátor által generált modellből kimutathatóak az adat- és kontrollfüggőségek. Az adatfüggőségek ábrázolhatóak

egy adatfolyam gráfként (DFG), amiben minden csúcs egy műveletnek felel meg és a csomópontok közötti utak jelentik a bemeneteket, kimeneteket és ideiglenes változókat. Ez a gráf létrehozható, ha töröljük a kontrollfüggőségeket a leírásból a kompilálás során. Ennek az eléréséhez teljesen kikell terjeszteni a ciklusokat, ami nagy modellhez vezethet és csak statikus ciklus számláló esetén valósítható meg. Az adatfolyam gráfot kibővítve a kontroll függőségekkel kapunk egy irányított gráfot, ahol az élek jelentik a vezérlő folyamatot. A csúcsok alap blokkokat jelentenek, amik nem tartalmaznak elágazásokat, csak szekvenciális állításokból épülnek fel. Ezzel a gráffal ábrázolhatóak ismeretlen ciklusszámlálóval rendelkező ciklusok, de a párhuzamosíthatóság csak az alap blokkokon belül egyértelműsíthető. Az alap blokkok közötti párhuzamosíthatóság felfedéséhez további elemzések lennének szükségesek. Ezek lehetnek automatikusok, vagy manuálisan megadhatóak a felhasználó által [3].

2.1.2. Erőforrás kiosztás (Allocation)

Második lépésként a HLS eszköz meghatározza a szükséges erőforrásokat. HLS eszköztől függően néhány komponens kiválasztható az ütemezés, vagy a műveletek funkcionális egységhez való kötésénél. Az erőforrásokat az RTL komponens könyvtárból választja ki. A könyvtárnak tartalmaznia kell a komponensek tulajdonságait (késleltetés, teljesítmény stb.). Minden művelethez kiválaszt legalább egy komponenst [3].

2.1.3. Ütemezés

Harmadik lépés az ütemezés. A műveletek végrehajthatóak 1 vagy több órajel alatt, a hozzájuk rendelt funkcionális egységtől függően. A műveletek egymás után láncolhatóak, vagyis egyik művelet eredménye a másik bemenetét képezi és párhuzamosíthatóak, ha nincs közöttük adattfüggőség és elegendő erőforrás áll rendelkezésre [3].

2.1.4. Összekapcsolás (Binding)

Negyedik lépés a változók és műveletek funkcionális egységekhez való kötése. Minden változót egy tároló egységhez kell kapcsolni. Azok a változók, amelyek nem fedik át egymás élettartamát köthetők ugyanahhoz az erőforráshoz. Minden műveletet egy olyan funkcionális egységhez kell kötni, amelyik képes elvégezni azt. Ha több ilyen funkcionális egység is rendelkezésre áll, akkor a kötést végző algoritmus kell kiválassza az optimálist. A tárolókat össze kell kapcsolni a funkcionális egységekkel, például síneken vagy multiplexereken keresztül [3].

2.1.5. RTL generálása (Generation)

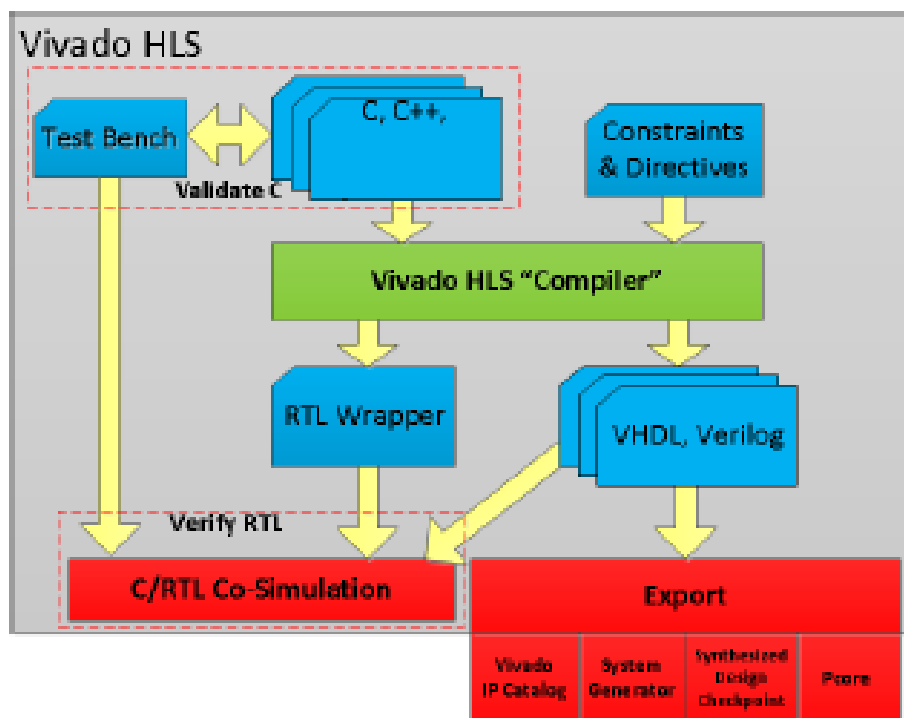
Miután a HLS eszköz elvégezte a kompilálást, kiosztotta az erőforrásokat, végrehajtotta az ütemezést és az összekapcsolást a következő és egyben utolsó lépés az RTL modell generálása az előzőleg meghozott tervezési döntések alapján.

Az erőforrás kiosztás ütemezés és összekapcsolás végezhető egyidőben is, de mivel mind összefüggenek egymással, a folyamat nagyon bonyolulttá válna, ezért általában sorban végzik el őket a HLS eszközök [3].

Az RTL regiszter szintű alkotó elemekből épül fel. Általában tartalmaz egy vezérlőt és egy adatútat. Az adatút tároló egységekből (regiszterek, memóriák stb.), funkcionális egységekből (aritmetikai logikai egységek, szorzó és eltoló áramkörök stb.) és a kapcsolatot biztosító elemekből (multiplexerek, sínek) áll. Az adat be- és kimenetek az adatúthoz vannak kapcsolva míg a kontroll jelek a vezérlőhöz. A kontroller egy véges állapotú gép (FSM), amely összehangolja az adat áramlását az adatúton a kontroll jelek segítségével [3].

2.1.6. Vivado HLS

A Vivado HLS a Xilinx HLS eszköze, ami C, C++ vagy SystemC leíró nyelvekből generál VHDL, Verilog és SystemC RTL leírásokat. A generált leírást IP magba lehet csomagolni. Az IP magok beilleszthetők a Vivado tervezői felületébe [4].



1. ábra: HLS folyamatok [5]

A funkcionális leírásban található függvényeket a HLS külön RTL modulként értelmezi, melyeknek a paraméter listán keresztül lehet meghatározni a ki és bemeneteiket. A függvény paraméterei átadhatóak érték szerint (adat port), cím szerint mutatóval (olvasható és írható be- és kimeneti porton), cím szerint tömbbel (alapértelmezetten memóriaként van kezelve, írható és olvasható). A függvénynek lehet visszatérítési értéke, ami az `ap_return` nevű kimeneti portot generálja. Minden RTL modul rendelkezik egy bemeneti órajel és reset porttal, opcionálisan beállítható egy órajel engedélyező port is. A HLS eszköz vezérlőjeleket rendel az RTL modulokhoz. Az `ap_ctrl_hs` kényszerfeltétel beállításával a modulhoz generálódik az `ap_start` jel, amivel lehet engedélyezni a modul működését, az `ap_idle` jelzi a modul állapotát, az `ap_done` jelez, ha a modul befejezte a feladatát, az `ap_ready` jelez, ha a modul készen áll a következő adat fogadására. Az `ap_ctrl_chain` opciót beállítva az előbb felsorolt jelek mellé generál egy `ap_continue` jelet is, amellyel lehet jelezni a modul számára, hogy folytassa a művelet végzést. Az `s_axilite` AXI sínrendszerre illeszthető interfészt generál. Az `ap_ctrl_none` protokollt kiválasztva nem generálódnak modul szintű vezérlőjelek. A modul szintű protokollok mellett léteznek függvény argumentumokhoz rendelt protokollok is. Ilyenek az `ap_none`, `ap_stable`, `ap_ack`, `ap_vld`, `ap_ovld` és `ap_hs`. Ha ezek valamelyike be van állítva külön port jön létre neki.

Fontosabb kényszerfeltételek [6]:

- **Allocation:** Korlátozza az erőforrás kiosztást, több modul fogja ugyanazt az erőforrást felhasználni
- **Array_map:** Több kisebb tömböt egy tömbbe egyesít, csökkentve a felhasznált BRAM-ok számát.
- **Array_partition:** Tömbök helyett regisztereket használ, ezáltal csökkentve az adatokhoz való hozzáférési időt.
- **Array_reshape:** Csökkenti a tömbök elemszámát azáltal, hogy megnöveli a szóméretet, javítva a hozzáférést anélkül, hogy több BRAM-ra lenne szükség.
- **Data_pack:** Egy struktúra adatmezőit egy nagyobb szószéllességű skálárba helyezi
- **Dataflow:** Feladatszintű csővezetékesítés, növeli az RTL modul egyidejűségét és az áteresztőlépességet
- **Dependence:** Információt szolgáltat, ami segíthet kiküszöbölni az iterációk közötti függőségeket. A HLS automatikusan azonosítja a ciklusokban a függőségeket, azonban bizonyos esetekben hamis függőségeket detektálhat, amik megakadályozzák a ciklusok

csővezetékesítését. A dependence direktíva által jelezhetjük a HLS-nek, hogy valóban létezik-e függőség vagy nem.

- **Expression_balance:** A C alapú programokban megjelenhetnek olyan kifejezések melyek több művelet láncolatából épülnek fel az RTL modulban. Kis órajel esetében ez növelheti a rendszer késleltetését. Alapértelmezetten a HLS átrendezi a műveleteket az kommutativitás és asszociativitás tulajdonságoknak megfelelően, hogy egy kiegyensúlyozott fát hozzon létre, csökkentve a késleltetést, többlet hardver felhasználás által. Az expression_balance direktívával ezt a funkciót kilehet kapcsolni.
- **Function_instantiate:** Minden függvény példányosítás esetén egy egyedi RTL megvalósítás keletkezik. Ez lehetővé teszi, hogy a minden példány helyileg legyen optimalizálva a függvény argumentumainak megfelelően.
- **Inline:** Eltünteti az adott függvényt a függvényhívás helyére való behelyettesítés által. A függvény ezután nem lesz egy külön álló egység RTL szinten. Néhány esetben ez lehetővé teszi a hatékonyabb optimalizálást a függvény körüli műveletekkel
- **Interface:** Meghatározza hogyan lesznek az RTL modul portjai generálva az interfész szintetizálása közben
- **Latency:** Megadható egy maximum vagy egy minimum késleltetési érték, vagy mindkettő.
- **Loop_flatten:** Lehetővé teszi egymásba ágyazott ciklusok egy ciklusként való kezelését, csökkentve a szükséges órajelciklusok számát és néhány esetben jobb optimalizációs lehetőségeket kínálva
- **Loop_merge:** Egymás után következő ciklusokat egy ciklusba von össze csökkentve a késleltetést, növelve az erőforrás használatot, javítva a logikai optimalizációt.
- **Loop_tripcount:** Manuálisan beállítható a ciklus által elvégzendő ismétlések száma. Nincs hatással a szintézisre, csak analízisre használható
- **Pipeline:** Csővezeték hoz létre, engedélyezve a műveletek párhuzamos végrehajtását, csökkentve az inicializálási intervallumot.
- **Protocol:** Meghatároz egy kód részt, melyben nem történnek órajel műveletek. Ez a rész használható egy interfész megadására, ami biztosítja, hogy a végső rendszer összekapcsolható lesz más hardverekkel ugyanazzal az interfész protokollal.
- **Reset:** Reset jelet ad hozzá globális vagy statikus változókhoz.
- **Resource:** Meghatározza, hogy egy tömb, aritmetikai művelet, vagy függvényargumentum milyen könyvtári erőforrással legyen megvalósítva.

- Stream: A tömb megvalósítására FIFO-t használ BRAM helyett. Ez hatékonyabb megoldás adatfolyamok esetében.
- Unroll: Kiterjeszti a ciklusokat, lehetővé téve azok iterációinak párhuzamos végrehajtását. Az RTL leírásban a ciklus tartalma többszöröződik.

A rendszerben felhasznált kényszerfeltételek a későbbiekben részletesen, példákon keresztül lesznek bemutatva.

2.1.7. HLS előnyei RTL szintű tervezéshez képest

- Csökkenti a tervezési időt. A tervező a rendszer viselkedésének leírására kell koncentráljon, a mikroarchitektúra részletes megvalósítása helyett. Megvalósítás során kisebb a valószínűsége hibák generálásának a fejlesztő által.
- Az ellenőrzési folyamat gyors. A rendszer viselkedését szoftveres eszközökkel tesztelhetjük egyszerűbben és gyorsabban.
- A tervezői lehetőségek feltárása (DSE) gyorsabb. A mikroarchitektúra változtatható a HLS eszköznek megadott kényszerfeltételek által, a forráskód változtatása nélkül. RTL szintű fejlesztésnél ez csak a forráskód jelentős részének megváltoztatásával lehetséges.
- A célplatform változtatható. A HLS képes a kiválasztott platformnak megfelelően újraütemezni a műveleteket.
- A HLS eszköz szoftver fejlesztők által is használható. A VHDL vagy Verilog nyelvekben történő fejlesztés alapos hardver ismereteket igényel. A HLS figyel a hardveres tervezés részleteire, megkönnyítve a tervező munkáját.
- A tervezési és ellenőrzési idő lecsökkentése által csökkenek a fejlesztési költségek és a piacra kerülési idő [7].

2.1.8. HLS hátrányai RTL szintű tervezéshez képest

- Jelenleg még a legújabb HLS eszközök sem biztosítanak annyira jó teljesítményt és erőforrás felhasználást, mint a kézzel írt RTL modellek, de sok esetben megközelíthetik azt [7].

2.2. Hasonló 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 létrejött több nyílt forráskódú és kereskedelmi neurális háló könyvtár és keretrendszer.

2.2.1. PyTorch

Az egyik legrégebbi a Torch¹, 2002-ben jelent meg. Lua² szkript nyelven keresztül biztosít interfészt C-ben és CUDA-ban implementált matematikai modellekre. PyTorch ugyan arra Lua szkript alatt rejlő könyvtár alapra épít, viszont kiegészíti a rendszert egy Python alapú keretrendszerrel.

A rendszer kialakítása során a következő főbb szempontokat [8] vették figyelembe:

- A szintaktika az hasonlítson minél jobban a Pythonéhoz, mivel a kutatok, aki ilyen témában dolgoznak, javarészt Pythont használnak.
- A kutató legyen a fő cél felhasználó. A modellek és a tanító algoritmusok azok egyszerű API függvények mögé vannak rejtve, a felhasználó csak ezeket a metódusokat kell használnia a kísérletezés során.
- Viszonylag hatékony legyen. Az egyszerűség az ne legyen a teljesítmény kárára. Valamint a felhasználó, ha szeretné tudja kikerülni az API hívásaikat és manuálisan tudja alítani a modelleket, ha szükség van rá.
- A fejlesztési időt inkább különböző funkciók fejlesztésére fordítsák mintsem aprólékos optimalizálásra.

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 nekik. Ugyanakkora a paramétereket osztott memórián keresztül osszák meg a párhuzamosan futó folyamatok között, így nincs szükség annyi folyamatközötti kommunikációra. [8]

2.2.2. Theano

Az egyik népszerű Python könyvtár az a Theano³, 2008-óta fejlesztik, a NumPy⁴ könyvtáron alapszik. Erőssége, hogy nagyon jól kezeli a több dimenziós tárolókat, és hatékonyan tud velük műveleteket végezni. Több keretrendszer épül erre a könyvtárra, főként azért, mert a könyvtár modelljei C-ben, illetve CUDA-ban vannak megvalósítva, ami gyors futást biztosít CPU-n és

¹ Torch: <http://torch.ch/>

² Lua: <https://www.lua.org/>

³ Theano: <https://pypi.org/project/Theano/>

⁴ NumPy: <https://numpy.org/>

GPU-n egyaránt [9]. Ha teheti, akkor a matematikai modellek kiszámítására ezért sok esetben nem használja ki eléggé a CPU erőforrásokat [12].

2.2.3. Caffe

A Caffe⁵ [10] 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 a bemenetek osztályozására. Python interfészen keresztül biztosítanak hozzáférést lassabb modellekhez is, amikkel lehet új prototípusokkal és új tanítóeljárásokkal kísérletezni.

2.2.4. TensorFlow

TensorFlow⁶ 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 (tenzor⁷) 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 operátorok (mátrix szorzás, konvolúció stb.) [11]. Ez az architektúra nagyon flexibilis és könnyedén skálázható. [12]

2.2.5. Microsoft Cognitive Toolkit

CNTK⁸ 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 [12]. Microsoft 2019-óta nem támogatja többet ezt a projektet.

⁵ Caffe: <https://caffe.berkeleyvision.org/>

⁶ TensorFlow: <https://www.tensorflow.org/>

⁷ Tenzor: Egy vagy több dimenziós adat tároló, mint például egy vektor vagy egy mátrix.

⁸ Microsoft Cognitive Toolkit: <https://github.com/microsoft/CNTK>

2.2.6. Keretrendszerek összehasonlítása és alkalmazásai

Property	CNTK	TensorFlow	Theano	Keras
Core	C++	C++	Python	Python
CPU	✓	✓	✓	✓
Multi-Threaded CPU	✓	Eigen	Blas, conv2D, Limited OpenMP	✓
GPU	✓	✓	✓	✓
Multi-GPU	✓	✓	X (experimental version)	✓
Nvidia cuDNN	✓	✓	✓	✓

2. ábra: A keretrendszerek funkcionalitásának az összehasonlítása. [12]

A mély tanulású hálózatok forradalmi változásokat hoztak olyan területeken, mint a számítógépes látás, beszédfelismerés és sok más területen. Ezért alkalmazni kezdték őket adatközpontokban, beágyazott rendszerekben, mint például robotoknál és telefonokban. Mivel a mély tanulású hálózatok számítás és memória igénye nagy, elkezdtek hardveres gyorsítókát kifejleszteni. Ezek közé tartoznak az FPGA-s gyorsítók is, melyek nagy népszerűséget szereztek rugalmasságuk, teljesítményük és kis energiafogyasztásuk miatt. Azonban az FPGA alapú gyorsítók programozása, sok tapasztalatot és szakértelmet igényel és nagyon időigényes folyamat. A neuronhálók gyorsan fejlődnek, ezek állandó újraprogramozása FPGA-ra kevésbé hatékony. [13]

Az FP-DNN keretrendszer “end-to-end” módon dolgozik. A szoftver alapú leírásból generálja az FPGA alapú megvalósítást, mindezt teljesen automatikusan. A TensorFlow által generált modellből a saját szimbolikus fordítója generálja a C++ programot, amiből a HLS eszköz generálja a hardvert [15].

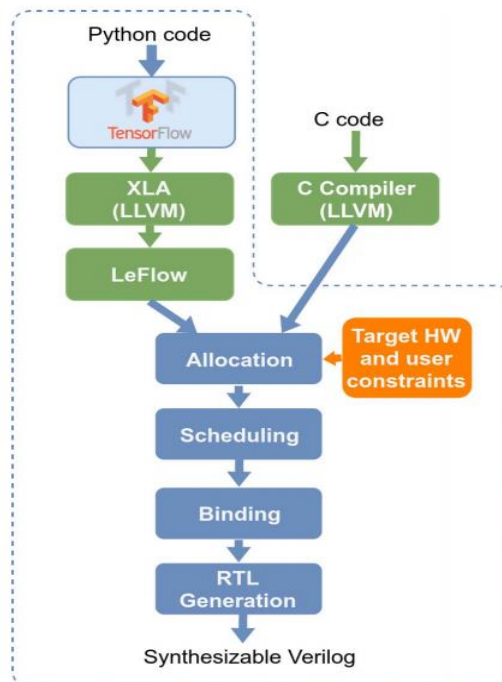
A Vitis AI a Xilinx fejlesztői platformja, neurális hálózatok Xilinx hardware platformok számára. Tartalmaz optimalizált IP-ket, eszközöket, könyvtárakat, modelleket és teljes példákat. Használja a népszerű keretrendszereket (Caffe, TensorFlow PyTorch). Rétegenkénti elemzést végez, hogy segítsen a szűk keresztmetszetek megtalálásában. A hatékony és méretezhető IP magok testre szabhatóak.

LeFlow⁹ 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 LegUp¹⁰ le tud fordítani regiszter szintű logikára (RTL). LeFlow ezt a

⁹ LeFlow: <https://github.com/danielholanda/LeFlow>

¹⁰ LegUp: <https://www.legupcomputing.com/>

fordítási folyamatot próbálja teljesen automatizálni, és optimalizálni HLS direktívák segítségével [14].



3. ábra: LeFlow folyamat ábrája [14].

3. Célkitűzések

3.1. A kutatás rövid bemutatása

Kutatásunk célja egy keretrendszer megvalósítása, mely lehetővé teszi különböző típusú és méretű neurális hálózatok FPGA áramkörön történő modellezését, paraméterezés általi megépítését, esetlegesen grafikus felületen való megtervezését.

A kutatás céljai közé tartozik a neurális hálózatok optimalizálási lehetőségeinek tanulmányozása gyorsítás és erőforrás felhasználásának szempontjából. A keretrendszer lehetőséget biztosítana a felhasználónak, hogy adott korlátok között meghatározza a gyorsítás és a felhasznált erőforrás arányát. A neurális hálókat szeretnénk egyaránt tesztelni és tanítani is az FPGA-n.

A kutatást az Accenture Student Research Scholarship pályázat keretén belül kezdtük el, Dr. Brassai Sándor Tihamér vezetőtanár támogatásával. Ez a kutatás képezi diplomadolgozatunk témáját is.

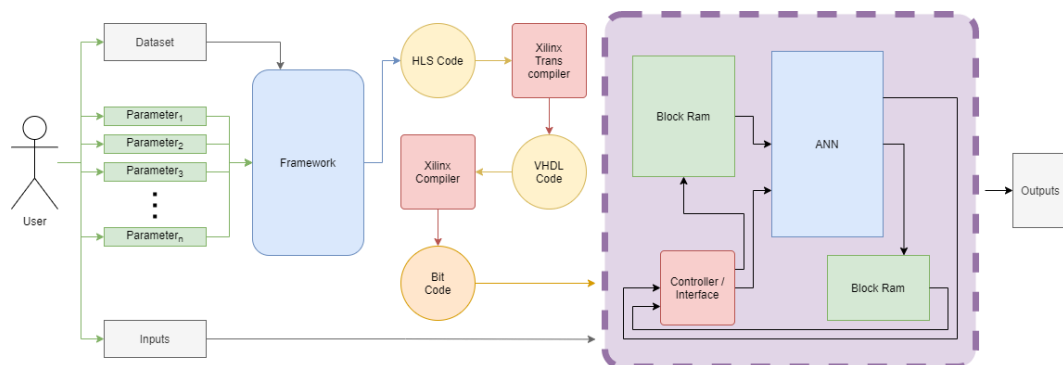
A kutatás során két technológiával kísérletezünk. Használunk VHDL (Very High Speed Integrated Circuit Hardware Description Language) hardware leíró programozási nyelvet, melynek előnye, hogy nagyfokú hatékonyság érhető el, viszont a fejlesztési folyamat nehézkes

és lassú. A másik technológia, amit használunk a HLS (High-Level Synthesis) eszköz. A HLS elemez egy algoritmikus leírást (C, C++, SystemC, Matlab) és kigenerálja belőle a hardware-t. A hardware implementációt direktívák segítségével tudjuk befolyásolni. Előnye a gyorsított fejlesztési és tesztelési folyamat (akár 10-szer gyorsabb fejlesztés, mint VHDL-ben). Hátránya, hogy nem tudunk órajel szinten ütemezni, ezért általában kisebb hatékonyság érhető el, mint VHDL használatával.

A kutatás várható eredményei neuronhálók HLS és VHDL implementációinak az összehasonlítása, az általunk épített keretrendszer összehasonlítása már létező keretrendszerekkel és neuronhálók hardware megvalósításaival, a már létező keretrendszerek hiányosságainak feltárása és ezek beépítése a saját keretrendszerünkbe, vagy a saját keretrendszer beépítése más keretrendszerbe úgy, hogy pótolja annak a hiányosságait.

3.2. A javasolt rendszer elvi architektúrája

A keretrendszer, a felhasználó által megadott paraméterek alapján generálja a neurális háló VHDL vagy 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. A bemeneti adatokat egy interfészen keresztül adagoljuk a felfprogramozott FPGA-nak és a kiszámolt eredményeket ugyancsak az interfészen keresztül fogja az a rendszer visszaküldeni.



4. ábra: A javasolt rendszer tömbvázlata.

Az interfész megvalósítására egy potenciális megoldás lenne a Xilinx által készített ChipScopy használata, amin keresztül csatlakozhatunk az FPGA virtuális ki-bemeneteire, amin keresztül tudjuk adagolni a neuronháló bemeneteit.

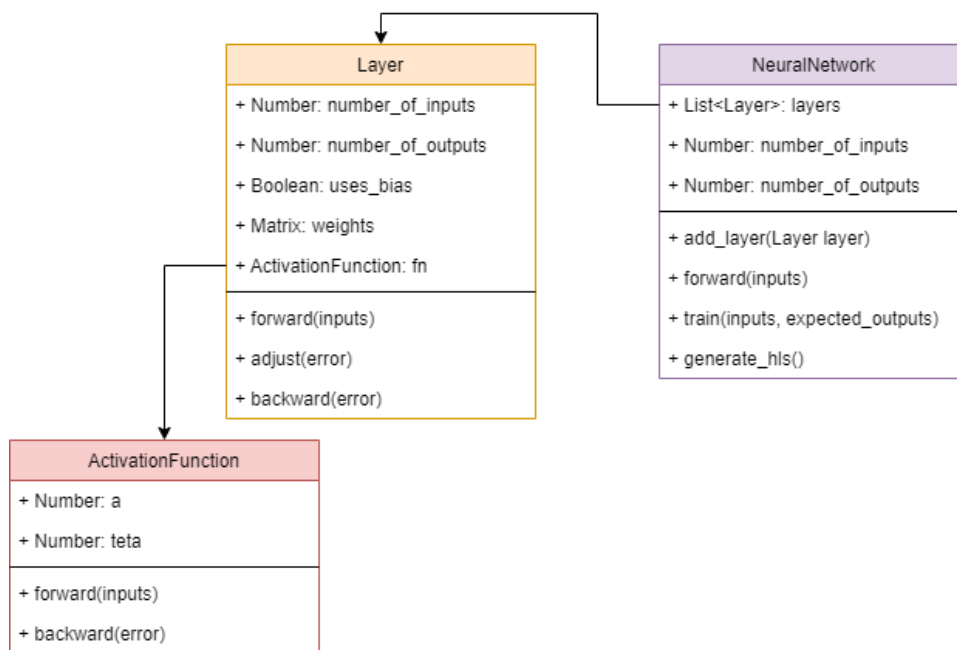
4. Módszerek

A neuronhálók megvalósítását a legkisebb funkcionális egységek implementálásával kezdjük, majd ezeket összeillesztve kapjuk meg a bonyolultabb szerkezeteket. A kiindulási pont egy neuron megvalósítása egy egyszerű lépcső aktivációs függvénnyel. Következő lépés több neuron generálása által egy neuron réteg kialakítása. A rétegek többszörösítésével és azok összekapcsolásával egy többrétegű neuronháló megvalósítása. Ezután az alapegységek különböző topológiákba történő rendezése által, vagy más architektúrák megvalósításával különböző típusú neuronhálókat alakítunk ki. A rendszer a későbbiekben bővíthető különböző aktivációs függvényekkel. Az egységek tervezésénél lehetőséget biztosítunk a paramétereizhetőségre és más egységekkel történő összekapcsolhatóságra. A mikroarchitektúra kényszerfeltételekkel történő optimalizálását funkcionális egységenként végezzük. A legjobb mikroarchitektúrák feltárására Design Space Exploration (DSE) alkalmazunk, vagyis kipróbálunk többféle direktíva konfigurációt és többféle megvalósítást egy adott modulra. Az eredményeket 3 fő szempont alapján értékeljük ki: teljesítmény, erőforrás felhasználás mértéke és pontosság.

5. Megvalósítás

5.1. Python modell

A tervezett keretrendszert azt Python programozási nyelv segítségével készült el. A rendszer tervezése során fontos volt az, hogy a kész rendszer átlátható és könnyedén tovább fejleszthető legyen, ezért az Objektum Orientált Programozási (OOP) elveket követi.



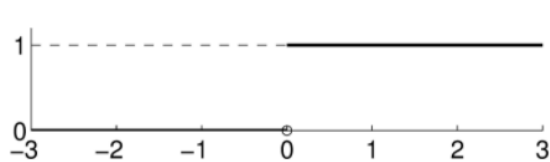
5. ábra: A rendszer osztály diagramja.

A rendszer 3 fő részből épül fel: neuronháló, réteg és aktivációs függvény.

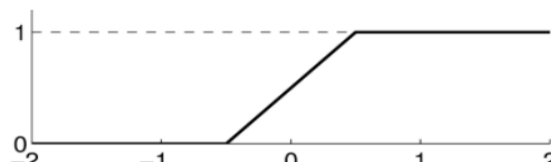
5.1.1. Aktivációs függvény

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

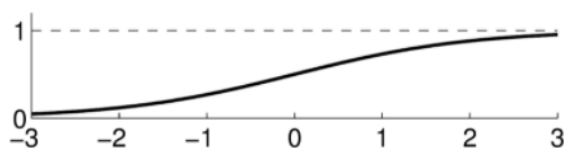
A rendszer használata során akármilyen aktivációs függvényt alkalmazhatunk, a fontosabbakat implementálva vannak:



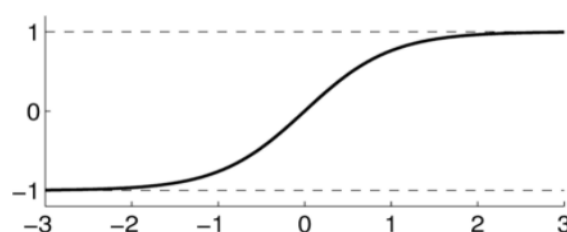
6. ábra: Küszöb függvény.



7. ábra: ReLU függvény



8. ábra: Logisztikus függvény.



9. ábra: Tangens hiperbolikus függvény.

Abban az esetben, ha a keresett függvény nem létezik, akkor a felhasználó létrehozhat egy új függvény modellt. Annak kell teljesítenie az „ActivationFunction” interfészt, azaz kell legyen

egy „forward” és egy „backward” metódusa. A „forward” metódus egy adott ingerre kell egy kimenetet megfeleltessen, míg a „backward” metódus a hibának a függvényen keresztüli visszafejtését kell megvalósítsa.

5.1.2. Perceptron réteg

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.

$$\begin{bmatrix} b \\ x_0 \\ x_1 \\ \vdots \\ x_m \end{bmatrix} * \begin{bmatrix} w_{b,0} & w_{b,1} & \cdots & w_{b,n} \\ w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,0} & w_{m,2} & \cdots & w_{m,n} \end{bmatrix} = \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ \vdots \\ s_n \end{bmatrix}$$

10. ábra: Egy réteg ingereinek a mátrix szorzással való felírása.

Amint a fenti ábrán (10. ábra) látható egy réteg ingereinek kiszámítása felírható egy mátrix szorzásként. Egy inger kiszámításához szükséges képlet alább látható (11. ábra).

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

11. ábra: Egy inger kiszámítási képlete.

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

5.1.3. Neurális háló

A neurális háló osztály az magába foglalja a rétegeket és ő felelős a neuronháló adott bementérea 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 hálónál

5.1.4. Tanítás

A háló paramétereinek tanítására egy gradiens alapú módszer van alkalmazva a hibát egyszer visszatérjesztjük az aktivációs függvényen, azaz hibát megszorozzuk az aktivációs függvény deriváltjával.

$$Error = Error * f'(S)$$

12. ábra: A hiba visszafejtése az aktivációs függvényen.

Az így kapott hibát majd alkalmazzuk a súly paramétereknek a javítására, optimalizálására:

$$W_{<t+1>} = W_{<t>} + u * Error^T * X + m * M$$

13. ábra: A réteg súlymátrixának tanítása gradiens módszerrel.

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 következő módon (14. ábra) fejtjük vissza:

$$Error_{visszafejtett} = \begin{bmatrix} S_0 \\ S_1 \\ \vdots \\ S_n \end{bmatrix} * \begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,0} & w_{m,2} & \cdots & w_{m,n} \end{bmatrix}^T = \begin{bmatrix} S_0 \\ S_1 \\ \vdots \\ S_n \end{bmatrix} * \begin{bmatrix} w_{0,0} & w_{1,0} & \cdots & w_{m,0} \\ w_{0,1} & w_{1,1} & \cdots & w_{m,1} \\ \vdots & \vdots & \ddots & \vdots \\ w_{0,n} & w_{1,n} & \cdots & w_{m,n} \end{bmatrix}$$

14. ábra: A hiba visszafejtése az előző rétegre.

5.2. HLS modell

5.2.1. Az inger kiszámolása

Adott neuron ingerének a kiszámítása felfogható egy vektor szorzásként. A neuron bemeneteinek vektorát kell összeszorozni a neuronhoz tartozó súlytényezővektorral. Ez a művelet intenzív szorzást és összeadást igényel. Mivel ez a művelet igen gyakran előfordul FPGA alkalmazásoknál a gyártók speciális hardvert építettek az FPGA lapokba. DSP áramkörön megvalósítani a MAC (szorzás és összeadás) műveletet hatékonyabb, mint a programozható logikával. A modern FPGA-k több száz DSP modult tartalmaznak.

```

void vector_multiply(const MY_FIXED A[],
                    const MY_FIXED B[], MY_FIXED * __restrict__ result,
                    const unsigned size)
{
    MY_FIXED sum=(MY_FIXED)0;
multiply_loop:for(int k=0; k<size; ++k)
    {
        #pragma HLS pipeline II=1
        #pragma HLS unroll factor=4

        sum+=A[k]*B[k];
    }
    *result=sum;
}

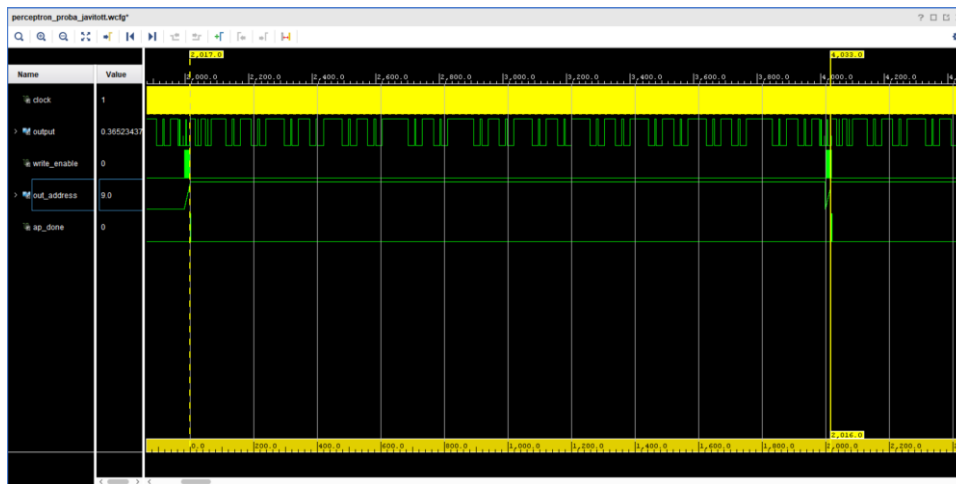
```

15. ábra: Vektor szorzás direktívákkal.

A HLS minden függvényt külön modulként kezel. A függvény paraméterei jelentik a modul ki- és bemeneteit. Jelen esetben az A és B tömbök, valamint a size változó lesznek a modul bemenetei és a result pointer lesz a kimenet. Az A és B tömbök fogják tartalmazni a neuron bemeneteit és a súlyzóit, a size változóban kapja meg a modul a neuron bemeneteinek a számát. A tömbök elemei páronként szorozódnak össze, majd az eredmény hozzáadódik a sum változóhoz.

A direktívák hatásának szemléltetésére végzett bemutató méréseket egy 2 rétegű perceptron típusú neuronhálón végeztem. A neuronháló 1. rétegje 36, a 2. 10 neuront tartalmaz. Minden neuron szigmoid aktivációs függvényt használt. Az adatok tárolása és a műveletek elvégzése 24 bites fixpontos számokkal történt, ahol 15 bit képviselte az egész részt. A neuronháló által megoldott feladatot a későbbiekben részletezem.

Az alábbi mérésnél a mérőpálcákat az ap_done jel két egymásutáni 1-es értékére állítottam. A két mérés között 2000 minta van. Ez egy 100 MHz-es órajel ciklussal 20 mikroszekundumot jelent. Ennél a mérésnél nem használtam direktívákat.

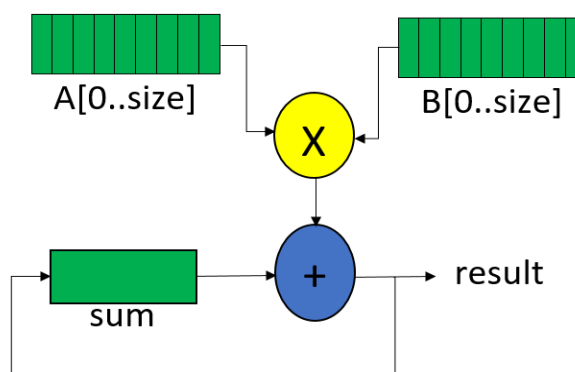


16. ábra: Mérés direktívák használata nélkül.

Ennek megfelelően a generált áramkör terület (erőforrás) takarékos. Az összes MAC művelet esetében ugyanaz a DSP van használva, így összesen csak egy DSP-re van szükség.

Utilization Estimates				
Summary				
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	63
FIFO	-	-	-	-
Instance	5	1	285	704
Memory	2	-	0	0
Multiplexer	-	-	-	376
Register	-	-	144	-
Total	7	1	429	1143
Available	120	80	35200	17600
Utilization (%)	5	1	1	6

17. ábra: Erőforrás felhasználás statisztika, direktívák használata nélkül.



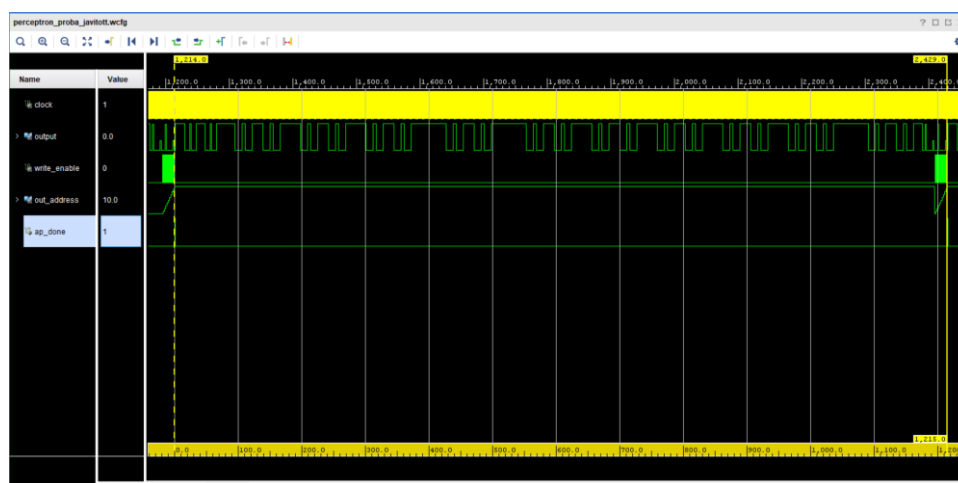
18. ábra: Vektor szorzás mikroarchitektúra direktívák alkalmazása nélkül

5.2.2. A ciklusok kiterjesztése (Unrolling loops)

Alapértelmezetten a HLS a for ciklusokat szekvenciálisan szintetizálja. Ez egy erőforrás (terület) hatékony megoldás, mert ugyanazon a hardver részen végzi el sorban a műveleteket. A `#pragma HLS unroll` direktívát használva a ciklusok kiterjeszthetők. A cikluson belül található műveletek sokszorozóttá lesznek egy faktornak nevezett érték-szer. Amennyiben ciklus iterációinak száma nem a kiterjesztési faktor egész számú többszöröse, a HLS még egy for ciklusban elvégzi a fennmaradó iterációkat. Ha ezek az iterációk nem fontosak a feladat szempontjából, kihagyhatjuk őket a `skip_exit_check` argumentum megadásával. A legjobb esetben a ciklus iterációi nem tartalmaznak adatfüggőséget (egyik állítás sem függ egy előző iterációban generált adattól), ekkor a ciklus teljesen kiterjeszthető. Ha nem adjuk meg a faktort akkor a HLS teljesen kiterjeszti a ciklust, ami jó megoldás lehet kisebb ciklusok esetében, de nagy iteráció számnál, mint például 1 millió a teljes kiterjesztés megvalósíthatatlan a hardver korlátok miatt [16].

Jelen esetben a ciklusban 4 művelet van: olvasás az A tömbből, olvasás a B tömbből szorzás és az összeadás. Az olvasás és szorzás műveletek végig függetlenek egymástól. Az összeadási művelet függ az akkumulátor előző értékétől, olvasás írás után adatfüggőség lép fel. A ciklust 4-es faktorral terjeszttem ki, hogy elkerüljem az adatfüggőséget.

Az alábbi mérésnél a ciklus 4-es faktorral volt kiterjesztve. Egyszerre 4 MAC műveletet végez el, ennek megfelelően 4 DSP-t is használ. A művelet elvégezhető 1215 órajel alatt.

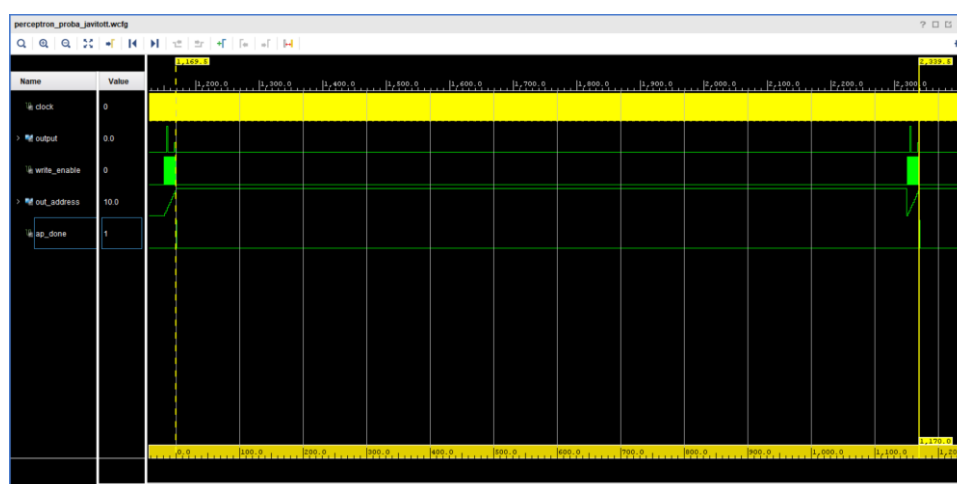


Utilization Estimates				
Summary				
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	98
FIFO	-	-	-	-
Instance	5	4	620	1844
Memory	0	-	68	20
Multiplexer	-	-	-	324
Register	-	-	45	-
Total	5	4	733	2286
Available	120	80	35200	17600
Utilization (%)	4	5	2	12

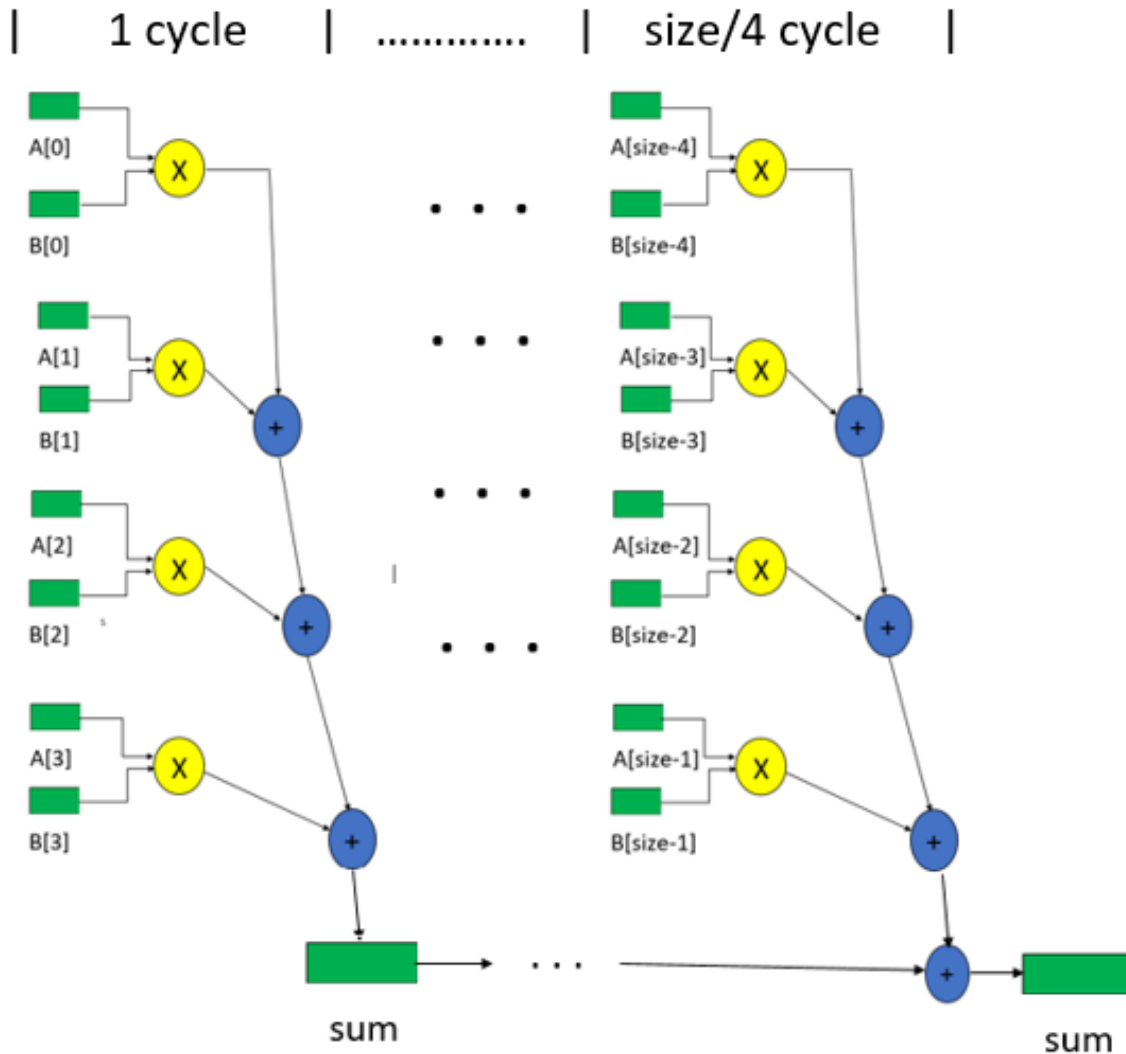
20. ábra: Erőforrás felhasználási statisztika, vektor szorzás kiterjesztve 4-es faktorrall.

A ciklusok kiterjesztése által a teljesítmény nagyban növelhető feltéve, ha figyelünk az írás és olvasás műveletekre. A tömbök értékei általában BRAM-okban vannak eltárolva. Több típusú BRAM létezik, az FPGA lapokban leggyakrabban használt BRAM-oknak két port-ja van, amelyeket lehet használni írásra vagy olvasásra, így 1-nél több olvasási és írási művelet nem végezhető el párhuzamosan. Erre a problémára megoldást jelent, ha tömb értékeit külön regiszterekbe tároljuk, mert egy regiszter írható és olvasható is egy órajel alatt. A HLS-t a `#pragma HLS array_partition variable=_tömb_neve_ complete` direktívával lehet utasítani tömb értékeinek külön regiszterekbe való eltárolására. Jelen esetben a 4-es kiterjesztésnek köszönhetően 4 olvasást kell elvégezzen a rendszer mindkét tömbből egyszerre.

Az A és B tömbök particionálása által a rendszer számítási ideje tovább csökken 1170 órajelre.



21. ábra: Mérés, tömbök particionálva.



22. ábra: Vektor szorzás mikroarchitektúra, ciklus kiterjesztve, tömbök partícionálva.

5.2.3. Ciklusok csővezetékészése (Loop pipelining)

A ciklusok kiterjesztésén kívül lehetőség van a ciklusok iterációinak konkurens, de késleltetett végrehajtására. Sok esetben a ciklusok nem kiterjeszthetőek, viszont a következő iteráció megkezdhető még az előző befejezte előtt, vagyis az iterációk konkurensen hajthatók végre egy bizonyos késleltetéssel.

A mérés eredménye 390 mintára csökkent, a pipeline alkalmazása után a ciklusban. A felhasznált erőforrások száma alig növekedett.



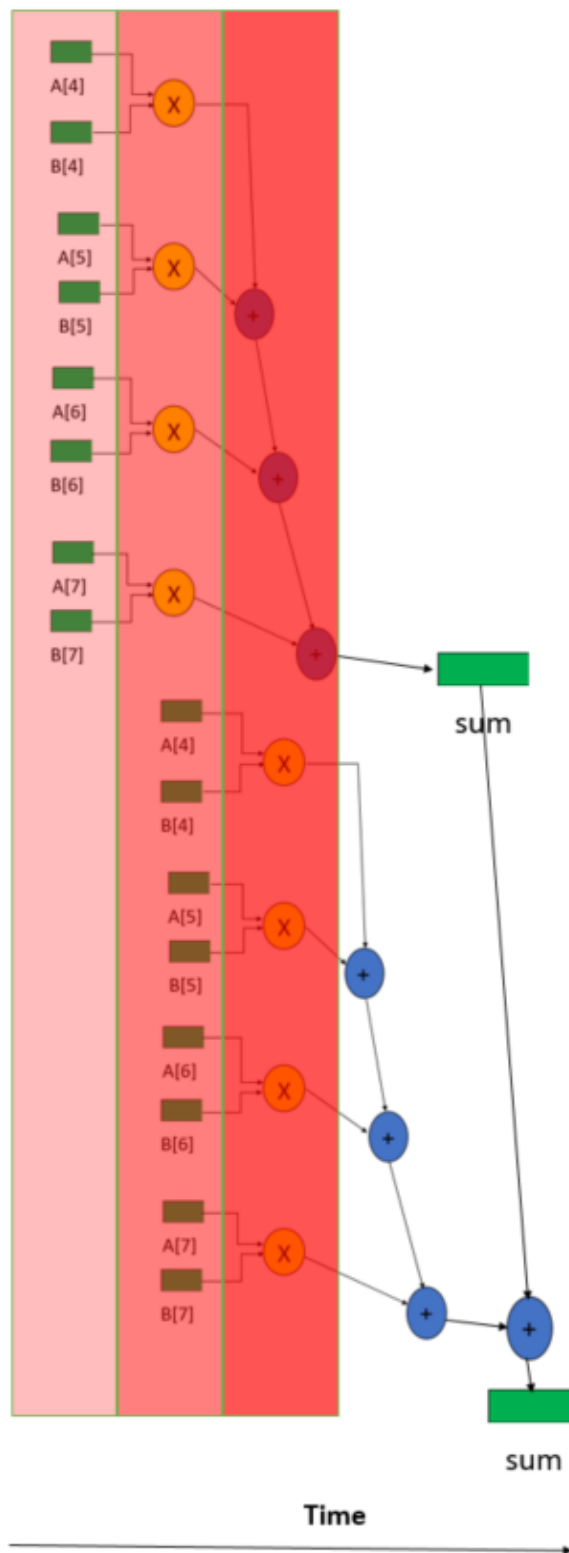
23. ábra: Mérés, vektor szorzás csővezetékvezve.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	453
FIFO	-	-	-	-
Instance	3	4	2139	4013
Memory	0	-	20	6
Multiplexer	-	-	-	206
Register	-	-	1375	-
Total	3	4	3534	4678
Available	120	80	35200	17600
Utilization (%)	2	5	10	26

24. ábra: Erőforrás felhasználási statisztika, vektor szorzás csővezetékvezve.



25. ábra: Vektor szorzás, ciklus kiterjesztve, tömbök particionálva, ciklus csővezetékkezve.

5.2.4. Egy réteg felépítése

Egy réteg az általa tartalmazott neuronokból épül fel. A réteg bemenete megegyezik neuronjainak a bemenetével. A réteg súlymátrixában soronként vannak eltárolva az egyes neuronok súlyzói.

```
void single_layer(const MY_FIXED IN[],
                  const MY_FIXED W[][greatest_layer_neurons],
                  unsigned neuron_number, MY_FIXED OUT[output_number])
{
    MY_FIXED y;

    neuron_loop:for(unsigned i=0;i<neuron_number;++i)
    {
        #pragma HLS unroll
        single_neuron(IN, W[i], &y);
        OUT[i]=y;
    }
}
```

26. ábra: Egy neuron réteg.

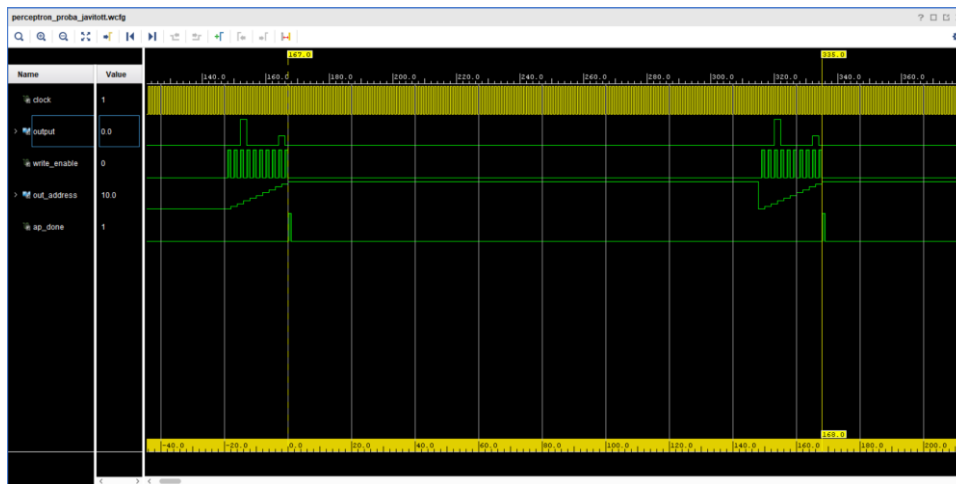
Egy neuron szerepe a saját ingerének átvezetése az aktivációs függvényén. Ennél a feladatnál szigmoid aktivációs függvény van használva.

```
void single_neuron(const MY_FIXED X[perceptron_input],
                   const MY_FIXED W[], MY_FIXED * __restrict__ y )
{
    //inger kiszámolása
    vector_multiply(X,W,y,perceptron_input);

    // aktivacios fuggveny
    //step_function(y);
    sigmoid(y);
}
```

27. ábra: Egy neuron.

Az eddigi mérések során a neuronok szekvenciálisan dolgoztak. A réteg neuronjain iteráló ciklust kiterjeszttem. A következő mérésben a neuronok már párhuzamosan dolgoznak. A mérés értéke több mint felére csökken 168 mintára, de a felhasznált erőforrások száma nagy mértékben megnő, még egy ilyen relatív kis méretű neuronháló esetében is.



28. ábra: Mérés, neuronok párhuzamosítva.

Utilization Estimates				
Summary				
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	300
FIFO	-	-	-	-
Instance	70	40	8690	30700
Memory	-	-	-	-
Multiplexer	-	-	-	59
Register	-	-	981	-
Total	70	40	9671	31059
Available	120	80	35200	17600
Utilization (%)	58	50	27	176

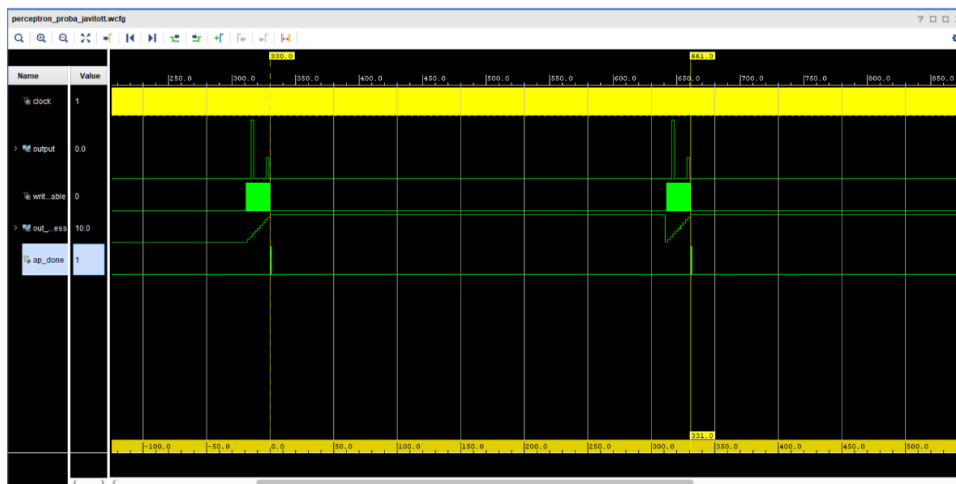
29. ábra: Erőforrás felhasználási statisztika, neuronok párhuzamosítva.

5.2.5. Fix pontos számábrázolás lebegőpontos számábrázolással szemben

A HLS-ben megvalósított neuronhálók számábrázolásánál fixpontos adattípusokat használtunk. A fixpontos számábrázolásnak számos előnye van a lebegőpontos számábrázoláshoz képest. A fixpontos rendszerek mindig jóval kevesebb erőforrást használnak. Kevesebb digitális jelfeldolgozó, kereső táblázat és bistabil áramkör szükséges fixpontos számokkal való munka során. Kisebb lesz az energia fogyasztás a kevesebb erőforrás felhasználása miatt. A számítások gyorsabbak, így kisebb a késleltetés. A számítások pontossága megközelíti a lebegőpontos számításokét. Érdemes alkalmazni a fixpontos adattípust, ha a kis pontosság veszteség nem befolyásolja az alkalmazás helyességét [17].

A különböző adattípusokkal készített méréseket, a HLS modell bemutatásánál használt neuronhálóval végzem, ugyanazzal a bemenettel, a legutolsó direktíva konfigurációt felhasználva. A lebegőpontos ábrázolásmódból indulok ki. Figyelembe veszem a generált neuronháló terület felhasználását, a kimenet kiszámításának idejét és az eredmény pontosságát.

5.2.5.1. Lebegőpontos számábrázolás



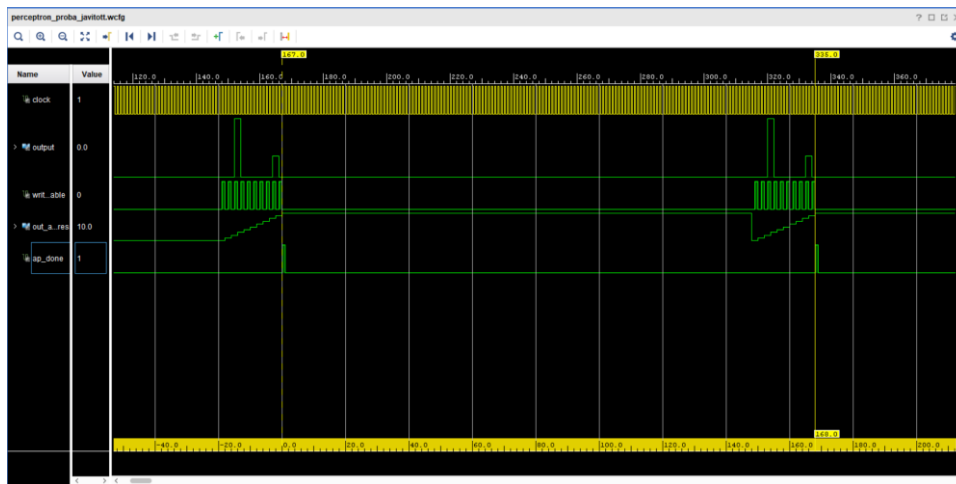
30. ábra: Mérés, 4 bájtos lebegőpontos adattípust használva.

Utilization Estimates				
Summary				
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	442
FIFO	-	-	-	-
Instance	120	50	11710	31830
Memory	-	-	-	-
Multiplexer	-	-	-	59
Register	-	-	1503	-
Total	120	50	13213	32331
Available	120	80	35200	17600
Utilization (%)	100	62	37	183

31. ábra: Erőforrás felhasználási statisztika, 4 bájtos lebegőpontos adattípust használva.

5.2.5.2. Előjeles fixpontos<32,16>

A számábrázolás során egy számot 32 biten ábrázoltunk, amiből 15 bit az egész rész és a 16 a tört részt és 1 bit az előjel.

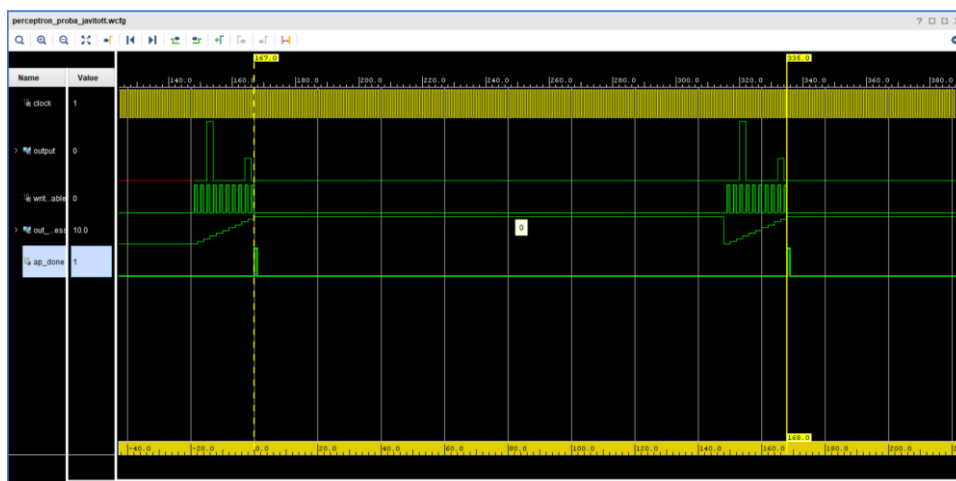


32. ábra: Mérés, fixpontos<32,16> adattípust használva.

Utilization Estimates				
Summary				
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	363
FIFO	-	-	-	-
Instance	110	120	10350	34020
Memory	-	-	-	-
Multiplexer	-	-	-	59
Register	-	-	1338	-
Total	110	120	11688	34442
Available	120	80	35200	17600
Utilization (%)	91	150	33	195

33. ábra: Erőforrás felhasználási statisztika, fixpontos<32,16> adattípust használva.

5.2.5.3. Előjeles fixpontos<16,8>



34. ábra: Mérés, fixpontos<16,8> adattípust használva..

Utilization Estimates				
Summary				
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	291
FIFO	-	-	-	-
Instance	70	40	7690	28270
Memory	-	-	-	-
Multiplexer	-	-	-	59
Register	-	-	690	-
Total	70	40	8380	28620
Available	120	80	35200	17600
Utilization (%)	58	50	23	162

35. ábra: Erőforrás felhasználási statisztika, fixpontos<16,8> adattípust használva

5.2.5.4. Előjeles fixpontos<12,6>

Utilization Estimates				
Summary				
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	273
FIFO	-	-	-	-
Instance	70	40	6980	27070
Memory	-	-	-	-
Multiplexer	-	-	-	59
Register	-	-	528	-
Total	70	40	7508	27402
Available	120	80	35200	17600
Utilization (%)	58	50	21	155

36. ábra: Erőforrás felhasználási statisztika, fixpontos<12,6> adattípust használva.

5.2.5.5. Mérések összefoglalása, kiértékelése

A mérések alapján megfigyelhető, hogy fixpontos számábrázolással nagyobb teljesítmény érhető el. A futási idő lebegőpontos számokkal 331 órajel, fixpontos számokkal csupán 168 órajel. Azonban 4 bájtos fixpontos számokkal a rendszer jóval több DSP-t használ fel, mint a 4 bájtos lebegőpontos (float) számokkal. A különböző méretű fixpontos adattípusok nem befolyásoltak a futási időt, viszont a kisebb méretűeket használva jelentősen csökken az erőforrás felhasználás. Az adattípus kiválasztásánál a felhasználónak mérlegelni kell, hogy milyen pontosságot szeretne elérni és mennyi erőforrást szeretne felhasználni. Döntési szabadságát korlátozzák a rendelkezésére álló erőforrások száma, valamint a feladat helyes megoldásához szükséges minimális adatméret. Jelen esetben a valós rész minimum 5 biten kell legyen ábrázolva, hogy az eredmény helyes legyen. Azon az értékekre, melyek a (-5,5) intervallumon kívül esnek a szigmoid értéke -1, 1 lesz. 0-5-ig a számok ábrázolására elegendő 4 bit, plusz még szükség van egy előjel bitre. Azok a számok, amelyek nem ábrázolhatóak ennyi biten szaturálni fognak. A törtész 8 biten történő ábrázolásával a kapott legnagyobb

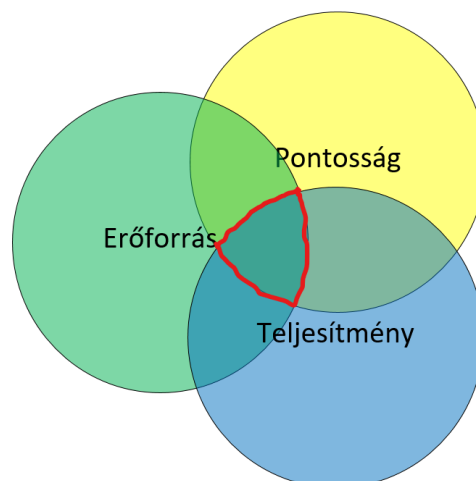
hiba a kimeneten 0.0017. Ez a pontosság bőven elégséges a jelenlegi feladathoz. Jelen esetben jó megoldás lenne egy fixpontos<14,5> adattípus használata.

Adattípus	BRAM	DSP48E	FF	LUT	Mintasám	Legnagyobb hiba
Lebegőpontos	120	50	13213	32331	331	0
fixpontos<32,16>	110	120	11688	34442	168	0.0000075
fixpontos<16,8>	70	40	8380	28620	168	0.0017688
fixpontos<12,6>	70	40	7508	27402	168	0.1783600

1. táblázat: Fixpontos és lebegő pontos számábrázolás összehasonlítása.

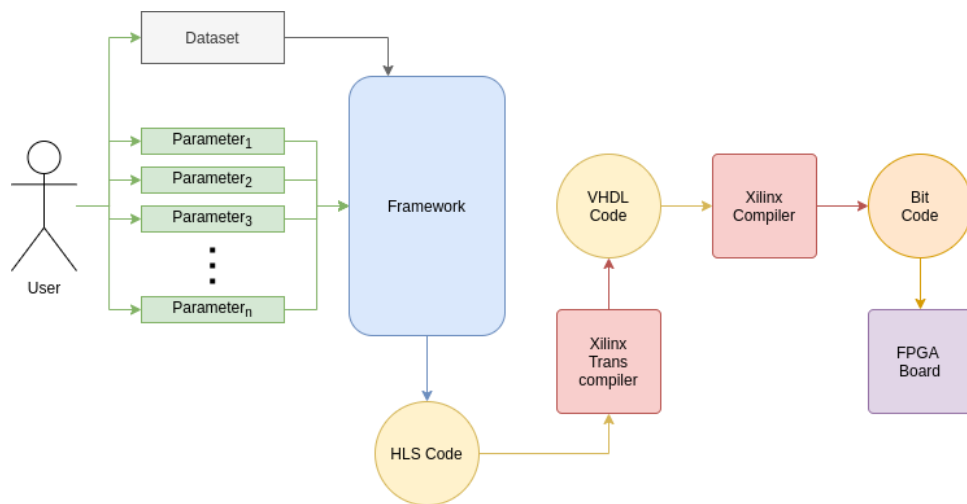
5.2.6. Optimális modell

Az optimális modell megalkotásához a tervezőnek mérlegelnie kell a rendelkezésére álló erőforrásokat, az elvárt teljesítményt, valamint a feladat megoldásához szükséges pontosságot. Ha ezek közül bármelyiken javítani szeretne azt egy másik, vagy a másik kettő rovására teheti meg. Ha kettőn szeretne javítani az biztosan a harmadik rovására megy. Az optimális modell eléréséhez a felhasználónak kompromisszumot kell kötnie a 3 paraméter között. Ahogy az alábbi ábrán is látható, optimális megoldások halmaza a három paraméter metszéspontján belül, a piros vonallal határolt területen található. A keretrendszer célja, hogy a felhasználó számára lehetőséget biztosítson a pontosság és a teljesítmény befolyásolására a rendelkezésre álló erőforrásokhoz mérten.



37. ábra: A modell korlátai.

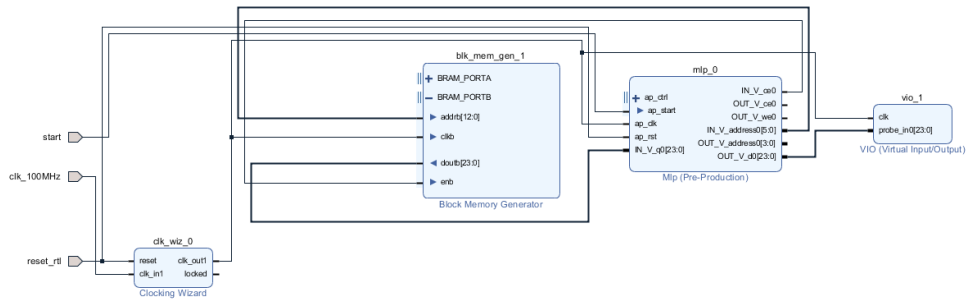
6. A rendszer



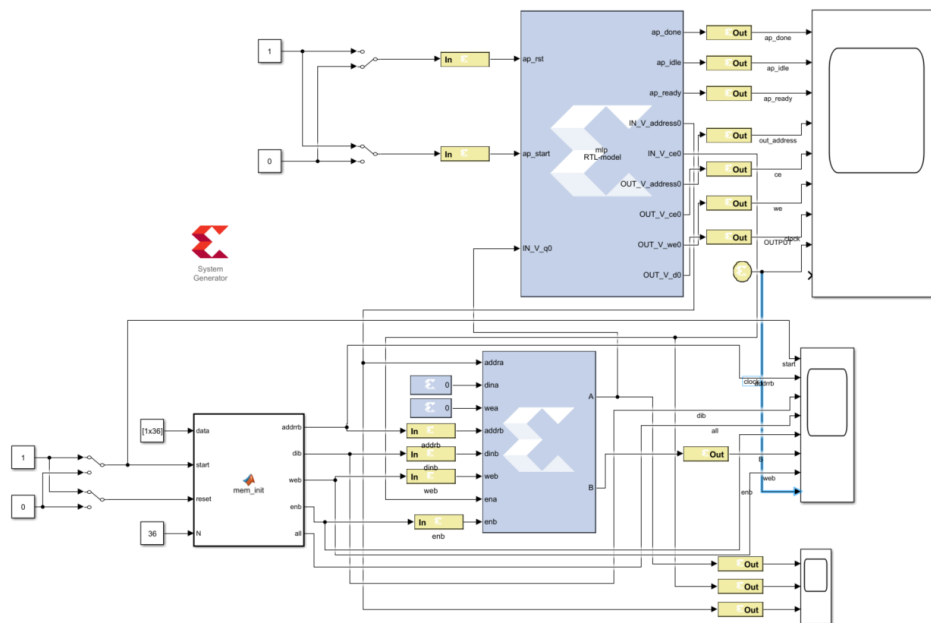
38. ábra: A rendszer tömbvázlata.

Egy Pythonban megírt program a felhasználói paraméterek alapján felépíti a neuronhálót és elvégzi a tanítást, majd generálja a C++ állományt a Vivado HLS számára. A HLS eszköz szintetizálja a C++ kódot és exportálja a kiépített neuronháló IP magját. Az IP mag beillesztődik egy Vivado-ban felépített rendszerbe, ahol egy BRAM-on keresztül kapja meg az adatokat. A Vivado szintetizálja a rendszert, elvégzi az implementálást (fordítás, leképezés, elhelyezés, huzalozás) és generálja a bit fájlt és felfprogramozza az FPGA áramkört. A HLS és Vivado feladatainak automatizálása Tcl szkriptekkel lesz megoldva. Az FPGA BRAM-jába a felhasználó kell betöltse az adatokat egy interfészen keresztül. A jelenlegi rendszerben az interfész kérdése még nincs megoldva. Az eredeti terv az interfészelés megoldására a Xilinx által tervezet ChipScopy alkalmazása volt, amin keresztül Co-Simulation szerűen virtuális kibemeneteken (VIO) keresztül lehetett volna az adatokat átadni. Sajnos ez a könyvtár még nem elérhető, így kézen fekvő megoldást jelentene egy USB interfész megvalósítása, amelyen keresztül akár 480 Mb/s-os sebesség érhető el.

A rendszer kivan építve Simulink környezetben is System Generator használatával. Ennek nagy előnye, hogy megoldja az interfész kérdését, egy Matlab modul segítségével számítógépről tudjuk küldeni a neuronháló bemeneti adatait USB-n keresztül.



39. ábra: Vivado-ban felépített rendszer.



40. ábra: Simulink-ben System Gnerator-al felépített rendszer.

7. Kitűzött példa feladat

7.1. A feladat leírása

Feltételezzük, hogy szeretnénk egy neurális hálót kialakítani, ami képes legyen különböző számjegyeket felismerni. A számjegyeket egy 7*5-ös mátrixszal ábrázoljuk.

a ketős számjegy ábrázolás:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \text{ és az elvárt kimenet: } \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

41. ábra: A számjegyek ábrázolása a feladatban.

A háló kimenetén “one hot” kódolást használunk, ami azt jelenti, hogy a kimeneti rétegben található 10 neuron közül az lesz aktív amelyikhez tartozik a kimenet

7.2. A feladat megoldása

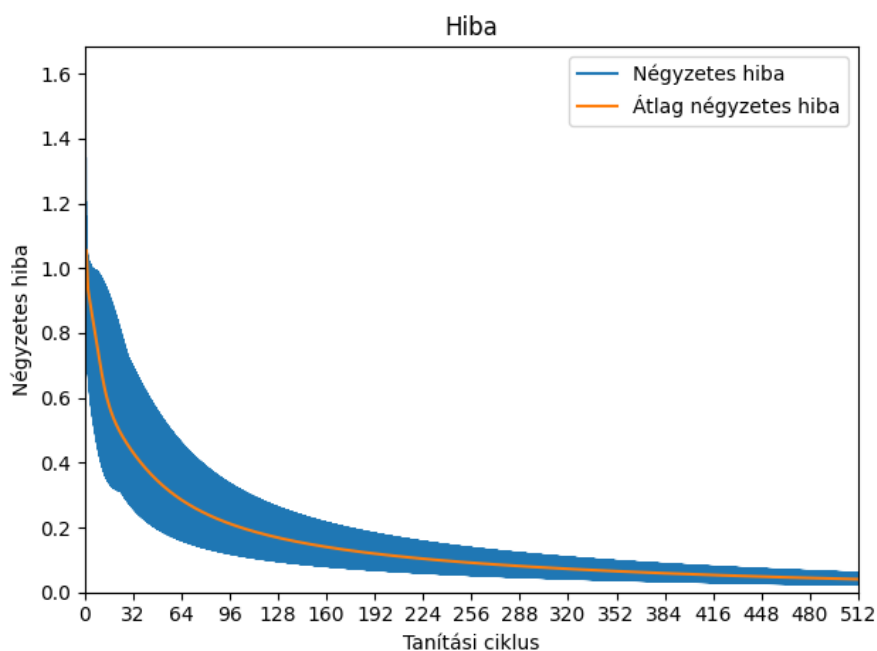
Terveztünk több architektúrát, ami képes a feladat megoldására, mindenik megoldásnak a bemenetén 36 elem van (35 adat és egy “bias”):

- Egy 2 rétegűt, aminek csak 10 neuronja van a kimeneti rétegben.
- Egy 3 rétegűt, aminek van egy 56 neuronból álló rejtett rétegje és a 10 neuron a kimeneti rétegében
- Egy 5 rétegűt, aminek van 3 rejtett rétegje (128, 64, 32) és egy 10 neuront tartalmazó kimeneti rétegje

A 3 különböző topológiájú háló tanítása során a kiinduló tanítási paraméterek azok megegyeztek: tanítási együttható (u): 0.08; momentum együttható (m): 0.005.

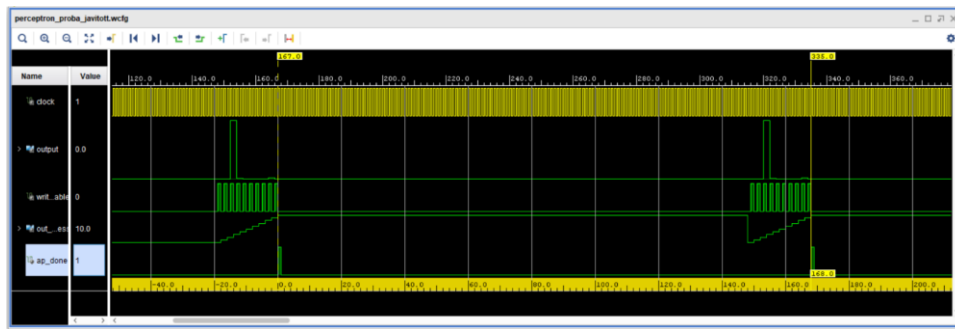
Valamint mindhárom esetén a szigmoid volt használva, mint aktivációs függvény. A kigenerált hálóban a fixpontos számok (5 bit az egészrésznek és 9 bit a törtrésznek).

Az alábbi méréseknél a 24. ábrán látható mérésnek a direktíva konfigurációját használtuk.



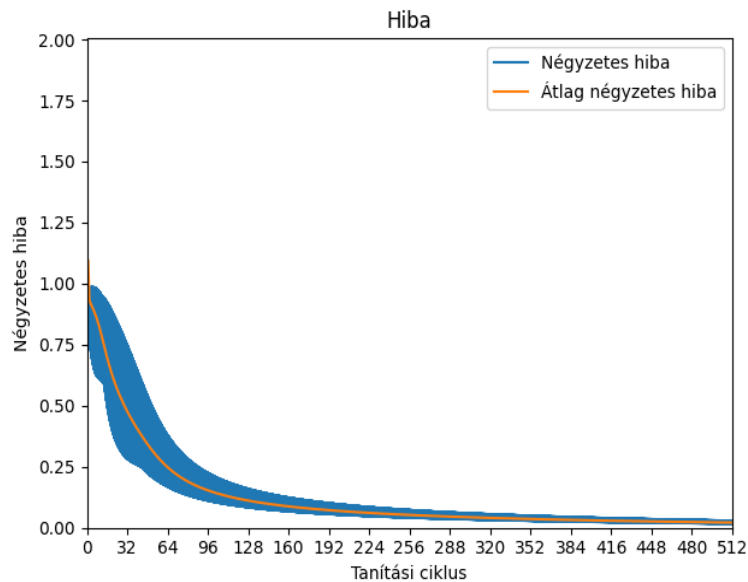
42. ábra: Tanítás (35->10).

Az első háló tanítása az megközelítőleg 1 percben (50s) telt, ez alatt 360 paramétert tanított meg. A fenti ábrán (42. ábra) látható a négyzetes hiba alakulása.



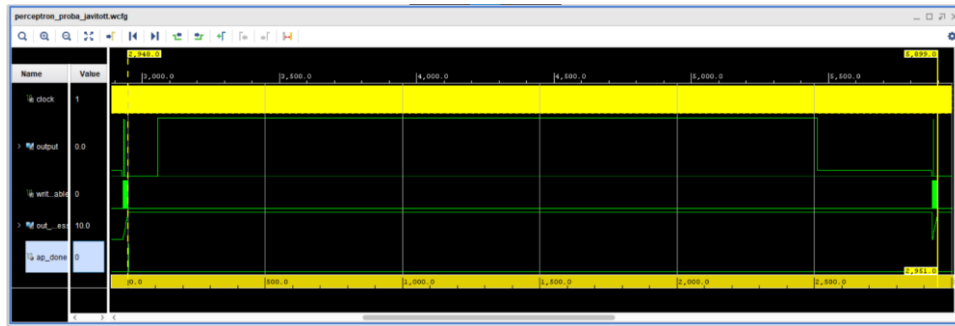
43. ábra: Szimuláció (35->10).

Az első mérés során a rendszer nagy gyorsítást hozott (42-szeres) a processzoron futatott Python modellhez képest. 168 minta telt el a rendszer indításától a kimenet kiszámolásáig. Ha ezt 100 MHz-es órajelen futtatjuk 1,68 μ s-ot jelent. Annak ellenére, hogy a HLS modellben az adatok típusa csak 14 bites fixpontos volt, melyből 5 bit képviselte az egész részt és 9 bit a törtrészt, minimális volt az eredmények különbsége Python-os modellhez képest (0.0052 átlagos négyzetes eltérés).



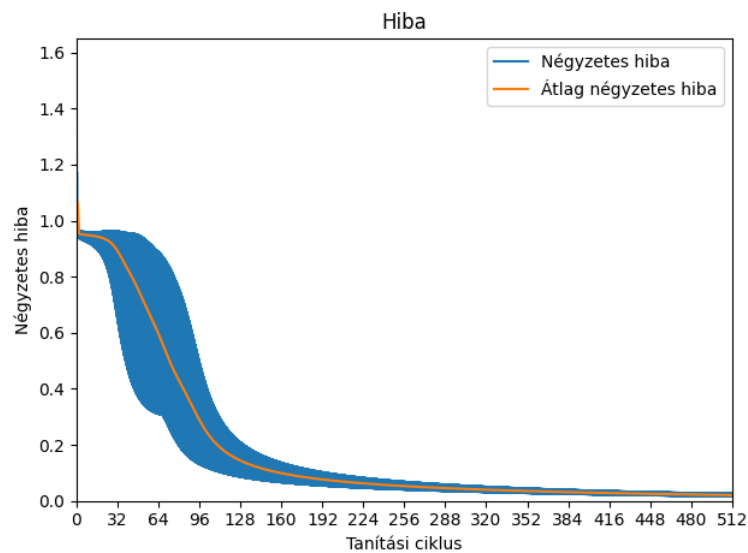
44. ábra: Tanítás (35->56->10).

Egy rejtett réteg bevezetésével a tanítási idő csak keveset haladta meg az egy percet, viszont 3-szor annyi paramétert optimalizált.



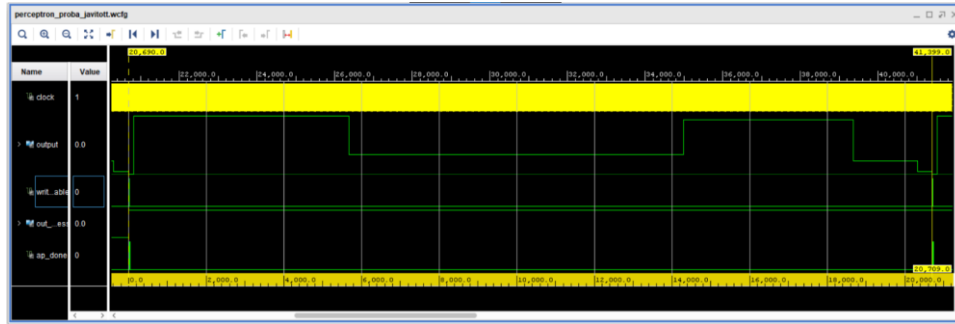
45. ábra: Szimuláció (35->56->10).

A második mérés során a rendszer gyorsítása kisebb volt (közel 11-szeres), de ez még mindig jó gyorsításnak számít. A rendszer 2951 minta alatt vett adta meg a kimenetet. A két modell közötti átlagos négyzetes eltérés nagyjából megegyezik az előző neuronhálónál mért értékkel (0.0054).



46. ábra: Tanítás (35->128->64->32->10).

Több rejtett réteg bevezetésével a tanítás majdnem 2 percbe (117s) telt, és 15274 paramétert tanított.



47. ábra: Szimuláció (35->128->64->32->10).

A harmadik mérésnél kapott gyorsítás már nem annyira jelentős mértékű (3-szoros). A két modell közötti átlagos négyzetes eltérés megduplázódott az előző neuronhálón végzett méréshez képest (0.0107), de még mindig jelentéktelennek tekinthető.

8. Eredmények

A mérés során a Python modell Ubuntu 20.10 operációs rendszeren, i7-7500u processzoron volt mérve, ami megközelítőleg 3.5GHz-n futott. Az FPGA-n való futási időt azt szimulációból becsültük, azzal a feltételezéssel, hogy az FPGA órajele az 100MHz. Becslések szerint akár 42-szeres gyorsítás is elérhető.

#	Rétegek száma	Tanítható paraméterek	Tanítási idő (s)	Futás idő Pythonban (µs)	Futás idő FPGA-n (becsült) (µs)
1	2 (35, 10)	360	50	72	1,68
2	3 (35, 36, 10)	1666	67	319	29,51
3	5(35, 128, 64, 32, 10)	31786	107	606	207,09

2. táblázat: A modellek tanítási és futási ideje.

#	Rétegek száma	Python modell négyzetes hibájának az átlaga	HLS modell négyzetes hibájának az átlaga	Modellek közötti eltérés
1	2	0.0401977984465673	0.0398001911983511	0.0052413841091167
2	3	0.0213411106963915	0.0205169712748613	0.0054848397495561
3	5	0.0210445014637294	0.0231401516365874	0.0107506224136835

3. táblázat: A modellek négyzetes hibáinak az átlaga.

Amint látható az előbbi táblázatban, annak ellenére, hogy a szimulált HLS modellben az adat típus sokkal kisebb és egyszerűbb a két megvalósítás közti négyzetes eltérésének az átlaga csak 1%-os. Meglepő módon az első két mérésnél a HLS modell átlagos négyzetes hibája kisebb lett, mint a Python modellé.

9. Következtetések

A HLS-ben és Python-ban, azonos neuronhálóval és azonos bemenetekkel végzett mérések alapján megfigyelhető, hogy az FPGA áramkörön elért gyorsítás jelentős (42-szeres) a legkisebb neuronhálóval végzett mérés esetében. A második nagyobb méretű neuronhálónál kisebb gyorsítás látható (közel 11-szeres), míg a legnagyobb neuronhálónál a gyorsítás csak 3-szoros a processzoron futtatott Python-ban felépített modellhez képest. Nagyobb modell esetén kisebb gyorsítást értünk el. Ez nagyrészt annak köszönhető, hogy az inger kiszámolására használt ciklust 4-es faktorral terjesztettük ki. Kisebb modell esetén ez nagyobb fokú párhuzamosítást jelent. Valószínűleg más direktíva konfigurációval jobb eredmények lennének elérhetőek. Különböző méretű és típusú neuronhálóknak különböző direktíva konfiguráció szükséges a legjobb eredmény eléréséhez. Ha az adat típusokat jobban optimalizáljuk akkor csökkenthetjük a szükséges erőforrásokat, ez által több erőforrást alkalmazhatunk majd párhuzamosításra.

Jelenleg a rendszer csak a háló paramétereit tudja generálni, viszont a továbbiakban szeretnénk, hogy a hardware architektúráját (csővezetékesítés, párhuzamosítás stb.) is tudja beállítani (és azt ne a felhasználónak kelljen kézből beállítani a HLS projektben.)

Ugyanakkor a rendszer nincs teljesen automatizálva, azaz a generált fájlt be kell másolni egy létező Vivado HLS projektbe (vagy egy projektet létrehozni a fájl köré). Viszont ezt lehetne teljesen automatizálni, ami megkönnyítheti és felgyorsíthatja a folyamatot.

Szeretnénk további neuronháló modelleket kiépíteni: RBF, Kohonen, CMAC. Végző célunk a napjainkban népszerű konvolúciós neuronháló megvalósítása.

10. Irodalomjegyzék

- [1] Van Gerven, M. and Bohte, S., 2017. Artificial neural networks as models of neural information processing. *Frontiers in Computational Neuroscience*, 11, p.114.
- [2] Liang, S., Yin, S., Liu, L., Luk, W. and Wei, S., 2018. FP-BNN: Binarized neural network on FPGA. *Neurocomputing*, 275, pp.1072-1086.
- [3] Coussy, P., Gajski, D.D., Meredith, M. and Takach, A., 2009. An introduction to high-level synthesis. *IEEE Design & Test of Computers*, 26(4), pp.8-17.
- [4] O'Loughlin, D., Coffey, A., Callaly, F., Lyons, D. and Morgan, F., 2014. Xilinx vivado high level synthesis: Case studies.
- [5] Xilinx, 2021, HLS Key Documents and Getting Started 23 április 2021, <<https://forums.xilinx.com/t5/High-Level-Synthesis-HLS/HLS-Key-Documents-and-Getting-Started-FAQ/td-p/1118001>>
- [6] Xilinx, 2018, HLS Pragmas, 23 április 2021, <https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/okr1504034364623-1.html>
- [7] Lahti, S., Sjövall, P., Vanne, J. and Hämäläinen, T.D., 2018. Are we there yet? A study on the state of high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(5), pp.898-911.
- [8] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L. and Desmaison, A., 2019. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*.
- [9] Al-Rfou, R., Alain, G., Almahairi, A., Angermueller, C., Bahdanau, D., Ballas, N., Bastien, F., Bayer, J., Belikov, A., Belopolsky, A. and Bengio, Y., 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, pp.arXiv-1605.
- [10] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S. and Darrell, T., 2014, November. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia* (pp. 675-678).
- [11] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M. and Kudlur, M., 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)* (pp. 265-283).

- [12] Shatnawi, A., Al-Bdour, G., Al-Qurran, R. and Al-Ayyoub, M., 2018, April. A comparative study of open source deep learning frameworks. In 2018 9th international conference on information and communication systems (icics) (pp. 72-77). IEEE.
- [13] Goldsborough, P., 2016. A tour of tensorflow. arXiv preprint arXiv:1610.01178.
- [14] Noronha, D.H., Gibson, K., Salehpour, B. and Wilton, S.J., 2018, December. Leflow: Automatic compilation of tensorflow machine learning applications to fpgas. In 2018 International Conference on Field-Programmable Technology (FPT) (pp. 393-396). IEEE.
- [15] Guan, Y., Liang, H., Xu, N., Wang, W., Shi, S., Chen, X., Sun, G., Zhang, W. and Cong, J., 2017, April. 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) (pp. 152-159). IEEE.
- [16] Kastner, R., Matai, J. and Neuendorffer, S., 2018. Parallel programming for FPGAs. arXiv preprint arXiv:1805.03648.
- [17] Finnerty, A. and Ratigner, H., 2017. Reduce power and cost by converting from floating point to fixed point. WP491 (v1. 0).
- [18] Brassai, S.T., 2019. Neurális hálózatok és fuzzy logika, Scientia kiadó.