

Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH



Instytut Systemów Elektronicznych

Praca dyplomowa magisterska

na kierunku Elektronika
w specjalności Mikrosystemy i Systemy Elektroniczne

Implementacja sztucznych sieci neuronowych w systemach
SoC i MPSoC z wykorzystaniem akceleratorów, realizowanych
w technice HLS

Krzysztof Wasilewski

Numer albumu 265956

promotor

dr inż. Wojciech Zabołotny

WARSZAWA 2020

Implementacja sztucznych sieci neuronowych w systemach SoC i MPSoC z wykorzystaniem akceleratorów, realizowanych w technice HLS

Streszczenie

Sztuczne Sieci Neuronowe są w dzisiejszych czasach wykorzystywane w wielu zastosowaniach. Duża złożoność algorytmów Sztucznej Inteligencji wymaga dużej mocy obliczeniowej oraz odpowiednich metod optymalizacji i właściwego wyboru sprzętu. W przypadku obliczeń, które można zrównoleglić, rozsądnym wyborem są układy FPGA (ang. *Field Programmable Gate Array*).

Główym celem pracy było zaprojektowanie i implementacja sztucznej sieci neuronowej przy wykorzystaniu systemu SoC (ang. *System on Chip*) z układem FPGA i techniki HLS (ang. *High Level Synthesis*). Użycie techniki HLS pozwala na projektowanie przy wykorzystaniu języka C, C++ lub System C i umożliwia korzystanie z wielu bibliotek zaimplementowanych w języku C i C++, co znacznie przyspiesza pracę nad projektem.

Założeniem pracy było stworzenie akceleratora, umożliwiającego osiągnięcie wzrostu wydajności algorytmu detekcji obiektów znajdujących się na obrazie w czasie rzeczywistym. W ostatnich latach można zaobserwować wielki postęp w dziedzinie komputerowego rozpoznawania obrazów (ang. *Computer Vision*), jednak większość dostępnych implementacji jest przeznaczona do uruchomienia na komputerze PC. W pracy dokonano porównania wydajności implementacji wykorzystującej układ FPGA do akceleracji obliczeń i wysokopoziomowego rozwiązania programowego, testowanego na komputerze PC.

Słowa kluczowe: Sztuczne Sieci Neuronowe, HLS, komputerowe rozpoznawanie obrazów, FPGA

Artificial Neural Networks implementation in SoC and MPSoC systems using accelerators synthesized by HLS method

Abstract

Nowadays, Artificial Neural Networks (ANN) are used in many applications. High complexity of Artificial Intelligence algorithms requires high computing power, appropriate optimization methods and efficient hardware. In the case of computation that is easy to parallelize it is reasonable to use FPGA systems.

The main aim of the thesis was to design and implement an Artificial Neural Network algorithm using SoC (System on Chip) with FPGA system and HLS (High-Level Synthesis) method. HLS method allows to design a project using C, C++ or System C language and use lots of C libraries, which makes working on the project faster.

Assumption was made that the created accelerator will allow to achieve efficiency improvement in the real-time object detection algorithm. It has been seen recently, that huge improvement was made in the field of Computer Vision, but most of the available implementations are made to run on PC. In the thesis a performance comparison was made between implementation using FPGA to accelerate the calculations and high level software solution, tested on PC.

Keywords: Artificial Neural Networks, HLS, Computer Vision, FPGA



.....
miejscowość i data

.....
imię i nazwisko studenta

.....
numer albumu

.....
kierunek studiów

OŚWIADCZENIE

Świadomy/-a odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie, pod opieką kierującego pracą dyplomową.

Jednocześnie oświadczam, że:

- niniejsza praca dyplomowa nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- niniejsza praca dyplomowa nie zawiera danych i informacji, które uzyskałem/-am w sposób niedozwolony,
- niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanego z nadawaniem dyplomów lub tytułów zawodowych,
- wszystkie informacje umieszczone w niniejszej pracy, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami,
- znam regulacje prawne Politechniki Warszawskiej w sprawie zarządzania prawami autorskimi i prawami pokrewnymi, prawami własności przemysłowej oraz zasadami komercjalizacji.

Oświadczam, że treść pracy dyplomowej w wersji drukowanej, treść pracy dyplomowej zawartej na nośniku elektronicznym (płycie kompaktowej) oraz treść pracy dyplomowej w module APD systemu USOS są identyczne.

.....
czytelny podpis studenta

Spis treści

1. Wstęp	11
1.1. Wprowadzenie	11
1.2. Wstęp teoretyczny	11
1.2.1. Model neuronu	12
1.2.2. Funkcja aktywacji	13
1.2.3. Perceptron wielowarstwowy	15
1.2.4. Sieci Głębokie	15
1.2.5. Proces uczenia sieci	16
1.2.6. Wsteczna propagacja błędu	17
1.2.7. Hiperparametry	17
1.2.8. Przeuczenie sieci	17
1.2.9. Splotowe Sieci Neuronowe	17
1.3. Zastosowania Sztucznych Sieci Neuronowych	18
2. Cel i zakres pracy	21
2.1. Motywacja	21
2.2. FPGA kontra GPU	21
2.3. Implementacja Sztucznych Sieci Neuronowych w układach FPGA	21
2.3.1. Reprezentacja liczb zmiennoprzecinkowych	21
2.3.2. Uczenie Sztucznej Sieci Neuronowej w układach FPGA	21
3. Wybór sprzętu	23
3.1. Xilinx Zynq-7000	23
3.2. Z-turn Board	24
3.2.1. Interfejsy komunikacyjne	25
3.3. Kamera	25
3.3.1. Moduł MY-CAM011B	26
3.3.2. Moduł MY-CAM002B	28
4. Implementacja	31
4.1. Budowa systemu	31
4.1.1. Schemat blokowy systemu	31
4.2. Wybór narzędzi	31
4.3. Projekt systemu w środowisku Vivado	32
4.3.1. Pamięć Block RAM	32
4.4. Wykorzystanie metody HLS	34
4.5. Zbiór danych wejściowych	35
4.6. Opracowanie modelu ANN	35
4.6.1. Uczenie Sztucznej Sieci Neuronowej	36
4.6.2. Implementacja modelu przy użyciu narzędzia Vivado HLS	36

4.7. Synteza projektu w narzędziu Vivado	37
4.8. Implementacja przy użyciu narzędzia Vitis	38
4.9. Test z wykorzystaniem systemu operacyjnego Petalinux	41
4.9.1. Wstępna konfiguracja systemu Petalinux	41
4.9.2. Konfiguracja jądra systemu Petalinux	42
4.9.3. Konfiguracja systemu plików	43
4.9.4. Przygotowanie obrazu systemu	44
4.9.5. Uruchomienie systemu Petalinux	44
5. Wyniki i wnioski	47
5.1. Test modelu sieci z jedną warstwą ukrytą	47
5.1.1. Test klasyfikacji cyfr z użyciem kamery	48
5.1.2. Test na płytce Z-turn z użyciem kamery	49
5.2. Test modelu posiadającego dwie warstwy ukryte	49
6. Podsumowanie	51
Bibliografia	53
Wykaz symboli i skrótów	54
Spis rysunków	55
Spis tabel	55

1. Wstęp

1.1. Wprowadzenie

W ostatnich latach można zaobserwować gwałtowny rozwój w dziedzinie Uczenia Maszynowego (ang. *Machine Learning*) i Sztucznej Inteligencji (ang. *Artificial Intelligence*). Coraz więcej urządzeń i systemów projektowanych jest na działanie autonomicznie, bez potrzeby ingerencji człowieka. Jednym z algorytmów, który powstał już dość dawno, są Sztuczne Sieci Neuronowe (ang. *Artificial Neural Networks*). Temat Sieci Neuronowych ma długą historię rozwoju, sięgającą początku lat 40. XX wieku, jednak w ostatnich latach można zaobserwować znaczny postęp w tej dziedzinie [1]. Rozwój technologii umożliwił zastosowanie algorytmów AI w wielu aplikacjach. Działalność naukowa w kierunku Sztucznych Sieci Neuronowych spowodowała powstanie nowych modeli i architektur.

Większość algorytmów wykorzystujących Sztuczną Inteligencję wymaga dużej mocy obliczeniowej i wyboru odpowiedniego sprzętu. Często powtarzaną operacją matematyczną w przypadku algorytmu Sztucznej Sieci Neuronowej jest mnożenie macierzy. Działanie to można w łatwy sposób zrównoleglić, implementując sieć w układzie FPGA i tym samym zwiększyć efektywność algorytmu.

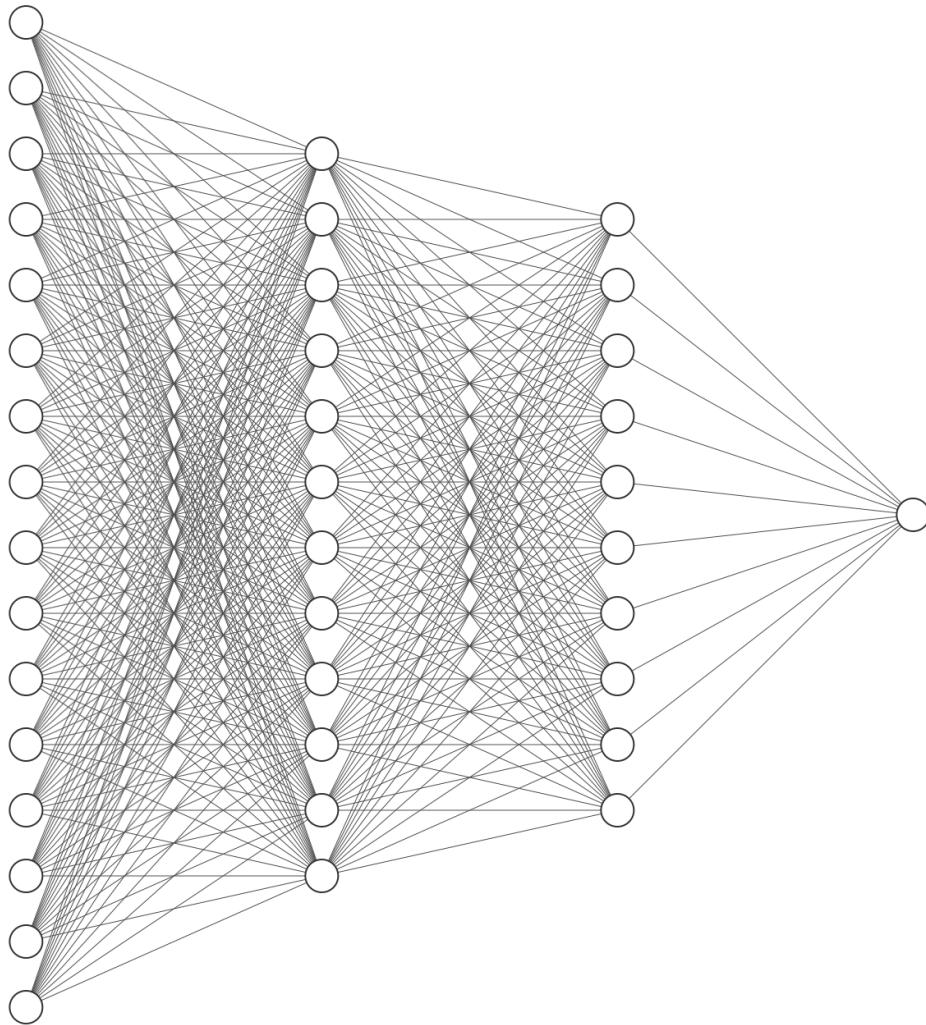
Praca nad implementacją algorytmów Sztucznej Inteligencji w większości przypadków zaczyna się od stworzenia i uruchomienia modelu. Do tego zadania często wykorzystywane są wysokopoziomowe biblioteki takie jak *Keras* lub *Theano*, które w znacznym stopniu przyspieszają proces tworzenia oprogramowania oraz ułatwiają wprowadzanie zmian w modelu sieci. Rozwój i dopracowywanie algorytmu Sztucznej Inteligencji wymaga wielu iteracji uruchamiania kodu z różnymi parametrami i właściwościami sieci. Aby w pełni wykorzystać potencjał Sztucznych Sieci Neuronowych, należy dobrać odpowiednio metody projektowania i testowania modeli.

1.2. Wstęp teoretyczny

Sieć neuronowa jest algorytmem przetwarzającym dane, inspirowanym działaniem mózgu. ANN składa się z wielu elementów przetwarzających informacje – neuronów. Model przetwarzania informacji jest inspirowany ludzkim mózgiem jednak w stosunku do komórki nerwowej, neuron w Sztucznej Sieci Neuronowej jest bardzo uproszczony [2]. Pomimo to ANN umożliwiają rozwiązywanie złożonych problemów w wielu dziedzinach z dużą dokładnością.

Pomiędzy neuronami prowadzone są połączenia, z których każde ma swoją wagę, która zmienia wartość w trakcie uczenia sieci. Warstwę sieci, w której wszystkie wyjścia każdego z neuronów są podłączone do wszystkich neuronów w warstwie następnej, nazywa się warstwą w pełni połączoną (ang. FC — *Fully Connected*). Sieć posiadająca wszystkie warstwy w pełni połączone jest zwana siecią w pełni połączoną.

W znacznej większości sieci neuronowe budowane są w ten sposób, że dane są propagowane w kierunku od warstwy wejściowej do wyjściowej. Takie sieci nazywamy sieciami jednokierunkowymi (ang. *feedforward*). Przykładową sieć neuronową w pełni połączoną typu *feedforward* przedstawiono na Rys. 1.1.



Rysunek 1.1. Schemat w pełni połączonej jednokierunkowej Sieci Neuronowej

1.2.1. Model neuronu

Sztuczne Sieci Neuronowe to algorytm wzorowany działaniem ludzkiego mózgu i znajdujących się w nim neuronów. Model matematyczny (Rys. 1.2) pojedynczego neuronu składa się z następujących elementów [3]:

- wektora wejściowego:

$$x = (x_1, x_2, \dots, x_j)^T$$

- wektora wag przypisanych do każdego z wejść

$$w = (w_{k1}, w_{k2}, \dots, w_{kj})^T$$

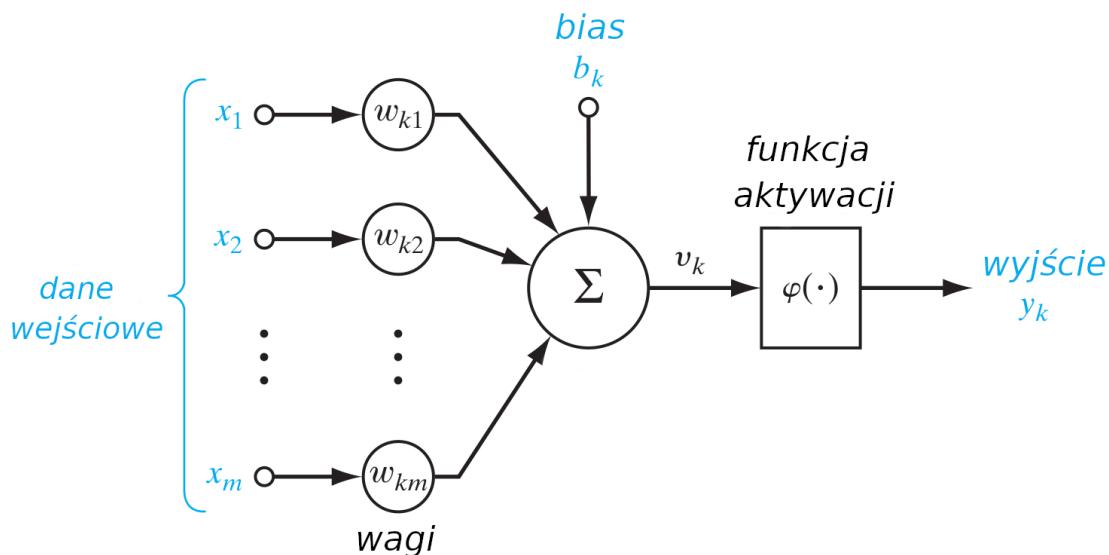
- wag obciążających (bias) b
- funkcji aktywacji $\phi(u)$
- wyjścia neuronu y .

Wagi obciążające (bias) umożliwiają kontrolowanie progu aktywacji neuronu, niezależnie od wartości wejściowych. W praktyce oznacza to przesuwanie wykresu funkcji aktywacji w kierunku w prawą lub w lewą stronę. Wyjście neuronu można policzyć, stosując następujące wzory:

$$u_k = \sum_{j=1}^N w_{kj} x_j$$

$$y_k = \phi(u_k + b_k)$$

Najprostsza Sieć Neuronowa składająca się z jednego neuronu, nazywana jest *Perceptronem*. Algorytm ten można wykorzystać do zadań klasyfikacji binarnej.



Rysunek 1.2. Model neuronu

1.2.2. Funkcja aktywacji

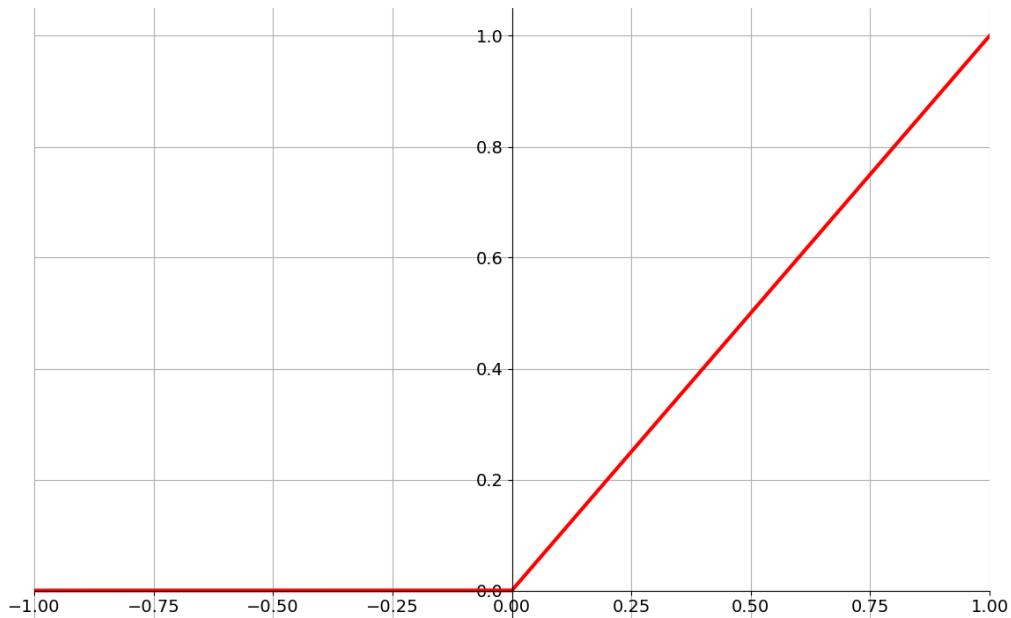
Na działanie algorytmu znaczny wpływ może mieć dobór odpowiedniej funkcji aktywacji. Wśród najczęściej stosowanych funkcji aktywacji wyróżnia się następujące funkcje:

- funkcja ReLu (ang. *Rectified Linear Units*):

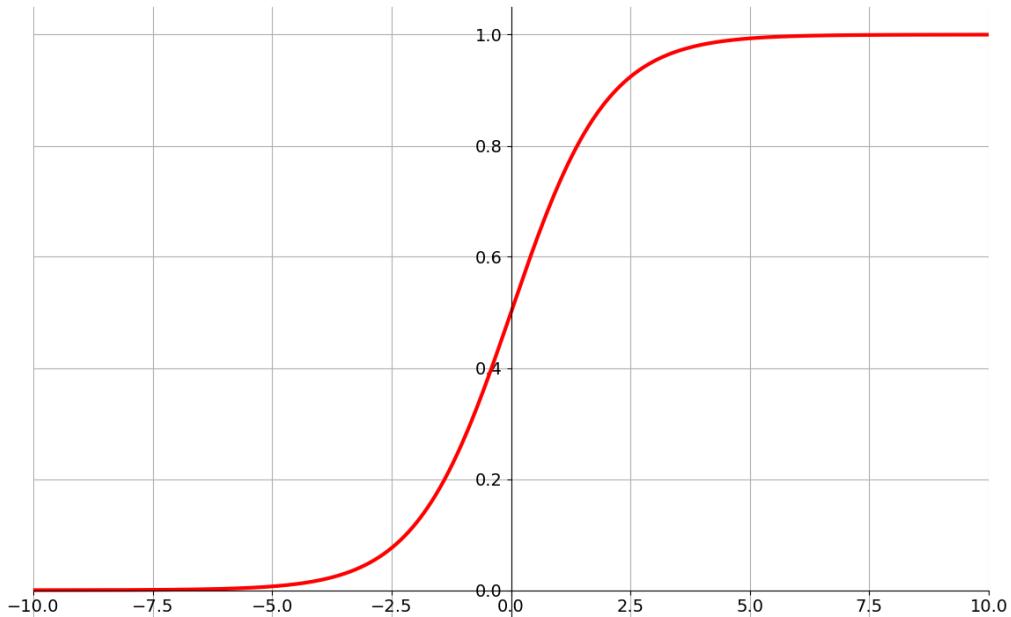
$$\phi(x) = \max(0, x)$$

- funkcja sigmoid:

$$\phi(x) = \frac{a}{a + e^{-bx}}$$



Rysunek 1.3. Wykres funkcji ReLU



Rysunek 1.4. Wykres funkcji sigmoid

- funkcja softmax (liczona dla każdego neuronu warstwy wyjściowej, $j = 1, \dots, N$):

$$\phi(x_j) = \frac{e^{x_j}}{\sum_{k=1}^N e^{x_k}}$$

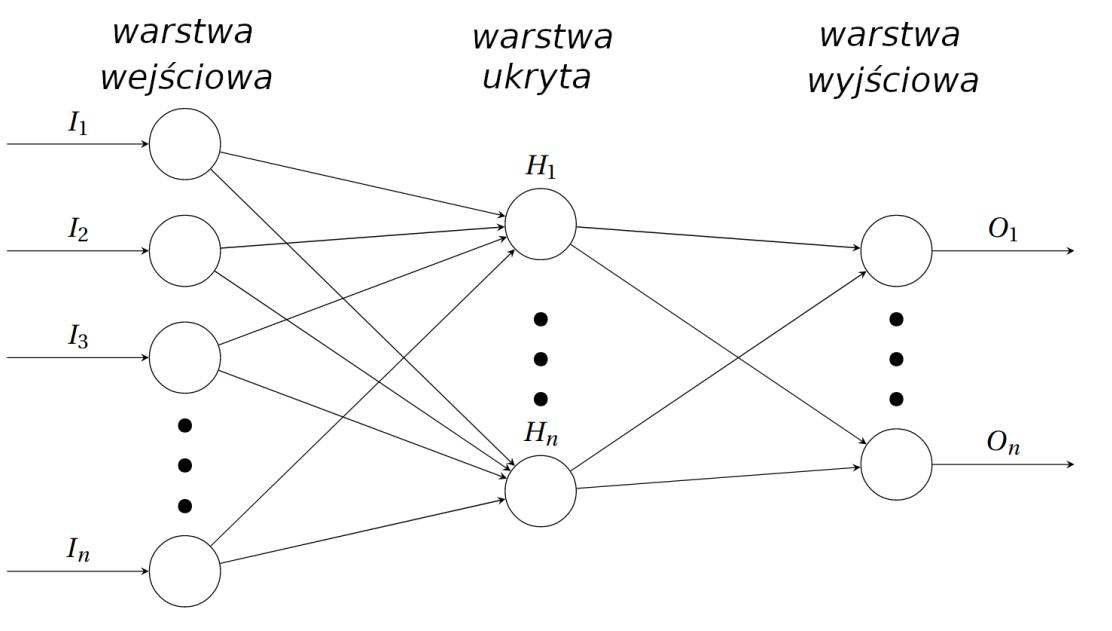
W warstwach ukrytych często używaną funkcją aktywacji jest funkcja ReLU oraz sigmoid. Funkcję sigmoid stosuje się również w warstwach wyjściowych przy zadaniach

klasyfikacji binarnej. Funkcja softmax jest używana w problemach klasyfikacji wieloklasowej (np. rozpoznanie cyfr). Stosując funkcję softmax w warstwie wyjściowej, można traktować wektor wartości wyjść sieci jako rozkład prawdopodobieństwa wyboru danego rozwiązania.

1.2.3. Perceptron wielowarstwowy

Jednym z pierwszych modeli Sztucznych Sieci Neuronowych był Perceptron Wielowarstwowy (ang. MLP — *Multi-Layer Perceptron*), składający się z warstwy neuronów wejściowej, ukrytych i wyjściowej (Rys.1.5). Wyjście neuronów w danej warstwie staje się wejściem neuronów warstwy następnej.

Sieć Neuronowa typu MLP najczęściej posiada warstwy w pełni połączone (ang. FC – *Fully Connected*) lub inaczej Gęste (ang. *Dense*). W warstwie FC każde z wyjść jest podłączone do wszystkich wejść neuronów w warstwie następnej.



Rysunek 1.5. Model Perceptronu Wielowarstwowego

1.2.4. Sieci Głębokie

Rozwój algorytmów AI doprowadził do powstania Głębokich Sieci Neuronowych (ang. DNN — *Deep Neural Networks*), czyli takich, które posiadają wiele warstw ukrytych [4]. Algorytm Ucznia Głębokiego (ang. *Deep Learning*) umożliwia rozwiązywanie skomplikowanych problemów takich jak rozpoznanie i klasyfikację obiektów na obrazie (Rys.1.6). Seria warstw ukrytych (ang. *hidden layers*) umożliwia ekstrakcję cech obiektów. Kolejne warstwy umożliwiają wykrywanie krawędzi, potem konturów, a na końcu całych kształtów i obiektów.



Rysunek 1.6. Model Głębokiego Uczenia sieci

1.2.5. Proces uczenia sieci

Uczenie Sztucznej Sieci Neuronowej można podzielić na nadzorowane (ang. *supervised learning*) i nienadzorowane (ang. *unsupervised learning*) oraz wersję pośrednią – uczenie częściowo nadzorowane (ang. *semi-supervised learning*). W praktyce najczęściej wykorzystywanym rodzajem uczenia jest *supervised learning*, jednak uczenie nienadzorowane, które zakłada tworzenie modelu sieci bez ingerencji człowieka, jest ciekawym tematem badań i rozwoju algorytmów Sztucznych Sieci Neuronowych [5]. Proces uczenia nadzorowanego sieci polega na zwiększeniu dokładności ANN w rozwiązywaniu danego problemu. Odbywa się to poprzez iteracyjne korygowanie wartości wag i biasów tak, aby zminimalizować błąd pomiędzy rozwiązaniem wzorcowym i otrzymanym. Jeden z algorytmów zmiany wag podczas uczenia jest reguła delty (ang. *delta rule*):

$$\Delta w_{kj} = \eta e_k(n) x_j(n)$$

$$e_k(n) = d_k(n) - y_k(n)$$

W powyższym wzorze $d_k(n)$ jest zakładanym rozwiązaniem, a $y_k(n)$ rozwiązaniem otrzymanym. Wartość η oznacza współczynnik szybkości uczenia (ang. *learning rate*). Gdy jest wystarczająco mały, algorytm uczenia osiąga rozwiązanie optymalne, jednak większy współczynnik η może przyspieszyć ten proces. Wpływ na szybkość uczenia ma również

wielkość miniserii (ang. *mini-batch*) uczących. Jest to liczba elementów zbioru uczącego, podawanych na wejście algorytmu, przed zaktualizowaniem wartości wag sieci. Z powodu ograniczonych rozmiarów zboru uczącego, jest on wykorzystywany wielokrotnie. Jedno przejście przez wszystkie próbki ze zbioru uczącego nazywane jest epoką (ang. *epoch*) [6].

1.2.6. Wsteczna propagacja błędu

Architekturę sieci MLP często stosuje się wraz z algorytmem wstecznej propagacji błędu (ang. *Backpropagation*), która umożliwia proces uczenia sieci. Poprzez obliczenie błędu w neuronach warstwy wyjściowej i propagacji wstecz błędu przez całą sieć, algorytm pozwala dostosować wartość wagi każdego z połączeń w taki sposób, aby zminimalizować wartość funkcji kosztu. Sieć jest uczona do momentu, gdy błąd stanie się akceptowalnie mały.

1.2.7. Hiperparametry

W procesie uczenia Sztucznej Sieci Neuronowej bardzo istotne jest odpowiednie ustawienie parametrów sieci, zwanych hiperparametrami (ang. *hyperparameters*). Są to parametry sieci, ustawiane przez użytkownika w celu otrzymania modelu, dającego najlepsze rozwiązania. Na wynik uczenia wpływ mają m.in. następujące Hiperparametry:

- liczba i rodzaj warstw sieci
- liczba neuronów w każdej z warstw
- współczynnik szybkości uczenia η
- wielkość miniserii
- liczba epok.

1.2.8. Przeuczenie sieci

Przy używaniu Sztucznych Sieci Neuronowych w zadaniach klasyfikacji często spotykanym problemem jest przeuczenie sieci (ang. *overfitting*). Przeuczona sieć jest zbyt mocno dopasowana do danych uczących i przy walidacji poprawność klasyfikacji jest znacznie mniejsza. Jednym ze sposobów na polepszenie zdolności do generalizacji trenowanego modelu sieci jest zastosowanie *Dropoutu*, czyli losowego usuwania pewnej ustalonej liczby neuronów. Technika ta, pomimo swojej prostoty, daje bardzo dobre wyniki w przeciwdziałaniu przeuczeniu sieci.

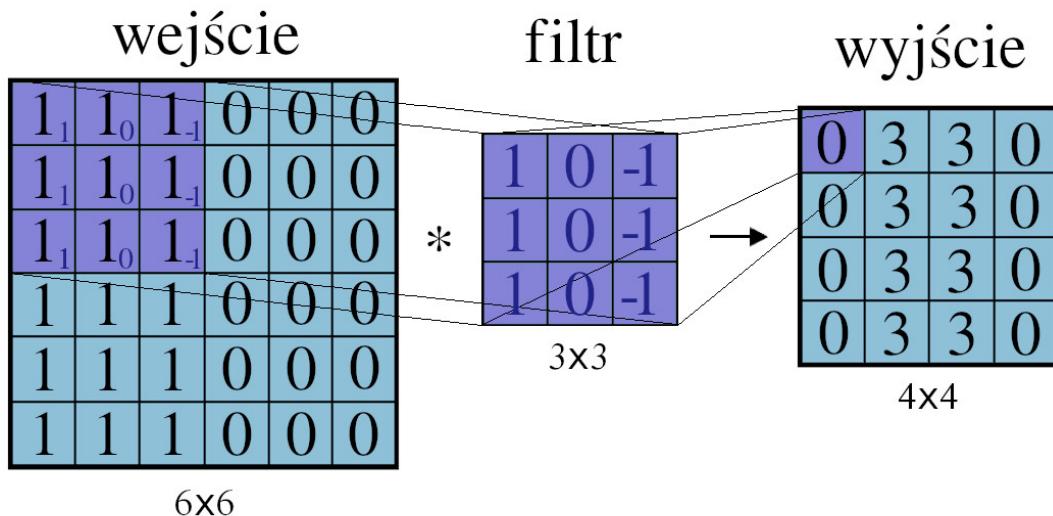
1.2.9. Splotowe Sieci Neuronowe

Splotowe Sieci Neuronowe (ang. *CNN – Convolutional Neural Networks*) są typem sieci głębokich, które znakomicie nadają się do rozpoznawania kształtów i z powodzeniem stosowane w wielu zagadnieniach klasyfikacji obrazów. Charakteryzują się dużą odpornością na rotację, skalowanie, zniekształcanie i inne zakłócenia [7]. Sieci te są uczone w trybie nadzorowanym.

Za ekstrakcję cech obiektów w dwuwymiarowym obrazie odpowiedzialna jest warstwa splotowa sieci. Wejście M tej warstwy jest przekształcane przy użyciu odpowiedniego filtra K (ang. *filter*). Operację splotu w sieciach CNN można opisać wzorem:

$$S(i, j) = (M * K)(i, j) = \sum_k \sum_j M(i - k, j - l)K(k, l)$$

Wynik $S(i, j)$ nazywany jest mapą atrybutów (ang. *feature map*). Warstwa splotowa umożliwia zmniejszenie rozmiaru przetwarzanych danych podczas uczenia sieci oraz zwiększa jej zdolność do generalizacji. Operację splotu przedstawiono na Rys. 1.7. Zaprezentowany przypadek to splot macierzy 6x6 przy użyciu filtru o rozmiarze 3x3.



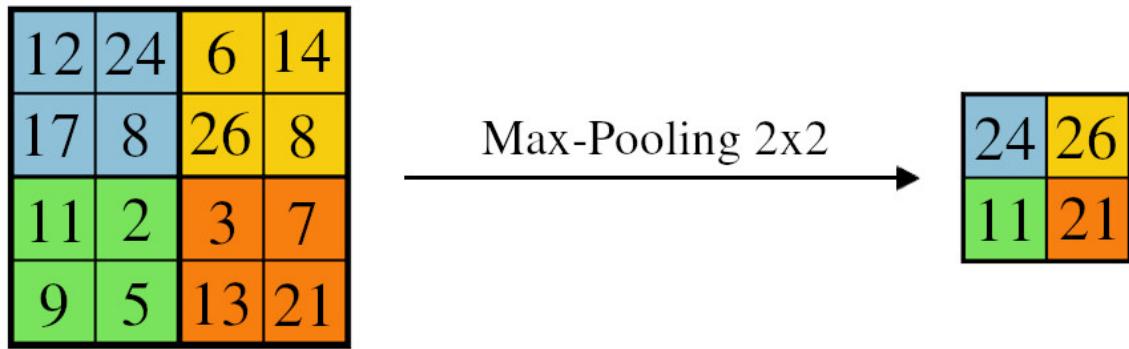
Rysunek 1.7. Splot macierzy 6x6 przy użyciu filtru o rozmiarze 3x3

Razem z warstwą splotową często stosowana jest warstwa grupowania (ang. *pooling*), występującą najczęściej w jednym z dwóch wariantów (ang. *max-pooling*) lub (ang. *average pooling*). W przypadku przetwarzania obrazu metoda (ang. *max-pooling*) polega na zmniejszeniu jego rozmiaru, poprzez wybranie wartości maksymalnej lub średniej, stosując okno o wybranym rozmiarze. Najczęściej wybieranym krokiem (ang. *stride*) przesuwania okna jest wartość, równa jego rozmiarowi. Na Rys. 1.8 przedstawiono przykład *max-poolingu* o rozmiarze 2x2 ze skokiem równym rozmiarowi okna.

1.3. Zastosowania Sztucznych Sieci Neuronowych

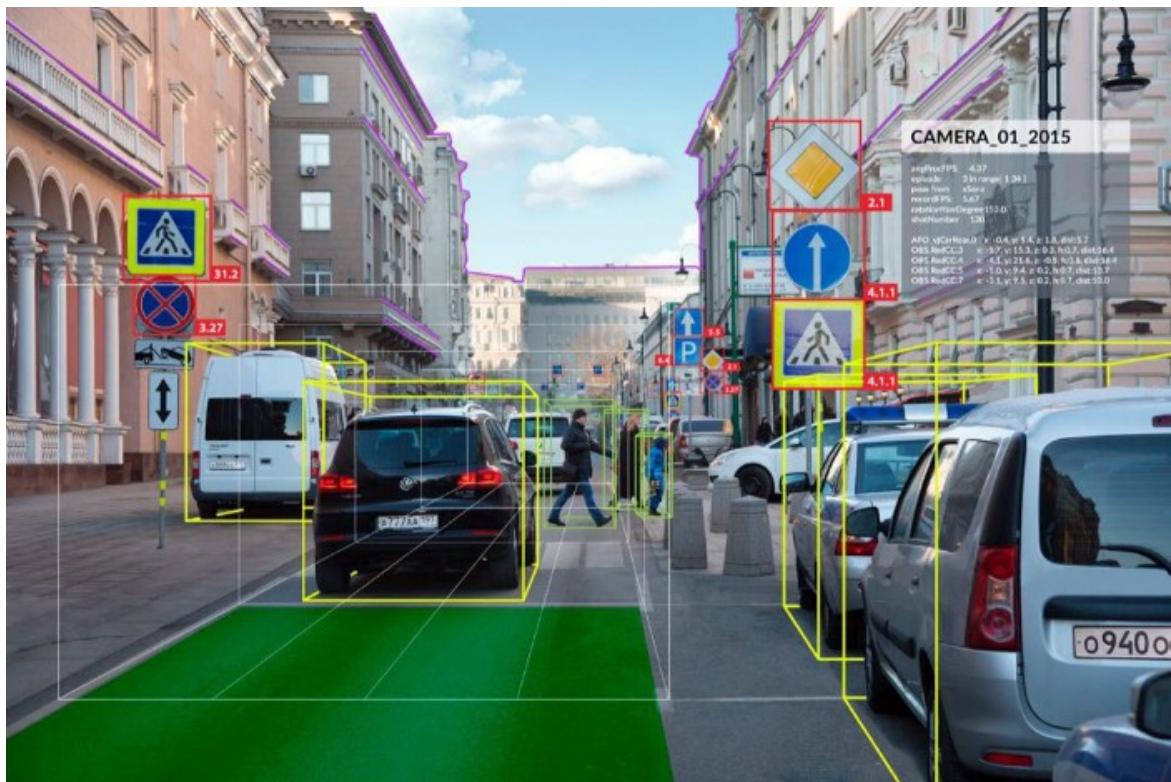
W dzisiejszych czasach Sztuczne Sieci Neuronowe są wykorzystywane w wielu zastosowaniach. Dziedziny, w których stosuje się algorytmy ANN to m.in.:

- medycyna – wspomaganie wystawiania diagnozy
- przemysł – automatyczna kontrola jakości produktów
- zastosowania militarne – wspomaganie radarów i termowizji



Rysunek 1.8. Max-pooling o rozmiarze 2x2

- ekonomia – przewidywanie notowań giełdowych
 - robotyka – autonomiczne samochody (Rys. 1.9)
 - rozpoznawanie i synteza mowy – sterowanie różnymi systemami (smartfon, komputer, urządzenia IoT) przy użyciu asystenta głosowego



Rysunek 1.9. Rozpoznawanie obiektów na obrazie z kamery samochodu autonomicznego

2. Cel i zakres pracy

Celem pracy było zaprojektowanie i implementacja sztucznej sieci neuronowej przy wykorzystaniu systemu SoC (ang. System on Chip) i techniki HLS. Użycie metody HLS pozwala na projektowanie przy wykorzystaniu języka C, C++ lub System C, co przyspiesza pracę nad projektem. Dodatkowo HLS umożliwia korzystanie z wielu bibliotek, które pozwalają wygodnie używać funkcji, które są wykorzystywane w implementacji Sztucznych Sieci Neuronowych. Efektem pracy powinno być stworzenie akceleratora, umożliwiającego osiągnięcie wzrostu wydajności w stosunku do rozwiązań software'owych.

2.1. Motywacja

Sztuczne Sieci Neuronowe są związane z dużą ilością obliczeń, które mogą być wykonywane równolegle. Pozwala to osiągnąć krótszy czas wykonania programu, co ma duże znaczenie dla zastosowań w systemach działających w czasie rzeczywistym np. w branży *Automotive*. Aby osiągnąć przyspieszenie obliczeń, stosuje się różne metody. Jednym z najpopularniejszych obecnie sposobów na zwiększenie wydajności algorytmów AI jest wykorzystanie kart graficznych GPU (ang. *Graphics Processing Unit*). Metodą najbardziej przyspieszającą obliczenia, lecz wymagającą najdłuższego czasu projektowania i najbardziej kosztowną, jest zastosowanie specjalizowanych układów ASIC (ang. *Application-Specific Integrated Circuit*). Opcją pośrednią pomiędzy powyższymi dwoma rozwiązaniami jest zastosowanie układów FPGA. To podejście umożliwia osiągnięcie znacznego przyspieszenia wykonywania obliczeń i nie powoduje wielkiego wzrostu kosztów. Dodatkowo zastosowanie metody HLS ułatwia i minimalizuje czas tworzenia sprzętowej implementacji modelu ANN oraz wprowadzanie zmian w projekcie.

2.2. FPGA kontra GPU

W większości przypadków Sztuczne Sieci Neuronowe są projektowane, uczone i uruchamiane na procesorze z akceleratorem obliczeń w postaci GPU. Jednak układy graficzne nie zawsze są dostępne w systemach wbudowanych.

Sposobem na przyspieszenie obliczeń związanych z implementacją algorytmów ANN jest zastosowanie układów FPGA. Dodatkowym atutem po stronie układów FPGA jest mniejsze zużycie mocy.

2.3. Implementacja Sztucznych Sieci Neuronowych w układach FPGA

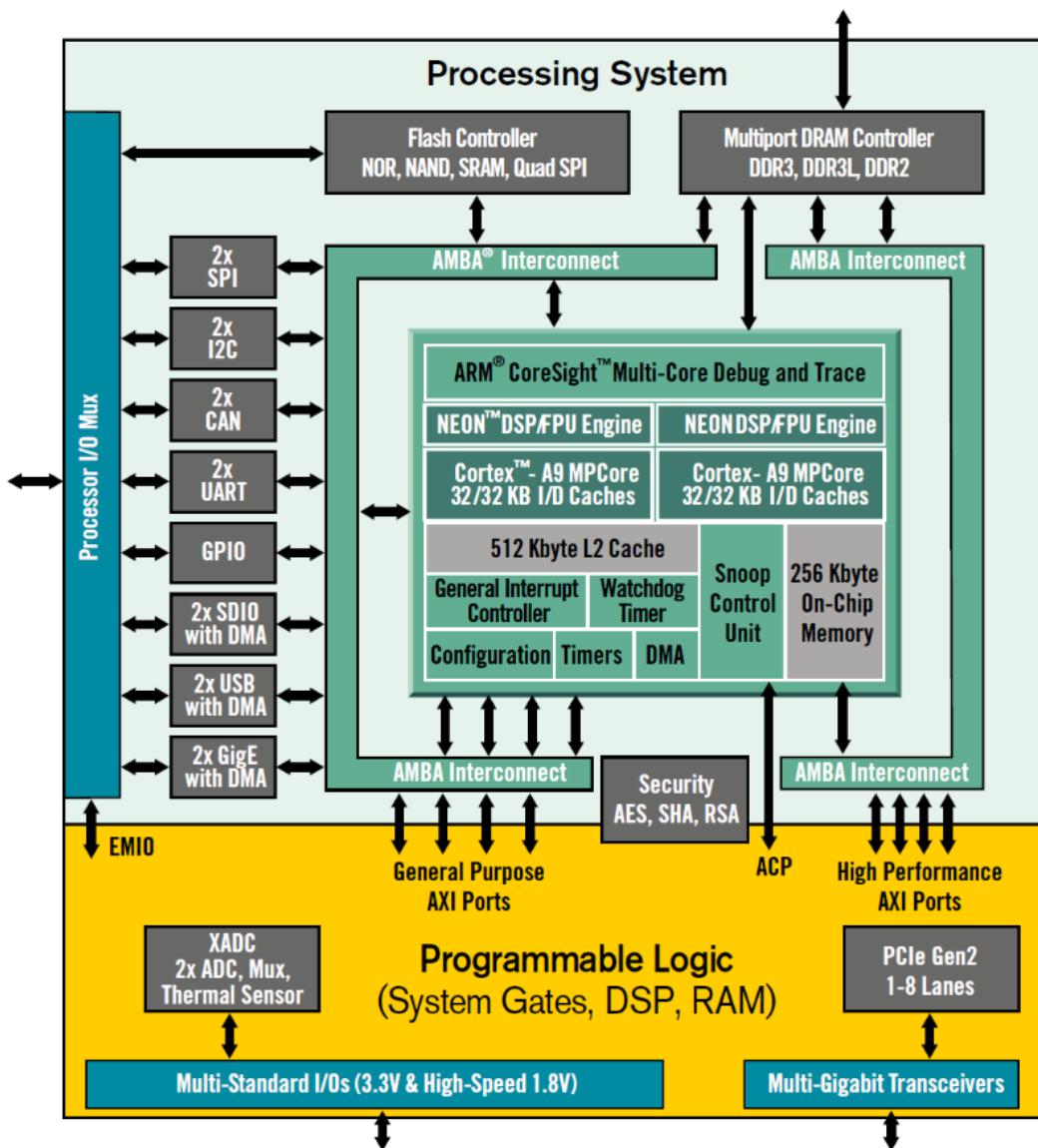
2.3.1. Reprezentacja liczb zmiennoprzecinkowych

2.3.2. Uczenie Sztucznej Sieci Neuronowej w układach FPGA

3. Wybór sprzętu

3.1. Xilinx Zynq-7000

Przy wyborze płytki wzięto pod uwagę posiadane interfejsy oraz cenę sprzętu i dostępność dokumentacji. Podjęto decyzję o zastosowaniu układu FPGA firmy *Xilinx* z racji na duże wsparcie techniczne. W celu optymalizacji kosztów systemu wybrano podstawowe dwa układy z rodziny *Zynq-7000 SoC Family* (Rys. 3.1) XC7Z010 oraz XC7Z020. Aby mieć pewność, że parametry układu będą wystarczające, wybrano układ XC7Z020.



Rysunek 3.1. Architektura serii ZYNQ-7000 SoC

Zgodnie z porównaniem zawartym w Tabeli 3.1, płytka Z-turn Board MYS-7Z020-C-S ma cenę sporo mniejszą od innych płytEK z układem XC7Z020. Dokumentacja techniczna dotycząca płytki Z-turn Board jest dostępna na stronie producenta. Wadą płytki jest małe

3. Wybór sprzętu

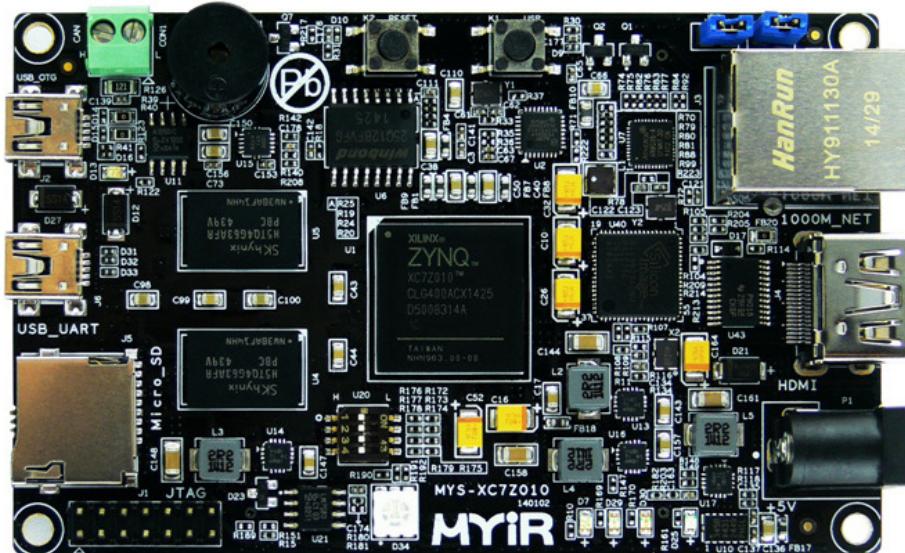
zaangażowanie społeczności w projekty z jej wykorzystaniem, jednak tak konkurencyjna cena zadecydowała o wyborze.

Tabela 3.1. Porównanie cen płytEK z układami Zynq firmy Xilinx

Nazwa płytki	Układ SoC	Cena
Z-turn Board MYS-7Z010-C-S	XC7Z010-1CLG400C	99\$ ¹
Z-turn Board MYS-7Z020-C-S	XC7Z020-1CLG400C	119\$ ¹
Zybo Z7-10 Development Board	XC7Z010-1CLG400C	199\$ ²
Zybo Z7-20 Development Board	XC7Z020-1CLG400C	299\$ ²
ZedBoard Zynq-7000	XC7Z020-CLG484-1	449\$ ³

3.2. Z-turn Board

Z-turn Board (Rys. 3.2) jest komputerem jednoplatformowym (ang. SBC – *Single Board Computer*), opartym o układ SoC Xilinx Zynq-7020 (XC7Z020-1CLG400C), zawierającym dwurdzeniowy procesor ARM Cortex-A9 i układ FPGA Artix 7. Producentem płytki jest firma MYIR Tech Limited (ang. *Make Your Ideas Real*), dostarczająca sprzęt bazujący na procesorach ARM oraz oprogramowanie do swoich produktów[8].



Rysunek 3.2. PłytkA Z-turn-Board 7020

Biorąc pod uwagę parametry, płytka charakteryzuje się wysokim stosunkiem ceny do jakości, podstawowa wersja kosztuje 99\$. Dla porównania płytka Zybo Z7-20 kosztuje 199\$. Zestawienie cen płytEK zawierających układ Zynq XC7Z010 oraz XC7Z020 znajduje się w Tabeli 3.1.

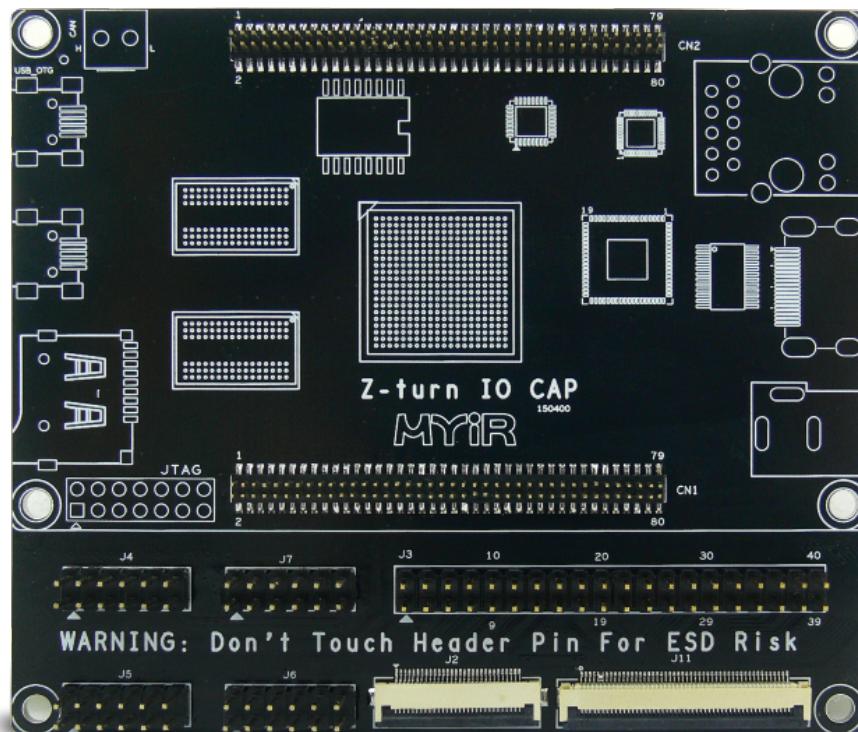
¹ <http://www.myirtech.com/list.asp?id=502>

² <https://store.digilentinc.com/zybo-z7-zynq-7000-arm-fpga-soc-development-board/>

³ <https://store.digilentinc.com/zedboard-zynq-7000-arm-fpga-soc-development-board/>

3.2.1. Interfejsy komunikacyjne

Płytkę Z-turn posiada interfejsy UART oraz Ethernet, które zostały wykorzystane do komunikacji komputera PC z systemem przy użyciu portu szeregowego i protokołu SSH (ang. *Secure Shell*). Istnieje również możliwość podłączenia wyświetlacza bezpośrednio do płytki przy użyciu portu HDMI oraz innych peryferiów przy użyciu portu USB. Dodatkowo producent oferuje płytę rozszerzeniową Z-turn IO-Cape (Rys. 3.3), która zawiera porty do podłączenia kamery przez protokół DVP (ang. *Digital Video Port*) oraz wyświetlacza LCD.



Rysunek 3.3. Płytki rozszerzeniowa Z-turn IO Cape

3.3. Kamera

Aby przetestować działanie systemu w czasie rzeczywistym, wykorzystano zewnętrzny moduł kamery. Przy wyborze sprzętu istotna była ceną modułu, interfejs komunikacji oraz kompatybilność z SBC Z-turn Board. W przypadku wykorzystania kamery w czasie rzeczywistym bardzo ważne jest niskie opóźnienie w wysyłaniu kolejnych ramek obrazu i duża szybkość transferu obrazu.

Producent płyt Z-turn Board oferuje kilka modułów kamer. Wśród nich znajdują się dwa moduły, które zostały przetestowane: MY-CAM002U USB Digital Camera Module (Rys.3.7) oraz MY-CAM011B BUS Camera Module. (Rys. 3.4). W Tabeli 3.2 przedstawiono porównanie testowanych modułów kamer.

⁴ <http://www.myirtech.com/list.asp?id=534>

⁵ <http://www.myirtech.com/list.asp?id=462>

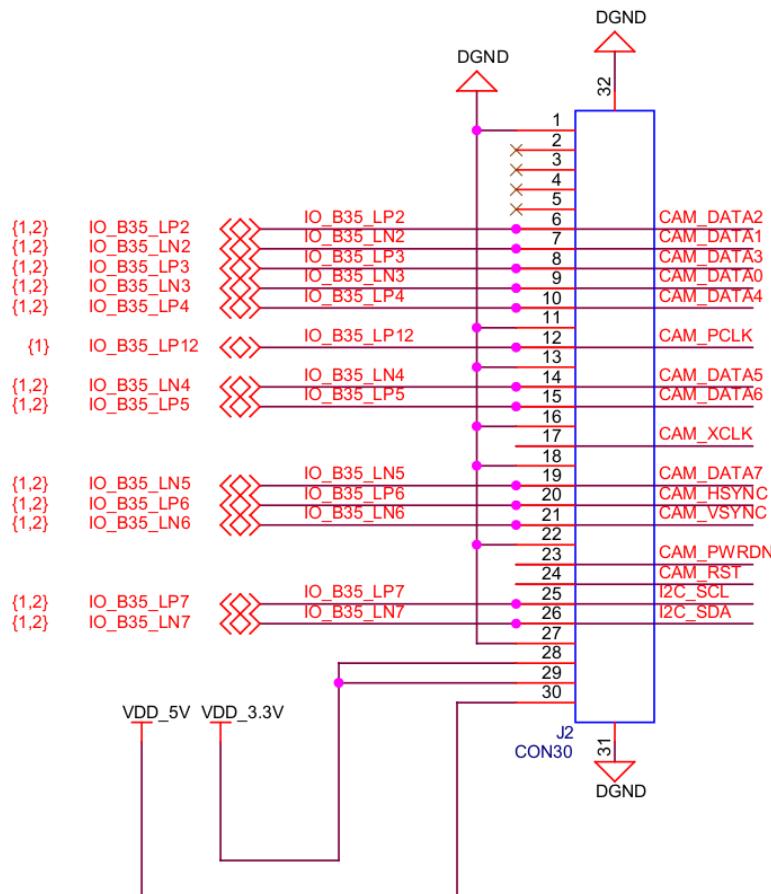
3. Wybór sprzętu

Tabela 3.2. Porównanie testowanych modułów kamer

	MY-CAM011B	MY-CAM002U
Maksymalna rozdzielcość	1600x1200 pikseli	1280x800 pikseli
Pobór mocy w stanie aktywnym	224 mW	110 mW
Format wyjścia	8/10-bit RAW RGB YUV422/YCbCr422 RGB565/555 GRB422	10-bit RAW RGB
Maksymalny transfer obrazu	15 fps (1600x1200) 30 fps (800x600) 30 fps (1280x720) 24 fps (1366x768)	30 fps (1280x800) 60 fps (640x480) 30 fps (1280x720)
Cena modułu	25\$ ⁴	19\$ ⁵



dołączony do żadnego portu płytki Z-Turn Board. Piny CAM_PWRDN i CAM_RST również nie są dołączone do złącza J2 od strony płytki Z-turn Board.

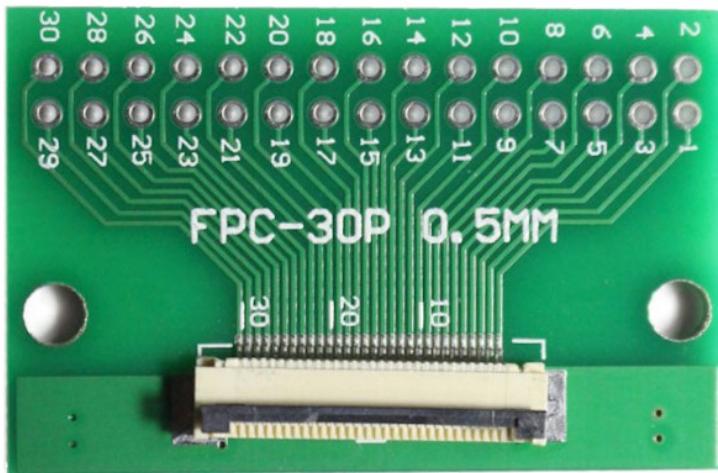


Rysunek 3.5. Schemat portu DVP na płytce rozszerzeniowej IO-Cape

Dołączenie samego sygnału zegara do konektora FPC byłoby problematyczne, więc zdecydowano się na zastosowanie adaptera⁶ złącza FPC/FFC o rastrze 0,5 mm na otwory DIP raster 2,54 mm. Dzięki temu możliwe było podłączenie modułu kamery przy użyciu przewodów do złącza J3 płytki rozszerzeniowej IO-Cape [9].

Po podłączeniu kamery, zainstalowaniu odpowiednich sterowników i *device-tree* oraz przeskanowaniu magistrali I^2C przy użyciu narzędzia *i2c-tools*, moduł kamery zwraca poprawnie swój adres. Jednak sterownik sensora kamery *ov2659.c* nie rozpoznaje podłączonego urządzenia.

⁶ <https://kamami.pl/zlaczka-ffc-fpc-zif/579387-adapter-zlaczka-fpcffc-05mm-30-pin-na-dip.html>



Rysunek 3.6. Adapter złącza FPC/FFC o rastrze 0,5 mm na otwory DIP

3.3.2. Moduł MY-CAM002B

Z uwagi na ograniczony czas projektu zdecydowano się na użycie modułu kamery MY-CAM002U USB Digital Camera Module (Rys.3.7). Moduł umożliwia rejestrowanie obrazu w 30 klatkach na sekundę przy rozdzielcości 1280x640 pikseli i jest tańszy od modułu MY-CAM011B. Główną zaletą tego modułu jest łatwość podłączenia sprzętu zarówno do płytki Z-turn Board, jak i do komputera PC, co w znacznym stopniu przyspieszyło debugowanie.

Kamera po podłączeniu do komputera PC działała poprawnie, bez wprowadzania jakichkolwiek zmian w systemie, zarówno w dystrybucji Ubuntu, jak i w systemie Windows. Po dodaniu odpowiednich sterowników do jądra i uruchomieniu systemu Petalinux na płytce Z-turn Board, w systemie pojawiły się pliki urządzenia pod nazwą `/dev/video0` oraz `/dev/video1`, co umożliwiło odebranie obrazu przy pomocy interfejsu V4L2 (ang. *Video for Linux 2*) oraz OpenCV.



Rysunek 3.7. Moduł kamery MY-CAM002U USB Digital Camera Module

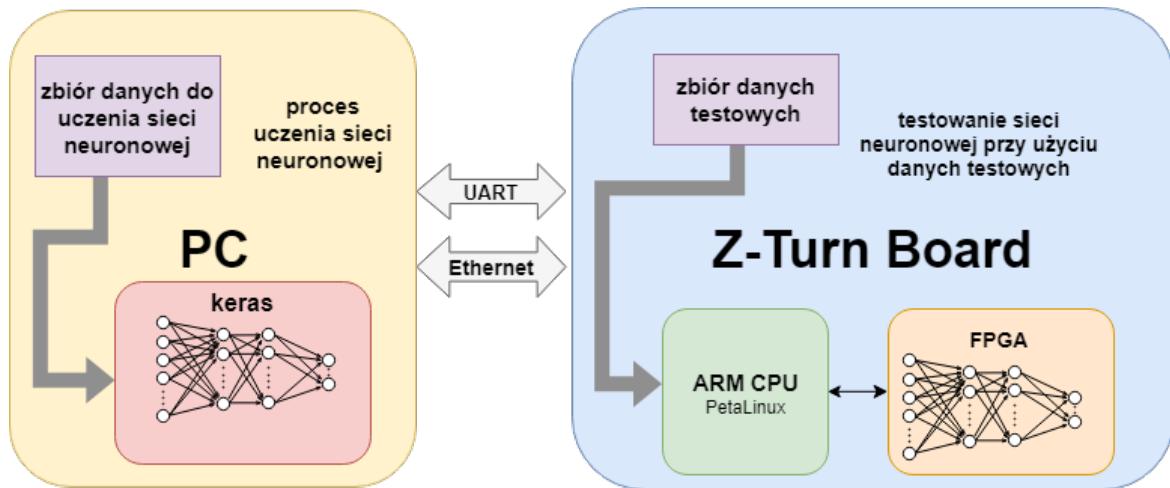
4. Implementacja

4.1. Budowa systemu

4.1.1. Schemat blokowy systemu

System składa się z dwóch głównych części (Rys. 4.1):

- aplikacja wykorzystująca pakiet keras, uruchamiana na komputerze PC
- część uruchamiana na płytce Z-Turn Board



Rysunek 4.1. Schemat blokowy systemu

4.2. Wybór narzędzi

Użycie w pracy płytki z układem Zynq determinuje użycie narzędzi wspieranych przez firmę Xilinx. Zdecydowano się na użycie najnowszej (w momencie rozpoczęcia projektu) wersji oprogramowania 2019.2. Producent zaleca[10] instalację programu Vivado na jednym ze wspieranych systemów operacyjnych:

- Microsoft Windows 7 SP1 Professional (64-bit), English/Japanese
- Microsoft Windows 10.0 1809 Update; 10.0 1903 Update (64-bit), English/Japanese
- Red Hat Enterprise Workstation/Server 7.4, 7.5, and 7.6 (64-bit)
- SUSE Linux Enterprise 12.4 (64-bit)
- CentOS 7.4, 7.5, and 7.6 (64-bit)
- Ubuntu Linux 16.04.5 LTS; 16.04.6 LTS; 18.04.1 LTS; 18.04.02 LTS (64-bit)
- Amazon Linux 2 LTS (64-bit).

W projekcie wykorzystywane było również narzędzie Petalinux, które wymaga zainstalowania na maszynie z systemem operacyjnym Linux. Zgodnie z dokumentacją[11] jest to jedna z trzech dystrybucji:

- Red Hat Enterprise Workstation/Server 7.4, 7.5, 7.6 (64-bit)

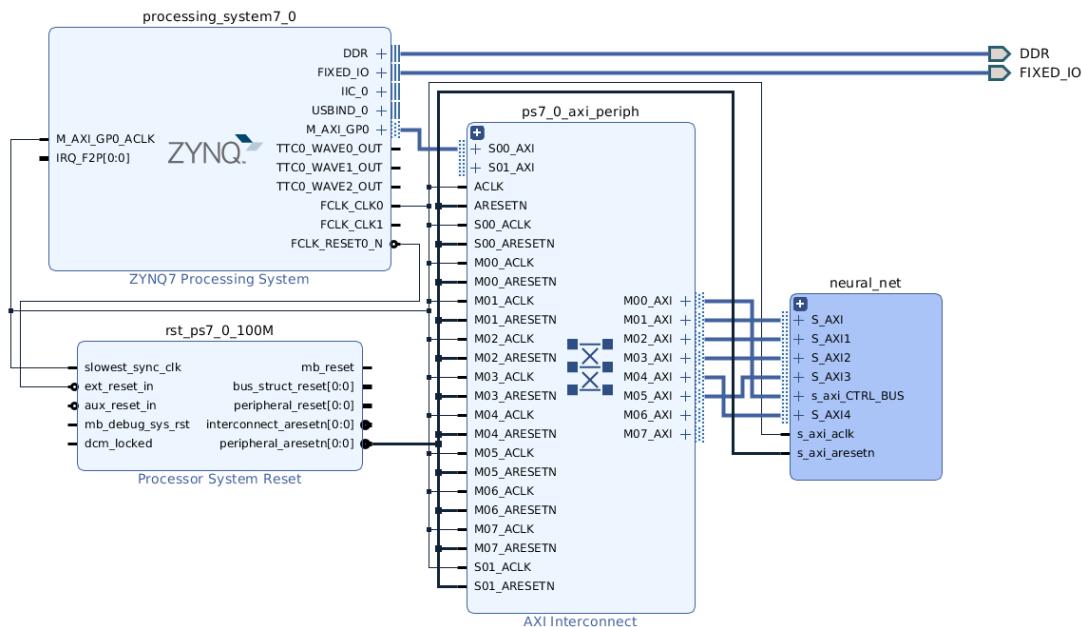
4. Implementacja

- CentOS Workstation/Server 7.4, 7.5, 7.6 (64-bit)
- Ubuntu Linux Workstation/Server 16.04.5, 16.04.6, 18.04.1, 18.04.02 (64-bit)

Aby zapewnić poprawne działanie narzędzi oraz z racji na sporą popularność i duże wsparcie społeczności wybrano dystrybucję Ubuntu 18.04.02 LTS. Przy instalacji Petalinuxa warto również zwrócić uwagę, że zalecane jest aż 100 GB wolnego miejsca na dysku twardym.

4.3. Projekt systemu w środowisku Vivado

Zdecydowano się na zastosowanie bloków pamięci BRAM do komunikacji pomiędzy blokiem IP implementowanym z użyciem HLS a procesorem. Zrzut ekranu przedstawiający schemat systemu w środowisku Vivado umieszczono na Rys. 4.2. Znajduje się na nim m.in. blok IP *ZYNQ7 Processing System*, przedstawiający procesor, oraz blok *AXI Interconnect*, umożliwiający podłączenie peryferiów do procesora. Po podwójnym kliknięciu na blok procesora pojawia się okno, w którym można ustawić odpowiednią konfigurację układu.

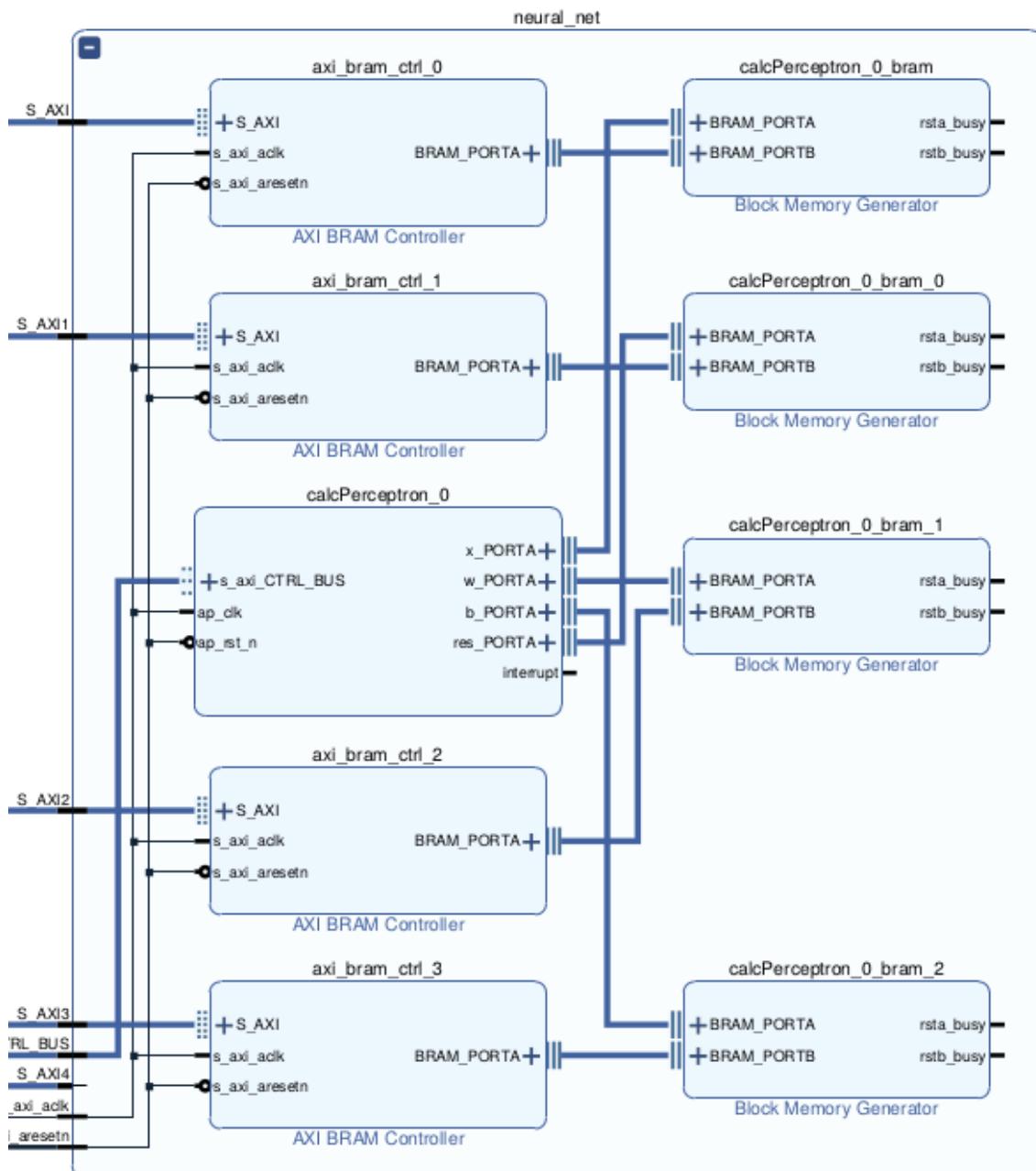


Rysunek 4.2. Schemat systemu przedstawiony w narzędziu Vivado

4.3.1. Pamięć Block RAM

Zdecydowano się na zastosowanie bloków pamięci BRAM do komunikacji pomiędzy blokiem IP implementowanym z użyciem HLS a procesorem. Zaletą pamięci BRAM jest szybkość działania jednak ilość pamięci, jaką można wykorzystać, jest mocno ograniczona.

Dzięki temu możliwy jest dostęp z poziomu sterownika w systemie PetaLinux do danych przetworzonych przez blok HLS. Pamięć alokowana jest przy użyciu bloku *Block Memory Generator* i podłączana do procesora dzięki blokom *AXI BRAM Controller*(Rys. 4.3) oraz *AXI Interconnect* (Rys. 4.2).

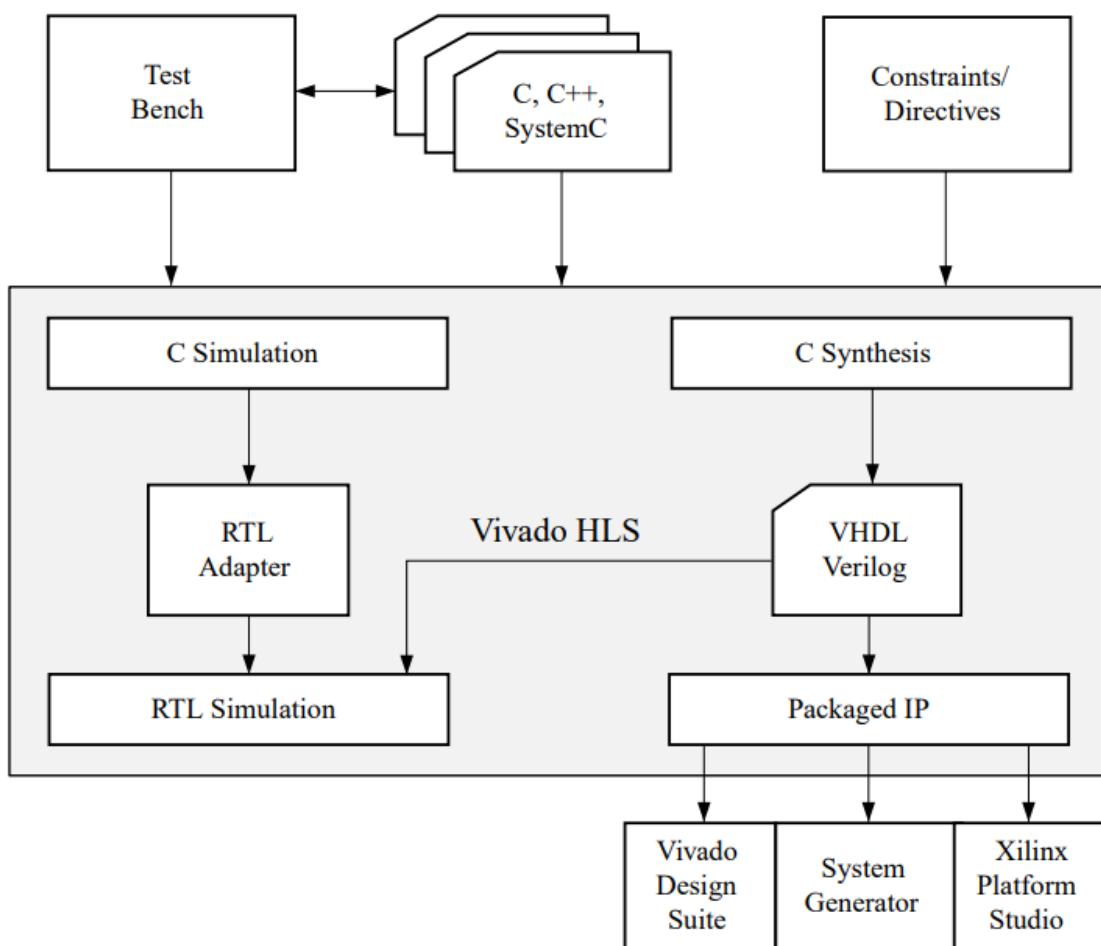


Rysunek 4.3. Szczegółowy schemat części systemu *neural_net*

4.4. Wykorzystanie metody HLS

Przy użyciu metody HLS możliwe jest stworzenie własnego bloku IP (ang. Intellectual Property), który następnie jest umieszczany w katalogu IP i można go wielokrotnie wykorzystać w projekcie RTL (ang. Register Transfer Level). Do projektu z użyciem HLS (Rys.4.4) potrzebny jest plik z algorytmem w języku C/C++ lub System C, plik testowy napisany w języku C (ang. *test bench*) oraz plik z opisem ograniczeń sprzętowych (ang. constraints). Kolejne etapy projektu z wykorzystaniem metody HLS [12]:

1. Kompilacja, wykonanie (symulacja) i debugowanie algorytmu napisanego w języku C
2. Synteza algorytmu w języku C w implementacji RTL
3. Wygenerowanie raportu i analiza projektu
4. Zweryfikowanie implementacji RTL
5. Spakowanie implementacji RTL w blok IP

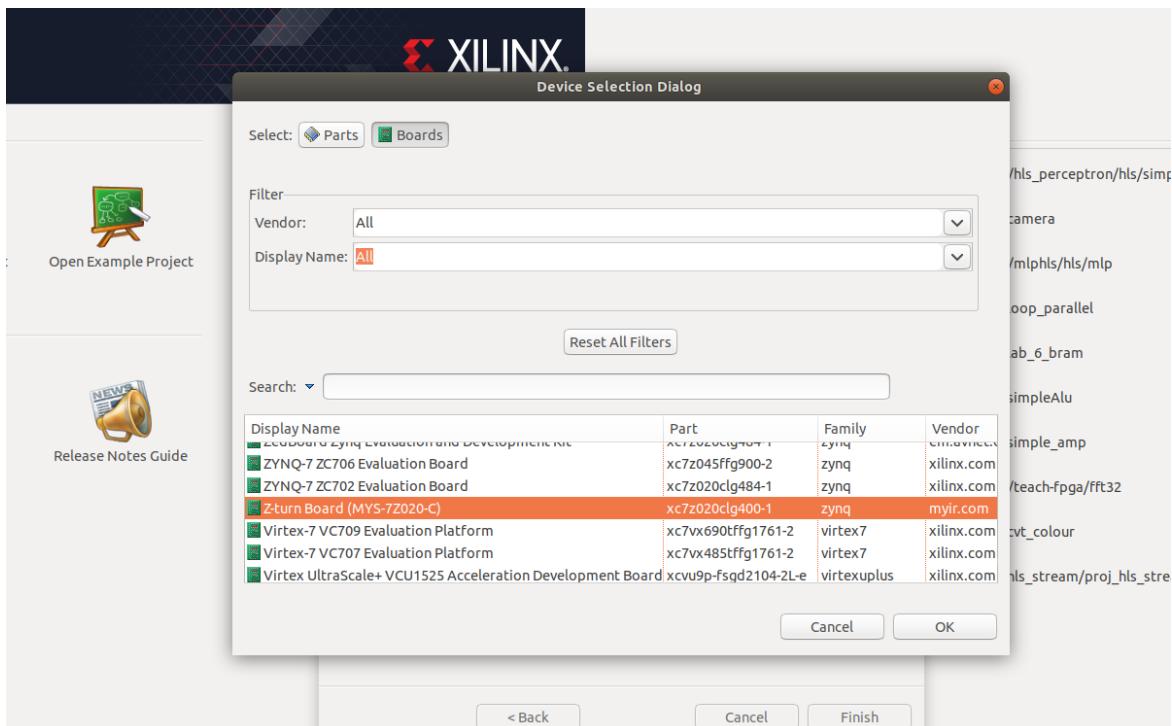


Rysunek 4.4. Proces projektowania przy użyciu metody HLS

Zastosowanie syntezy wysokiego poziomu umożliwia przeniesienie algorytmu napisanego w języku C/C++ lub System C na implementację w układzie FPGA. Dodatkową

zaletą metody HLS jest dostępność bibliotek do przetwarzania obrazów oraz ułatwiających implementację operacji matematycznych.

Przy tworzeniu nowego projektu w Vivado HLS (Rys. 4.5) trzeba podać urządzenie, na którym uruchamiany będzie blok IP. Jeśli płytka nie jest widoczna na liście, należy dodać pliki opisujące ją w odpowiednim katalogu, gdzie zostało zainstalowane narzędzie Vivado HLS.



Rysunek 4.5. Wybór płytki podczas tworzenia projektu w Vivado HLS

4.5. Zbiór danych wejściowych

W procesie uczenia oraz testowania poprawności działania modelu sztucznej sieci neuronowej wykorzystano zbiór odręcznie pisanych cyfr MNIST (ang. THE MNIST DATABASE of handwritten digits)[13]. Jest to baza 60000 obrazów do przeznaczonych do uczenia sieci oraz 10000 do walidacji. Każdy obraz przedstawia jedną cyfrę w formacie 28x28 pikseli. Fragment zbioru MNIST przedstawiono na Rys. 4.6.

4.6. Opracowanie modelu ANN

Przy projektowaniu algorytmu ANN bardzo ważnym aspektem jest odpowiednie dopasowanie modelu do danych wejściowych. Najczęściej odbywa się to poprzez wielokrotne testowanie systemu dla różnych parametrów sieci. W tej pracy początkowym wyborem była architektura MLP. Dane wejściowe w postaci obrazów 28x28 pikseli są konwertowane na tablicę 784 wartości typu *float*. Fragment kodu przedstawia Listing 1. Model zawiera następujące warstwy:



Rysunek 4.6. Fragment zbioru odręcznie pisanych cyfr MNIST

- warstwę wejściową (784 wejścia)
- warstwę ukrytą (16 neuronów)
- warstwę wyjściową (10 neuronów).

Listing 1. Implementacja modelu ANN MLP z jedną warstwą ukrytą

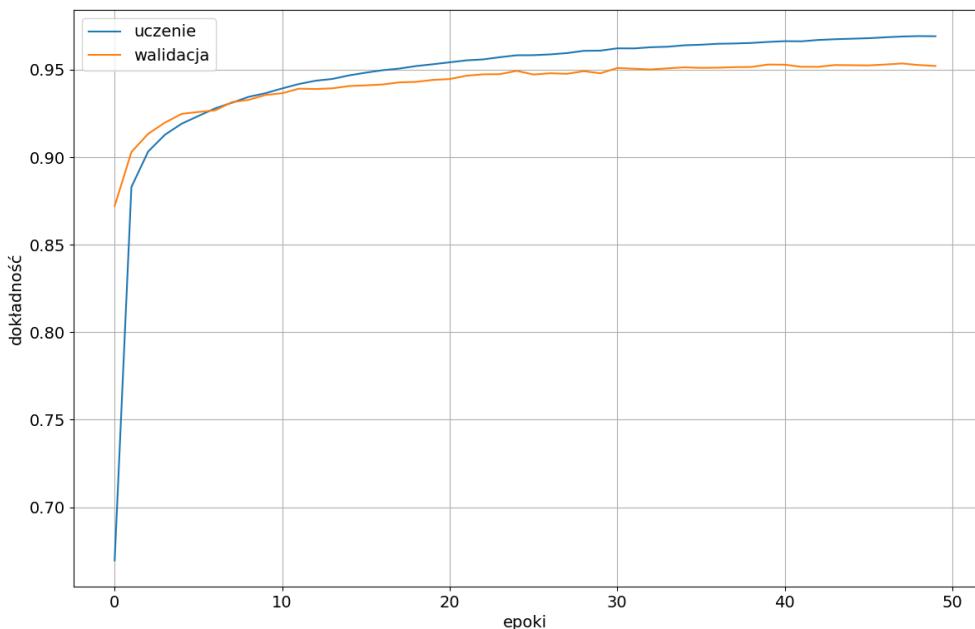
```
55 model = Sequential()
56 model.add(Flatten())
57 model.add(Dense(16, use_bias=True, activation='sigmoid'))
58 model.add(Dense(num_classes, use_bias=True, activation='sigmoid'))
```

4.6.1. Uczenie Sztucznej Sieci Neuronowej

Podczas uczenia i testowania modelu użyto zbioru MNIST, który zainportowano, wykorzystując funkcję z pakietu keras. Dokonano uczenia sieci neuronowej przy użyciu zbioru 60000 obrazów i testowania modelu, podając na wejście sieci 10000 obrazów. Osiągnięto dokładność na poziomie 94,97%. Na Rys. 4.7 przedstawiono, jak zmieniała się dokładność (ang.*accuracy*) w kolejnych epokach dla zbioru uczącego i walidacyjnego.

4.6.2. Implementacja modelu przy użyciu narzędzia Vivado HLS

Korzystając z narzędzia Vivado HLS, napisano program w języku C++ implementujący zaprojektowany wcześniej model sieci. W procesie uczenia ustalono wartości wag i biasów. Dwa główne pliki projektu w narzędziu Vivado HLS to core.cpp, zawierający implemen-



Rysunek 4.7. Wykres zmian dokładności w kolejnych epokach – ANN z jedną warstwą ukrytą

tację algorytmu ANN oraz `test_core.cpp`, który służy do przetestowania algorytmu po przeprowadzeniu syntezy (ang. test bench). Pliki zawierające wartości wejściowe, a także wagi i biasy są importowane do programu `test_core.cpp`.

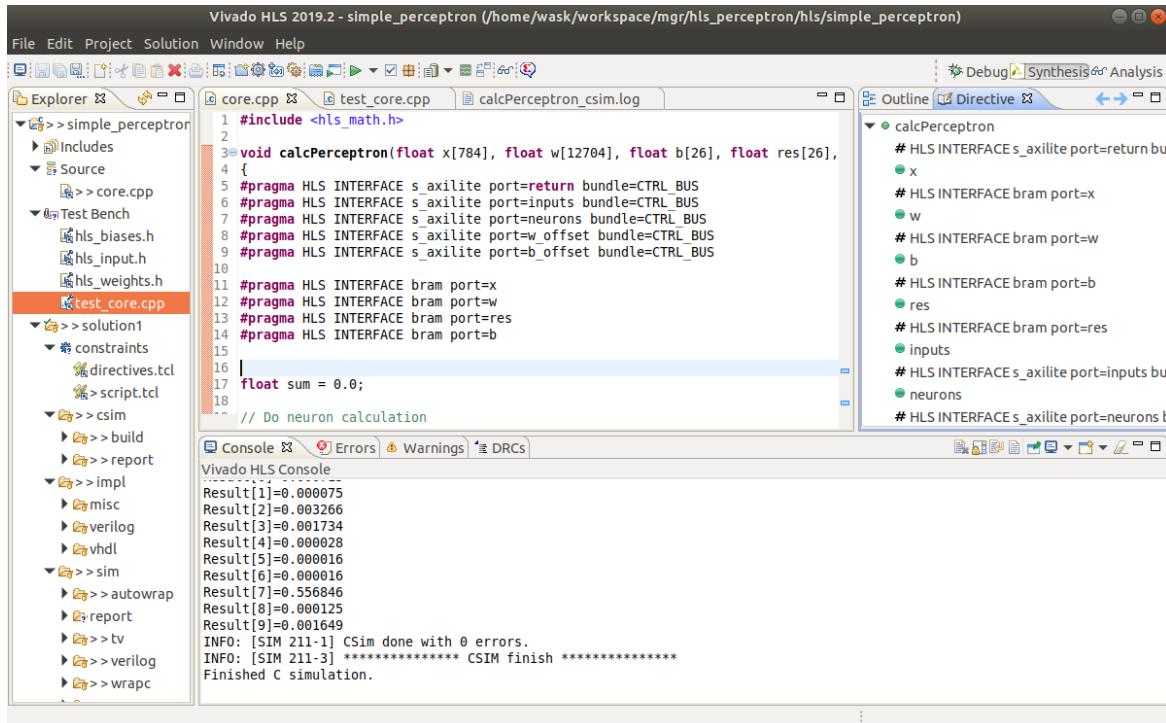
Pierwszym etapem jest Symulacja C, która jest wstępnią weryfikacją poprawności algorytmu. Do przeprowadzenia symulacji wykorzystano dane wyeksportowane przy użyciu skryptu `keras2fpga.py` i zapisane w plikach `hls_biases.h`, `hls_weights.h` oraz `hls_input.h`. Wynikiem symulacji jest plik `.log`, w którym można znaleźć informacje o tym, jak przebiegało wywołanie testowanych funkcji. 4.8 Symulację w języku C wykonuje się dużo szybciej niż późniejszą symulację RTL, więc stosuje się ją jako pierwszy etap przed podjęciem dalszych kroków, które zajmują więcej czasu.

Następnie wykonywana jest synteza oraz kosymulacja, umożliwiająca weryfikację poprawności syntezy. Ponadto narzędzie generuje raport, który przedstawia informacje na temat zużycia zasobów i opóźnień czasowych. Po prawidłowym przeprowadzeniu kosymulacji należy użyć opcji *Export RTL*, co umożliwia dodanie nowego bloku IP do projektu w narzędziu Vivado.

4.7. Synteza projektu w narzędziu Vivado

Aby przetestować działanie nowego bloku IP w narzędziu Vivado HLS, należy dodać i odpowiednio podłączyć blok do schematu blokowego (ang. *Block Design*) w narzędziu Vivado. Następnie należy sprawdzić, czy urządzenia zostały właściwie zaadresowane w

4. Implementacja



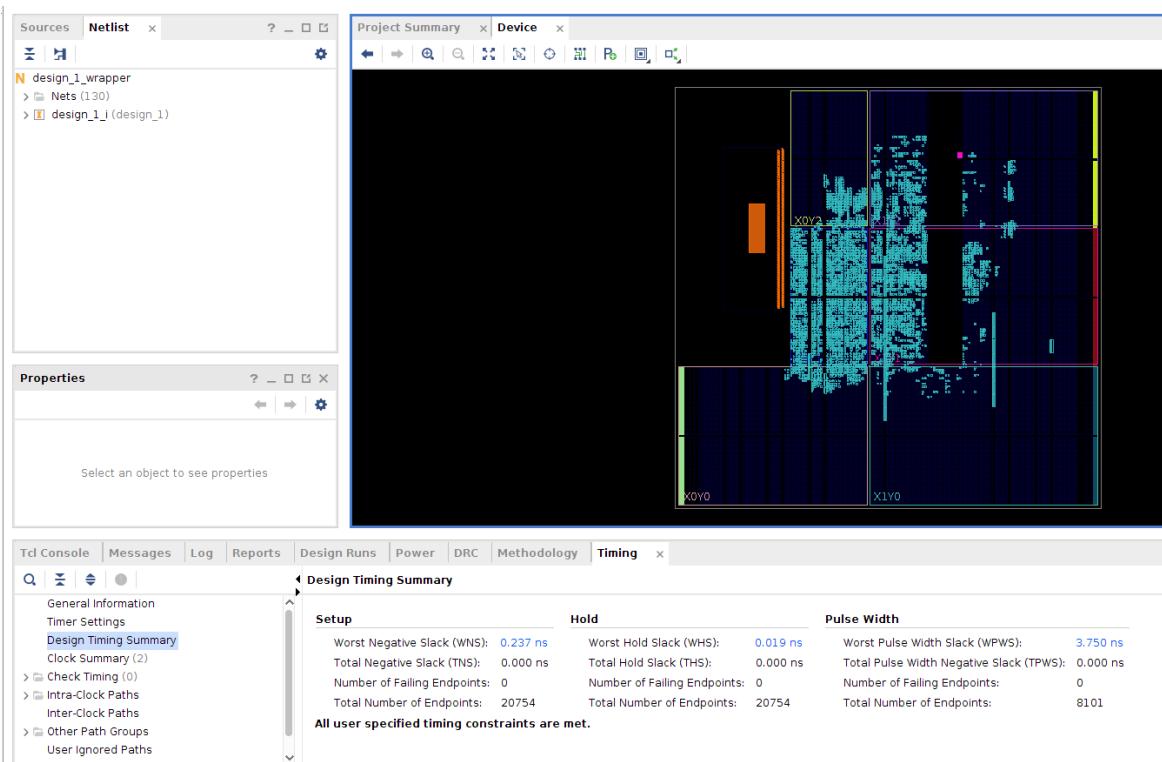
Rysunek 4.8. Wynik poprawnie przeprowadzonej symulacji w Vivado HLS

zakładce *Address Editor* i wprowadzić ewentualne zmiany. Gotowy projekt można poddać automatycznej weryfikacji za pomocą funkcji *Validate Design* i uruchomić syntezę i implementację. Po poprawnie przeprowadzonej implementacji narzędzie umożliwia otwarcie realizacji sprzętowej projektu (opcja *Open Implemented Design*). Zakładka *Timing* (Rys. 4.9) umożliwia sprawdzenie, czy w zaimplementowanym projekcie zostały spełnione wymagania czasowe, a w oknie *Device* można zweryfikować jakie zasoby zostały użyte w implementacji.

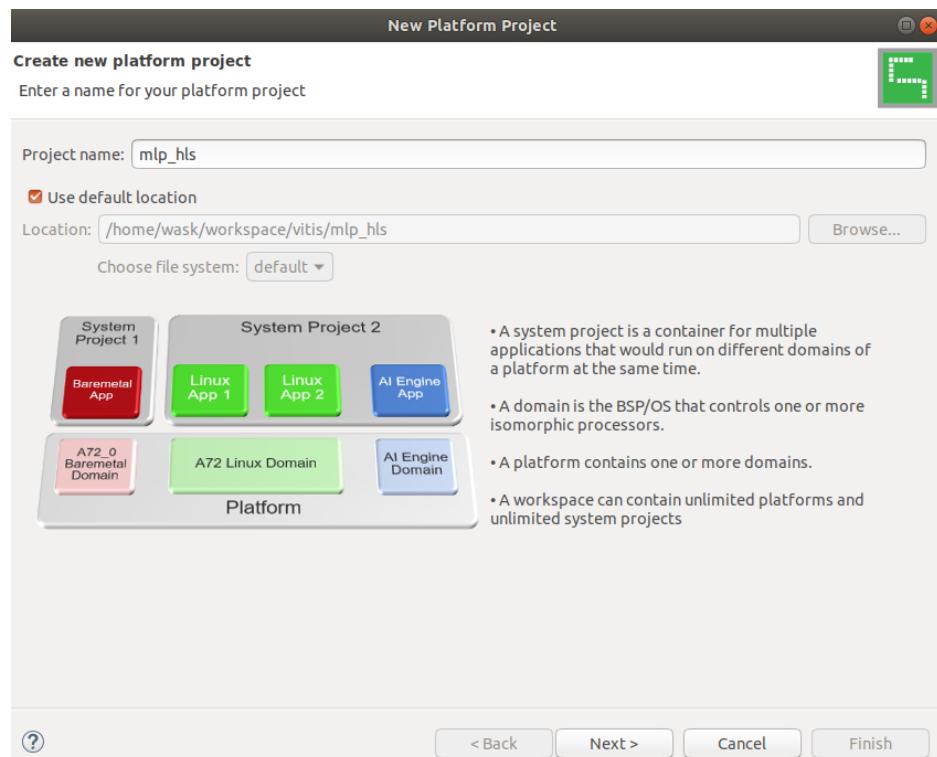
Następnym krokiem jest wygenerowanie *Bitstreamu* i eksportu sprzętu w postaci pliku z rozszerzeniem *.xsa*. Dzięki temu projekt sprzętu stworzony w programie Vivado można użyć w narzędziu *Vitis* lub *Petalinux*. Aplikacja uruchamiana w trybie *standalone* bez systemu operacyjnego w programie Vitis umożliwia szybką weryfikację poprawności działania zaprojektowanego systemu. Dlatego przed przejściem do implementacji w systemie Petalinux przystąpiono do stworzenia projektu oprogramowania w programie Vitis.

4.8. Implementacja przy użyciu narzędzia Vitis

Po wyeksportowaniu projektu sprzętu w narzędziu Vivado można uruchomić program Vitis. Narzędzie umożliwia stworzenie nowego projektu platformy, dzięki opcji *New Platform Project* (Rys. 4.10). Przy tworzeniu projektu należy wybrać odpowiedni plik *.xsa* (Rys. 4.11), wyeksportowany wcześniej z narzędzia Vivado.

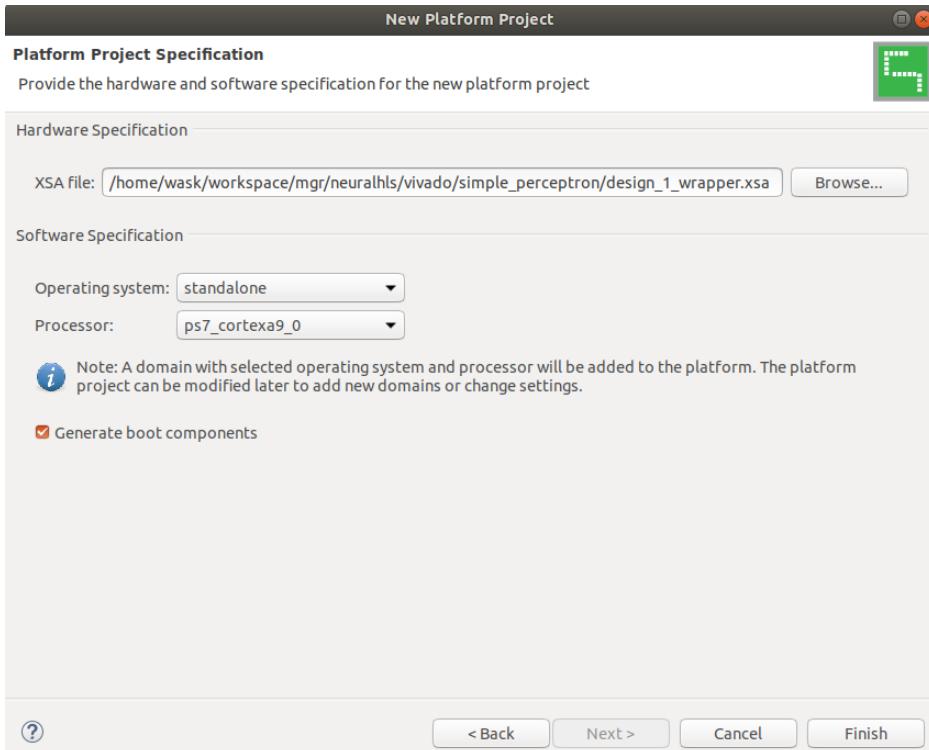


Rysunek 4.9. Wynik poprawnie przeprowadzonej implementacji w narzędziu Vivado



Rysunek 4.10. Tworzenie nowego projektu w programie Vitis

4. Implementacja



Rysunek 4.11. Wybór pliku .xsa z opisem konfiguracji sprzętu w programie Vitis

Następnie należy stworzyć projekt aplikacji, za pomocą opcji *New Application Project* i wybraniu odpowiedniego, stworzonego wcześniej projektu platformy. Narzędzie Vitis zawiera szereg aplikacji przykładowych, z których warto skorzystać przy uruchamianiu systemu po raz pierwszy. Przed zbudowaniem projektu, aby zapewnić komunikację płytki Z-turn Board z komputerem za pomocą konwertera USB-UART znajdującego się na płytce, należy zmienić następujące ustawienia BSP (ang. *Board Support Package*) dla domeny *standalone*:

- stdin z ps_7uart_0 na ps_7uart_1
- stdout z ps_7uart_0 na ps_7uart_1

Po zbudowaniu projektu platformy można przejść do pisania kodu aplikacji. Warto zaznaczyć, że do uruchomienia i debugowania aplikacji na płytce Z-turn Board, potrzebny jest programator JTAG *Xilinx Platform Cable*. Aby obejrzeć wyjście standardowe programu, należy otworzyć okno *Vitis Serial Terminal* i wybrać odpowiedni port USB. Trzeba również pamiętać o wybraniu odpowiedniej opcji uruchamiania płytki, poprzez ustawienie zwojki na JP1 zwarte i JP2 rozwarste.

Jak widać na Rys. 4.12, algorytm działa poprawnie. Po uruchomieniu i sprawdzeniu poprawności obliczeń w programie Vitis można przystąpić do konfigurowania systemu operacyjnego przy użyciu narzędzia Petalinux.

The screenshot shows the Vitis IDE interface. On the left, the code editor displays the `main.cpp` file, which contains C++ code for a neural network application. On the right, the `Vitis Serial Terminal` window shows the execution results. The terminal output includes the connection information, the neural network test details, and the resulting values for each digit in the sample.

```

86     XCalcperceptron_Set_b_offset(&calcPerceptron, 16);
87 }
88
89 int main() {
90 // first layer -> set offset to 0
91     printf("Neural Network MNIST test for 10 digits\n");
92     init_PerceptronCore();
93
94     for (int j=0; j<10; j++) {
95         printf("load initial data for sample %d\n", j);
96         init_load_data(j); //set sample 0-9
97
98         XCalcperceptron_Start(&calcPerceptron);
99         while(!XCalcperceptron_IsDone(&calcPerceptron));
100
101        load_2_layer_data();
102
103        XCalcperceptron_Start(&calcPerceptron);
104        while(!XCalcperceptron_IsDone(&calcPerceptron));
105
106        for (int i = 0; i < 10; i++) {
107            printf("result[%d]: %f\n", i, resHW[i]);
108        }
109    }
110    printf("End of test\n");
111
112    return 0;
113 }
114
115
116
117
118 }
119

```

Vitis Serial Terminal Output:

```

Connected to /dev/ttyUSB2 at 115200
Neural Network MNIST test for 10 digits
load initial data for sample 0
result[0]: 0.000006
result[1]: 0.000000
result[2]: 0.000011
result[3]: 0.000088
result[4]: 0.000001
result[5]: 0.000002
result[6]: 0.000000
result[7]: 0.112744
result[8]: 0.000001
result[9]: 0.000082
load initial data for sample 1
result[0]: 0.000458
result[1]: 0.000498
result[2]: 0.024287
result[3]: 0.000338
result[4]: 0.000000
result[5]: 0.000055
result[6]: 0.000153
result[7]: 0.000001
result[8]: 0.000001
result[9]: 0.000000
load initial data for sample 2
result[0]: 0.000000
result[1]: 0.023842
result[2]: 0.000084
result[3]: 0.000005

```

Rysunek 4.12. Wynik uruchomienia aplikacji w programie Vitis

4.9. Test z wykorzystaniem systemu operacyjnego Petalinux

PetaLinux Software Development Kit (SDK) jest narzędziem zawierającym wszystkie niezbędne elementy do budowania, rozwijania, testowania i wdrażania systemów wbudowanych opartych na systemie Linux. PetaLinux jest przeznaczony głównie do systemów, opartych o układy FPGA i składa się z trzech najważniejszych elementów:

- prekonfigurowany obraz binarny systemu
- system Linux konfigurowalny do zastosowania na sprzęcie firmy Xilinx
- PetaLinux SDK zawierający narzędzia służące do projektowania i wdrażania systemu

4.9.1. Wstępna konfiguracja systemu Petalinux

Zgodnie z zaleceniami producenta, projekcie wykorzystano wersję narzędzia Petalinux kompatybilną z wersją Vivado – 2019.2. Każdorazowo, przed użyciem Petalinuxa należy pamiętać o załadowaniu ustawień przy pomocy komendy:

```
source <katalog-instalacyjny>/petalinux/2019.2/settings.sh
```

Narzędzie zużywa dużo przestrzeni dyskowej, jeśli wolnego miejsca będzie zbyt mało, zostanie wyświetlony odpowiedni komunikat. Następnie można przejść do tworzenia nowego projektu z ustawieniami domyślnymi dla płyt z układami *Zynq*:

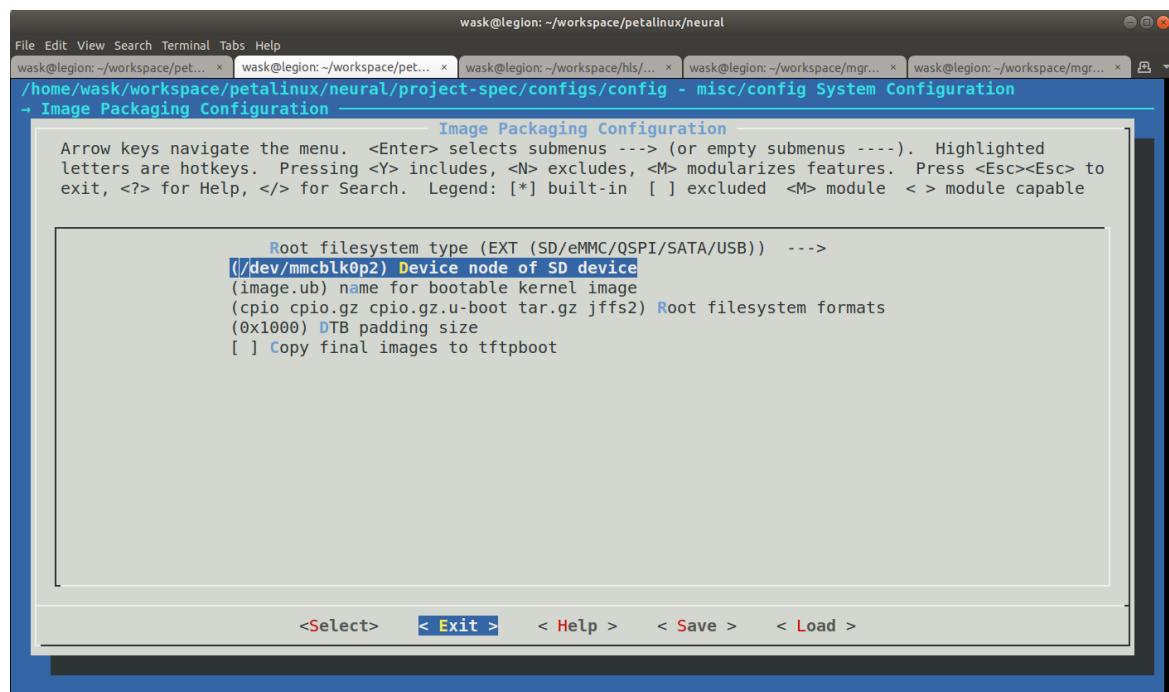
```
petalinux-create -t project -n <nazwa_projektu> --template_zynq
```

4. Implementacja

Projekt sprzętu w programie Vivado został wyeksportowany do pliku z rozszerzeniem *.xsa*. Aby zimportować plik opisujący konfigurację sprzętu, utworzonego w programie Vivado stosujemy następującą komendę:

```
petalinux-config --get-hw-description <katalog_projektu_vivado>
```

Po wywołaniu komendy *petalinux-config* pojawia się interfejs graficzny narzędzia Petalinux, który umożliwia zmianę ogólnych ustawień projektu (Rys. 4.13). Przechodząc do zakładki *Image Packaging Configuration* (Rys. 4.13), można ustawić lokalizację zewnętrznego systemu plików oraz wybrać partycję karty sd, na której będzie się znajdował. Stosując opcję *-c* komendy *petalinux-config* można dostosować ustawienia jądra Linuxa (argument *kernel*) oraz systemu plików (argument *rootfs*).

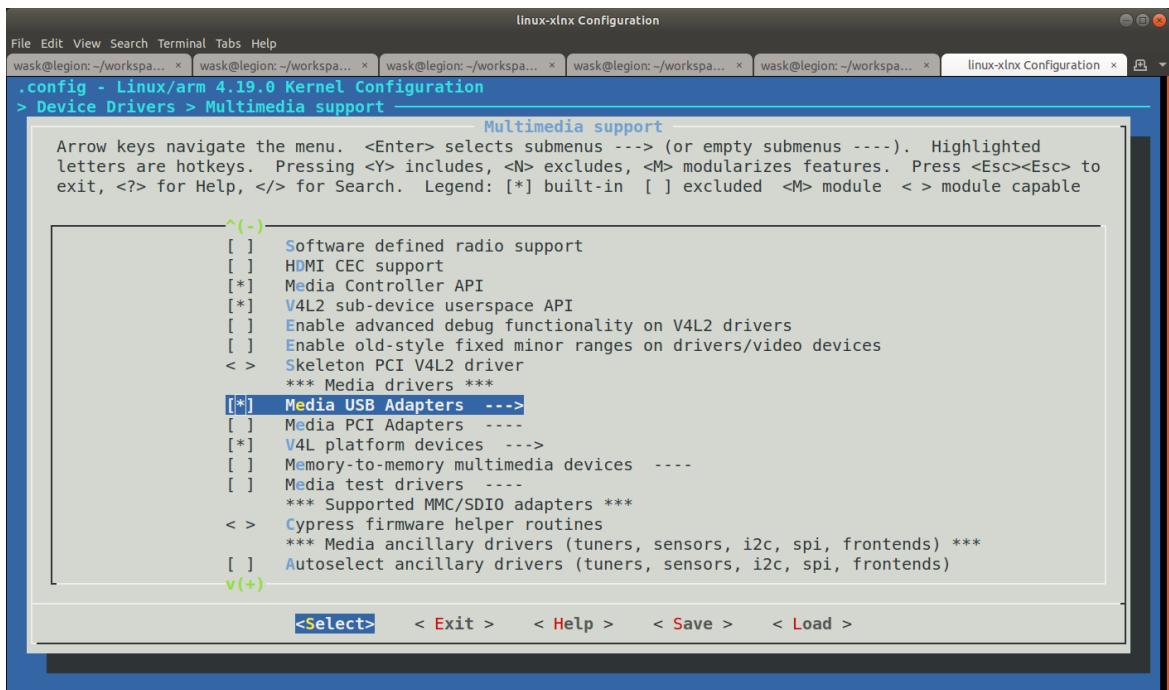


Rysunek 4.13. Wstępna konfiguracja przy użyciu funkcji *petalinux-config*

4.9.2. Konfiguracja jądra systemu Petalinux

Po wywołaniu komendy *petalinux-config -c kernel* w terminalu pojawia się menu konfiguracyjne jądra systemu Petalinux. W oknie jest wiele opcji, jednak najwięcej zmian wprowadzono w zakładce *Device Drivers*.

W projekcie podjęto decyzję o zastosowaniu kamery podłączonej przy użyciu portu USB. Aby umożliwić aplikacji korzystającej z biblioteki OpenCV dostęp do urządzenia, potrzebne były odpowiednie sterowniki [14] dostępne w zakładce *Multimedia Support*. Zastosowany w projekcie moduł kamery MY-CAM002U jest kompatybilny ze sterownikiem UVC (ang. *USB Video Class*). W konfiguracji jądra dodano również interfejs V4L2, który daje dostęp do modułu kamery z poziomu aplikacji z przestrzeni użytkownika.



Rysunek 4.14. Konfiguracja jądra PetaLinuxa

Aby uzyskać dostęp z poziomu systemu Petalinux do bloku IP zaimplementowanego w narzędziu Vivado HLS, należy zainstalować odpowiednie sterowniki urządzeń lub utworzyć własne. Dla wielu typów urządzeń tworzenie nowego sterownika nie jest konieczne. Alternatywnym rozwiązaniem jest zastosowanie sterownika UIO (ang. *Userspace Input Output Driver*) [15], który umożliwia dostęp do urządzenia w aplikacji z przestrzeni użytkownika. Znacznie upraszcza to proces tworzenia oprogramowania do obsługi urządzenia i zmniejsza ryzyko powstawania trudnych w debugowaniu i często niebezpiecznych dla działania systemu operacyjnego błędów. Jednak należy pamiętać, że sterowniki UIO nie są przeznaczone dla dowolnego typu sprzętu. Stosuje się je do obsługi urządzeń, generujących przerwania, posiadających pamięć, którą można zmapować i sterować urządzeniem poprzez pisanie do tej pamięci.

4.9.3. Konfiguracja systemu plików

Narzędzie Petalinux umożliwia również dostosowanie systemu plików do potrzeb użytkownika. W menu konfiguracyjnym jest wiele pakietów i bibliotek do różnych zastosowań. W zakładce *apps* użytkownik ma dostęp do aplikacji, które stworzył w obrębie danego projektu. Narzędzie Petalinux umożliwia utworzenie nowej aplikacji przy użyciu komendy:

```
petalinux-create -t apps -n <nazwa_aplikacji>
```

W projekcie zdecydowano się na użycie skryptów napisanych w języku Python i biblio-

4. Implementacja

teki OpenCV do rejestrowania obrazu i wysyłania go poprzez port Ethernet do komputera PC. W tym celu zaznaczono następujące opcje w konfiguracji systemu plików:

- python i python-math
- python-numpy
- packagegroup-petalinux-v4lutils
- packagegroup-petalinux-opencv

Kolejnym etapem w procesie konfigurowania systemu Petalinux jest dostosowanie drzewa urządzeń (ang. *device-tree*). Drzewo urządzeń można edytować za pomocą plików system-user.dtsi oraz pl-custom.dtsi dostępnych w katalogu:

```
<katalog_projektu>/project-spec/meta-user/recipes-bsp/device-tree
```

W przypadku wykorzystania sterowników UIO należy wpisać w polu *compatible* każdego z węzłów urządzeń (pamięci BRAM) wartość "*generic-uio*" [16]. Dostęp do urządzeń jest zapewniony dzięki plikom /dev/uioX, gdzie X to numer urządzenia (zaczynając od 0 dla pierwszego urządzenia).

4.9.4. Przygotowanie obrazu systemu

Po dostosowaniu pliku *device-tree* można uruchomić komplikację petalinuxa komendą *petalinux-build*. Gdy system zostanie zbudowany, należy spakować wszystkie potrzebne pliki przy użyciu funkcji *petalinux-package*. Wynikiem tej operacji są dwa pliki dostępne w katalogu images/linux BOOT.bin i image.ub, które należy przekopiować na odpowiednią partycję karty sd. Jeśli w konfiguracji Petalinuxa wyłączona została opcja wspierania Initial RAM filesystem, należy dodatkowo wypakować system plików na drugiej partycji na karcie sd. Tak przygotowaną kartę sd można umieścić w slocie na płytce Z-turn, podłączyć zasilanie i uruchomić system. Istnieje opcja komplikacji jedynie wybranej aplikacji stworzonej przez użytkownika przy pomocy komendy *petalinux-build -c <nazwa_aplikacji>*. W przypadku wprowadzenia zmian w systemie, przed ponownym wypakowaniem systemu plików, należy sformatować partycję zawierającą system plików.

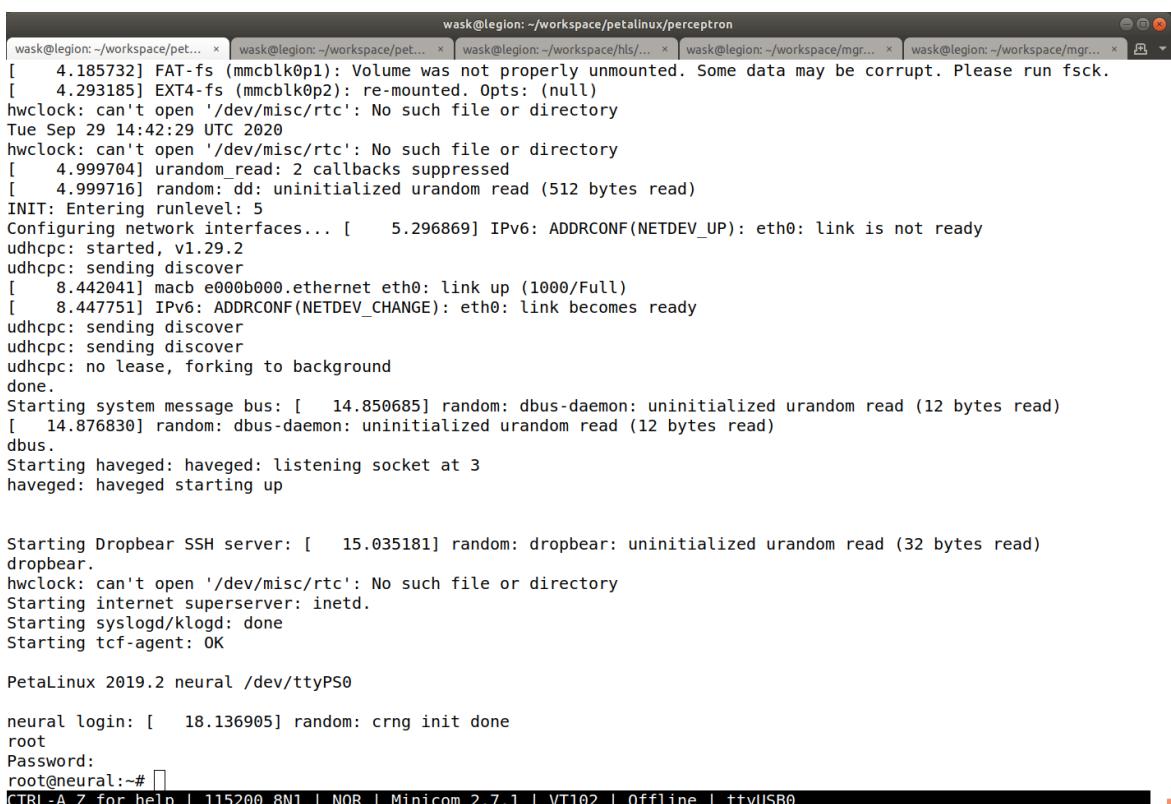
4.9.5. Uruchomienie systemu Petalinux

Komunikacja płytki Z-turn z komputerem PC odbywa się na dwa sposoby:

- przez interfejs UART przy użyciu aplikacji Minicom (Rys. 4.15)
- przez port Ethernet za pomocą klienta SSH

Oba rozwiązania były stosowane równolegle na każdym etapie projektu.

Zaletą konsoli podłączonej za pomocą portu szeregowego jest to, że są w niej wyświetlane komunikaty jądra Linuxa. Ułatwia to debugowanie błędów, które pojawiają się w fazie uruchamiania systemu operacyjnego. Zaletą protokołu SSH jest duża przepustowość i możliwość wysyłania nawet dużych plików.



```
wask@legion: ~/workspace/petalinux/perceptron
[    4.185732] FAT-fs (mmcblk0p1): Volume was not properly unmounted. Some data may be corrupt. Please run fsck.
[    4.293185] EXT4-fs (mmcblk0p2): re-mounted. Opts: (null)
hwclock: can't open '/dev/misc/rtc': No such file or directory
Tue Sep 29 14:42:29 UTC 2020
hwclock: can't open '/dev/misc/rtc': No such file or directory
[    4.999704] urandom_read: 2 callbacks suppressed
[    4.999716] random: dd: uninitialized urandom read (512 bytes read)
INIT: Entering runlevel: 5
Configuring network interfaces... [    5.296869] IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready
udhcpc: started, v1.29.2
udhcpc: sending discover
[    8.442041] macb e000b000.ethernet eth0: link up (1000/Full)
[    8.447751] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
udhcpc: sending discover
udhcpc: sending discover
udhcpc: no lease, forking to background
done.
Starting system message bus: [   14.850685] random: dbus-daemon: uninitialized urandom read (12 bytes read)
[   14.876830] random: dbus-daemon: uninitialized urandom read (12 bytes read)
dbus.
Starting haveged: haveged: listening socket at 3
haveged: haveged starting up

Starting Dropbear SSH server: [   15.035181] random: dropbear: uninitialized urandom read (32 bytes read)
dropbear.
hwclock: can't open '/dev/misc/rtc': No such file or directory
Starting internet superserver: inetd.
Starting syslogd/klogd: done
Starting tcf-agent: OK

PetaLinux 2019.2 neural /dev/ttys0
neural login: [   18.136905] random: crng init done
root
Password:
root@neural:~# 
CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7.1 | VT102 | Offline | ttys0
```

Rysunek 4.15. Uruchomienie systemu PetaLinux

5. Wyniki i wnioski

Celem pracy było zaprojektowanie, nauczenie i przetestowanie działania algorytmu Sztucznej Sieci Neuronowej z użyciem układu FPGA oraz porównanie z rozwiązaniem programowym. Podczas projektu powstało kilka modeli Sztucznej Sieci Neuronowej klasyfikującej odręcznie pisane cyfry. Aby porównać rozwiązanie, realizowane w technice HLS z implementacją przy użyciu pakietu *keras*, każdy z modeli poddano testom, które zostały podzielone na dwie części:

1. Uruchomienie sieci przy użyciu zbioru testowego 10000 cyfr z bazy MNIST, w celu oszacowania dokładności i szybkości działania algorytmu.
2. Test wykonany w czasie rzeczywistym przy użyciu modułu kamery.

Wyniki przeprowadzonych testów zostały zestawione w dalszej części rozdziału.

5.1. Test modelu sieci z jedną warstwą ukrytą

Wykonano test, podając na wejście sieci 10000 obrazów. Wynik testu, widoczny na Rys. 5.1 potwierdził poprawność działania algorytmu. Osiągnięto dokładność na poziomie 94,97%, co pokrywa się z wynikiem uzyskanym z wykorzystaniem biblioteki *keras*.

```
wask@legion: ~/workspace/petalinux/perceptron
[1] wask@legion: ~/workspace/petalinux/perceptron
[2] wask@legion: ~/workspace/petalinux/perceptron
[3] wask@legion: ~/workspace/hls/...
[4] wask@legion: ~/workspace/mgr...
[5] wask@legion: ~/workspace/mgr...
1st layer calculation took 0.636000 ms.
2nd layer calculation took 0.015000 ms.
x_test[9993]: Recognized digit: 0 with output = 0.000729
Reading file: x/x9994.txt
Setting input values.
1st layer calculation took 0.636000 ms.
2nd layer calculation took 0.014000 ms.
x_test[9994]: Recognized digit: 1 with output = 0.013115
Reading file: x/x9995.txt
Setting input values.
1st layer calculation took 0.636000 ms.
2nd layer calculation took 0.014000 ms.
x_test[9995]: Recognized digit: 2 with output = 0.020074
Reading file: x/x9996.txt
Setting input values.
1st layer calculation took 0.636000 ms.
2nd layer calculation took 0.014000 ms.
x_test[9996]: Recognized digit: 3 with output = 0.064796
Reading file: x/x9997.txt
Setting input values.
1st layer calculation took 0.636000 ms.
2nd layer calculation took 0.014000 ms.
x_test[9997]: Recognized digit: 4 with output = 0.391272
Reading file: x/x9998.txt
Setting input values.
1st layer calculation took 0.636000 ms.
2nd layer calculation took 0.014000 ms.
x_test[9998]: Recognized digit: 5 with output = 0.205821
Reading file: x/x9999.txt
Setting input values.
1st layer calculation took 0.635000 ms.
2nd layer calculation took 0.014000 ms.
x_test[9999]: Recognized digit: 6 with output = 0.063876
Misclassified: 503 digits.
Accuracy: 0.949700
End of test
root@neural:~#
```

Rysunek 5.1. Wynik testu uruchomionego na SBC Z-turn – ANN z jedną warstwą ukrytą

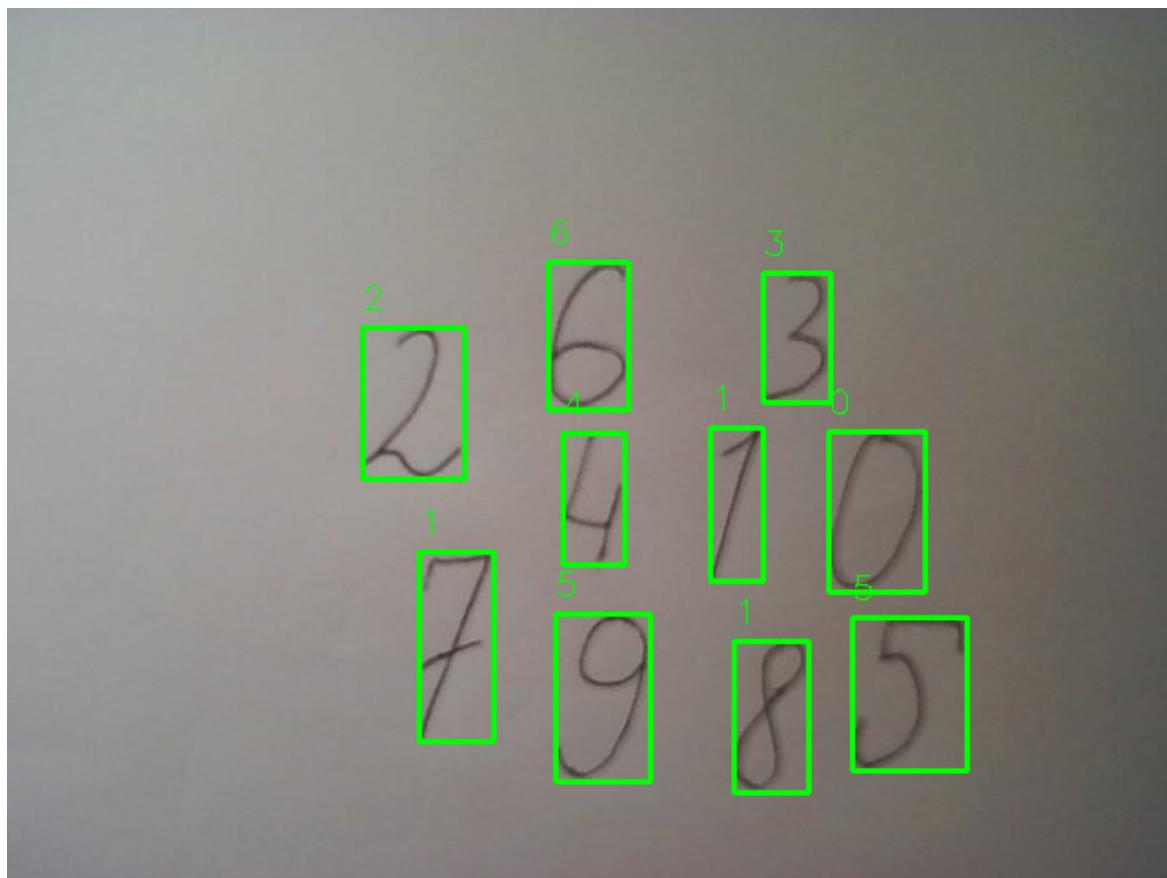
5.1.1. Test klasyfikacji cyfr z użyciem kamery

Następnym krokiem był test przeprowadzony w czasie rzeczywistym z użyciem kamery. Rozpoznawanie obiektów na obrazie w czasie rzeczywistym podzielono na 3 części:

- detekcja kształtów przypominających cyfry i odrzucenie niewłaściwych obiektów
- przygotowanie obrazów do klasyfikacji (odpowiedni rozmiar obrazu i padding)
- klasyfikacja obrazów przy użyciu ANN

Pierwszą symulację wykonano na komputerze PC przy użyciu biblioteki OpenCV i pakietu *keras*. Ze względu na sporą ilość obliczeń początkowo zdecydowano się na zarejestrowanie obrazu oraz detekcję cyfr przy użyciu biblioteki OpenCV.

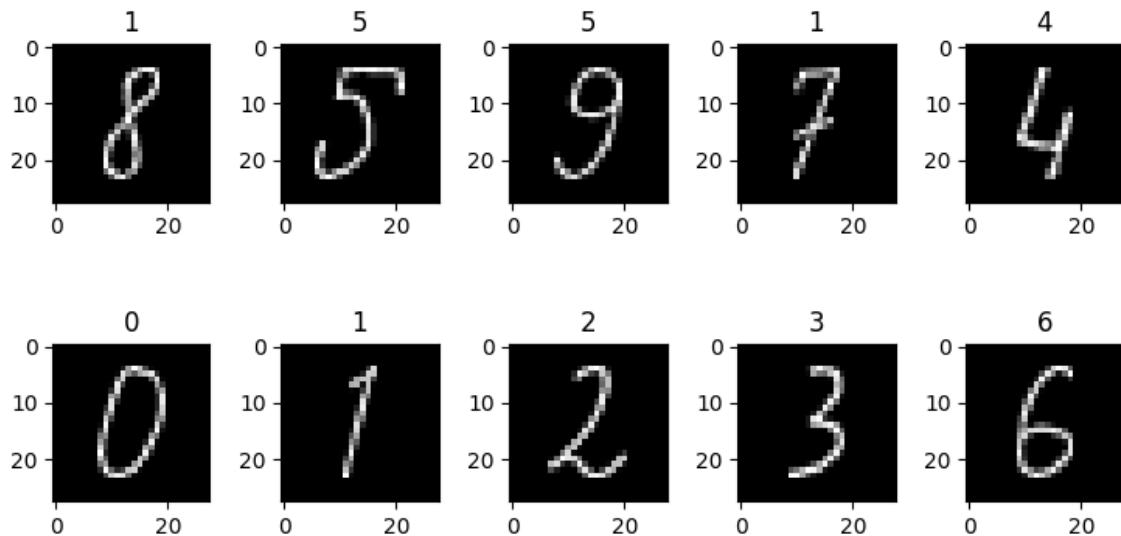
Obraz był rejestrowany w rozdzielcości 640x480 pikseli przy użyciu funkcji *cv2.VideoCapture(2)*. Następnym krokiem było przekształcenie barwy obrazu na skalę szarości, rozmycie obrazu oraz za pomocą funkcji *cv2.adaptiveThreshold* przekształcenie w obraz binarny. Następnie użyto funkcji *cv2.findContours*, która zwraca współrzędne prostokątów, w które wpisane są kontury znalezione przez algorytm. Po wyeliminowaniu niewłaściwych konturów można przejść do przygotowania obrazów do klasyfikacji.



Rysunek 5.2. Ramka obrazu podczas testu ANN z jedną warstwą ukrytą uruchomionego na PC

Odpowiednio przycięty do rozmiaru 28x28 pikseli i wycentrowany obraz ręcznie pi-

sanej cyfry może zostać poddany klasyfikacji za pomocą nauczonego wcześniej modelu ANN. W wyniku testu otrzymano wyniki przedstawione na Rys. 5.2.



Rysunek 5.3. Wynik testu ANN z jedną warstwą ukrytą uruchomionego na PC

Rysunek Rys. 5.3 zawiera znalezione na obrazie cyfry i wynik klasyfikacji (nad każdą z cyfr). Widać, że 3 z 10 cyfr zostały sklasyfikowane nieprawidłowo, co daje dokładność klasyfikacji sieci 70%.

5.1.2. Test na płytce Z-turn z użyciem kamery

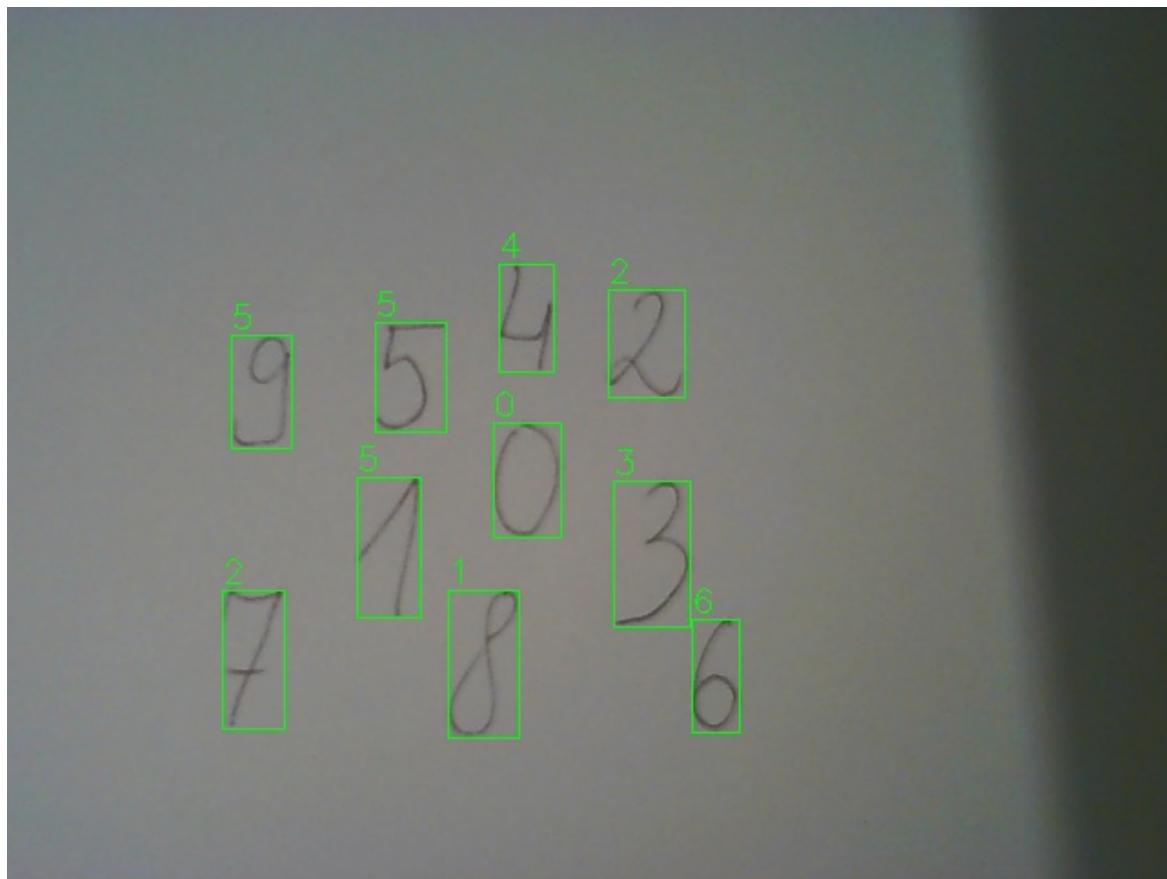
W teście na płytce Z-turn zastosowano metodę rejestracji obrazu taką jak na komputerze PC, jednak do rozpoznania cyfr wykorzystano model sieci ANN zaimplementowany w układzie FPGA. Aby umożliwić użytkownikowi wyświetlanie obrazu w czasie rzeczywistym, wykorzystano pakiety *pickle* i *socket* do wysyłania kolejnych ramek obrazu przez protokół TCP (ang. *Transmission Control Protocol*). Wynik testu przedstawiono na Rys. 5.4.

Po otrzymaniu wyników pierwszego testu podjęto decyzję o modyfikacji modelu Sztucznej Sieci Neuronowej. Pierwszą zmianą było dodanie kolejnej warstwy ukrytej.

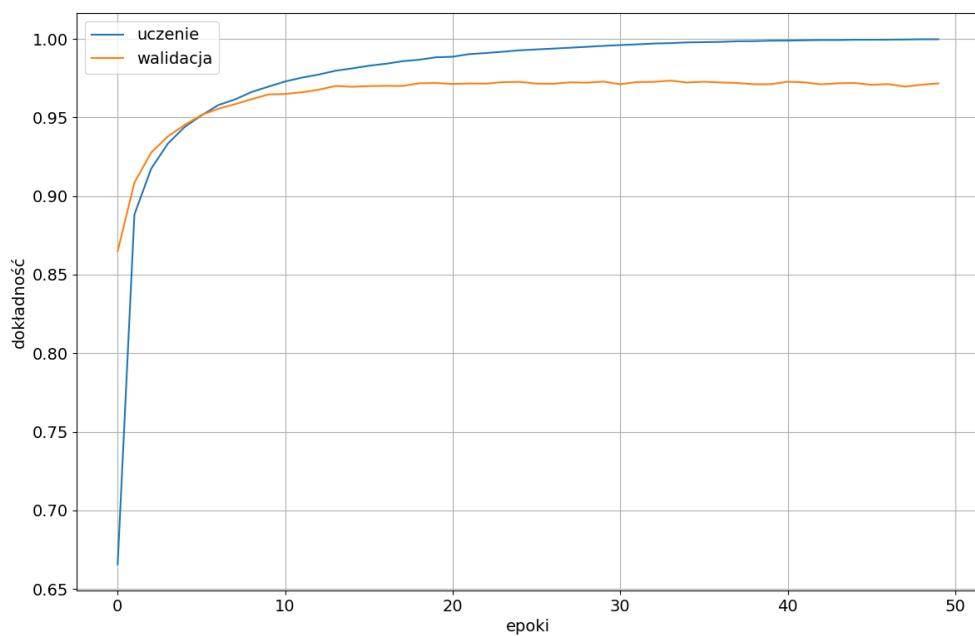
5.2. Test modelu posiadającego dwie warstwy ukryte

Dodano do istniejącego modelu kolejną warstwę ukrytą zawierającą 64 neurony. Po wykonaniu 50 epok otrzymano dokładność na poziomie 97,17%. Wykres zmiany dokładności w kolejnych epokach przedstawiono na Rys. 5.5.

5. Wyniki i wnioski



Rysunek 5.4. Wynik testu ANN z jedną warstwą ukrytą uruchomionego na płytce Z-turn Board



Rysunek 5.5. Wykres zmian dokładności w kolejnych epokach – ANN z dwoma warstwami ukrytymi

6. Podsumowanie

Bibliografia

- [1] D. Kriesel, *A Brief Introduction to Neural Networks*. 2007. adr.: <http://www.dkriesel.com>.
- [2] R. Tadeusiewicz, *Sieci neuronowe*. 1993.
- [3] A. Omondi i J. Rajapakse, "FPGA Implementations of Neural Networks", 2006.
- [4] I. Goodfellow, Y. Bengio i A. Courville, *Deep Learning*. MIT Press, 2016, <http://www.deeplearningbook.org>.
- [5] M. A. Nielsen, *Neural networks and deep learning*. Determination press San Francisco, CA, 2015, t. 2018.
- [6] R. Tadeusiewicz i M. Szaleniec, *Leksykon sieci neuronowych*. Projekt Nauka. Fundacja na rzecz promocji nauki polskiej, 2015.
- [7] S. S. Haykin i in., *Neural networks and learning machines/Simon Haykin*. 2009.
- [8] MYIR, *Company Profile*, Dostęp zdalny (22.09.2020): <http://www.myirtech.com/company.asp>, 2020.
- [9] *Z-turn IO Cape Schematic*, 2015.
- [10] Xilinx, *Vivado Design Suite User Guide (UG973 (v2019.2))*, 2019.
- [11] *PetaLinux Tools Documentation Reference Guide (UG1144 (v2019.2))*, 2019.
- [12] *Vivado Design Suite User Guide High-Level Synthesis (UG871 (v2019.2))*, 2020.
- [13] Y. LeCun i C. Cortes, "MNIST handwritten digit database", 2010. adr.: <http://yann.lecun.com/exdb/mnist/>.
- [14] C. Niroshan, *Interfacing a USB WebCam and Enable USB Tethering on ZYNQ-7000 AP SoC Running Linux*, Dostęp zdalny (29.09.2020): <https://medium.com/developments-and-implementations-on-zynq-7000-ap/interfacing-a-usb-webcam-and-enable-usb-tethering-on-zynq-7000-ap-soc-running-linux-1ba6d836749d>, 2017.
- [15] L. Hans-Jürgen Koch Linux developer, *The Userspace I/O HOWTO*, Dostęp zdalny (29.09.2020): <https://www.kernel.org/doc/html/v4.14/driver-api/uio-howto.html>, 2006.
- [16] Confluence, *Testing UIO with Interrupt on Zynq Ultrascale*, Dostęp zdalny (29.09.2020): <https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842490/Testing+UIO+with+Interrupt+on+Zynq+Ultrascale>, 2019.

Wykaz symboli i skrótów

- ANN** – ang. *Artificial Neural Network*
ASIC – ang. *Application-Specific Integrated Circuit*
BSP – ang. *Board Support Package*
DVP – ang. *Digital Video Port*
FPGA – ang. *Field Programmable Gate Array*
GPU – ang. *Graphics Processing Unit*
HLS – ang. *High Level Synthesis*
MPSoC – ang. *Multi-Processor System-on-Chip*
SBC – ang. *Single Board Computer*
SoC – ang. *System on Chip*
SSH – ang. *Secure Shell*
TCP – ang. *Transmission Control Protocol*
UIO – ang. *Userspace Input/Output (Driver)*
V4L2 – ang. *Video for Linux 2*

Spis rysunków

1.1.	Schemat w pełni połączonej jednokierunkowej Sieci Neuronowej	12
1.2.	Model neuronu	13
1.3.	Wykres funkcji ReLU	14
1.4.	Wykres funkcji sigmoid	14
1.5.	Model Perceptronu Wielowarstwowego	15
1.6.	Model Głębokiego Ucznia sieci	16
1.7.	Splot macierzy 6x6 przy użyciu filtra o rozmiarze 3x3	18
1.8.	Max-pooling o rozmiarze 2x2	19
1.9.	Rozpoznawanie obiektów na obrazie z kamery samochodu autonomicznego .	19
3.1.	Architektura serii ZYNQ-7000 SoC	23
3.2.	Płytki Z-turn-Board 7020	24
3.3.	Płytki rozszerzeniowe Z-turn IO Cape	25
3.4.	Moduł kamery MY-CAM011B BUS Camera Module	26
3.5.	Schemat portu DVP na płytce rozszerzeniowej IO-Cape	27
3.6.	Adapter złącza FPC/FFC o rastrze 0,5 mm na otwory DIP	28
3.7.	Moduł kamery MY-CAM002U USB Digital Camera Module	29
4.1.	Schemat blokowy systemu	31
4.2.	Schemat systemu przedstawiony w narzędziu Vivado	32
4.3.	Szczegółowy schemat części systemu <i>neural_net</i>	33
4.4.	Proces projektowania przy użyciu metody HLS	34
4.5.	Wybór płytki podczas tworzenia projektu w Vivado HLS	35

4.6. Fragment zbioru odręcznie pisanych cyfr MNIST	36
4.7. Wykres zmian dokładności w kolejnych epokach – ANN z jedną warstwą ukrytą	37
4.8. Wynik poprawnie przeprowadzonej symulacji w Vivado HLS	38
4.9. Wynik poprawnie przeprowadzonej implementacji w narzędziu Vivado	39
4.10. Tworzenie nowego projektu w programie Vitis	39
4.11. Wybór pliku .xsa z opisem konfiguracji sprzętu w programie Vitis	40
4.12. Wynik uruchomienia aplikacji w programie Vitis	41
4.13. Wstępna konfiguracja przy użyciu funkcji petalinux-config	42
4.14. Konfiguracja jądra PetaLinuxa	43
4.15. Uruchomienie systemu PetaLinux	45
5.1. Wynik testu uruchomionego na SBC Z-turn – ANN z jedną warstwą ukrytą	47
5.2. Ramka obrazu podczas testu ANN z jedną warstwą ukrytą uruchomionego na PC	48
5.3. Wynik testu ANN z jedną warstwą ukrytą uruchomionego na PC	49
5.4. Wynik testu ANN z jedną warstwą ukrytą uruchomionego na płytce Z-turn Board	50
5.5. Wykres zmian dokładności w kolejnych epokach – ANN z dwoma warstwami ukrytymi	50

Spis tabel

3.1. Porównanie cen płyt z układami Zynq firmy Xilinx	24
3.2. Porównanie testowanych modułów kamer	26