Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Wojciech Lepich

Nr albumu: 1146600

Rozpoznawanie cyfr przez sieć neuronową zaimplementowaną na układzie FPGA

Praca licencjacka
na kierunku Informatyka

Praca wykonana pod kierunkiem
dr. Grzegorza Korcyla
z Zakładu Technologii Informatycznych

Kraków 2020

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

.....	.....
Kraków, dnia	Podpis autora pracy

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

.....	.....
Kraków, dnia	Podpis kierującego pracą

Spis treści

Wstęp

Wraz z intensywnym w ostatnich latach rozwojem dziedziny nauczania maszynowego, w przemyśle komputerowym na dobre zagościło rozpoznawanie obrazów. Stało się to możliwe dzięki nieustającemu postępowi technologicznemu oraz dostępowi do coraz większej ilości zasobów komputerowych. Nowe architektury sieci neuronowych rosną do niewyobrażalnych kilkadziesiąt lat temu rozmiarów. Przy ciągle rosnącej ilości danych potrzebnych do przetworzenia w czasie rzeczywistym, konieczne jest zastosowanie specjalistycznego sprzętu. Potencjał drzemiący w układach FPGA w postaci ogromnego zrównoleglenia obliczeń, którego nie są w stanie zaoferować układy CPU, elastyczności, której nie dają układy ASIC oraz niskich opóźnień, których nie da się osiągnąć programując układy GPU, można wykorzystać budując odpowiednio przygotowany model sieci.

W niniejszej pracy zaimplementowano na FPGA model sieci neuronowej rozpoznającej cyfry. Przygotowano również część odpowiedzialną za wstępne przygotowanie obrazu z kamery i wyświetlającej go na monitorze oraz tej przedstawiającej wyniki na terminalu komputera.

Praca składa się ze wstępu, trzech rozdziałów podsumowania oraz bibliografii.

W rozdziale pierwszym przedstawione są zagadnienia teoretyczne dotyczące projektu, potrzebne dla zrozumienia reszty pracy. Znajdują się tam opis architektury FPGA, problematyka przetwarzania obrazów oraz krótki wstęp do rozległego tematu sieci neuronowych.

Rozdział drugi to przedstawienie projektu zaczynając od opisu sprzętu oraz wykorzystanych narzędzi, następnie wyjaśnione zostają kroki czynione w trakcie tworzenia projektu, na koniec przedstawiono informacje dotyczące uruchamiania.

Ostatni rozdział zawiera wyniki predykcji sieci wraz z dyskusją co wpływa na te osiągnięte oraz na jakie sposoby można potencjalnie uzyskać ich poprawę.

Na zakończenie składam serdeczne podziękowania dr. Grzegorzowi Korcylowi za pomoc w nakreśleniu projektu oraz cenne uwagi podczas pisania pracy.

1 Teoria

1.1 Architektura FPGA

Field-Programmable Gate Array (FPGA) to układy scalone, które mogą być elektronicznie przeprogramowane bez potrzeby demontażu samego układu z urządzenia. W porównaniu do układów ASIC znacznie taniej zaprojektować pierwszy działający układ. Elastyczna natura układów FPGA wiąże się z większym zużyciem powierzchni krzemu, opóźnień oraz zużycia energii.[1]

Podstawowa struktura układów FPGA składa się z różnych bloków logicznych, które mogą być łączone ze sobą w zależności od wymagań projektowych. Przykładami takich bloków są: DSP (jednostka przeprowadzająca obliczenia dodawania/mnożenia), LUT (Look-Up Table, de facto tablica prawdy dowolnej funkcji boolowskiej), Flip Flop (przechowują wynik LUT), BRAM (Block RAM, pamięć dwuportowa, jest w stanie przechowywać względnie dużą ilość danych).

Układy FPGA przeważnie pracują na kilku-, kilkunastukrotnie niższych częstotliwościach niż CPU. Osiągają wysoką wydajność dzięki masywnemu zrównolegleniu obliczeń.

Programowanie FPGA polega na pisaniu logiki w językach HDL (Hardware Description Language) takimi jak VHDL czy też Verilog. Napisana logika definiuje zachowanie układu FPGA. Gotowy opis logiki syntetyzuje się, czyli generuje połączenia pomiędzy zasobami układu. Kolejnym etapem jest implementacja — odzworowanie połączeń w konkretnym układzie.

HLS (High-Level Synthesis) to proces ułatwiający pisanie skomplikowanej logiki. Algorytmy można pisać w językach wysokiego poziomu, takich jak C, C++, SystemC. Przygotowany kod jest transpilowany poprzez odpowiedni kompilator HLS do języka RTL (Register-Transfer Level; język opisu sprzętu na poziomie bramek i rejestrów), a ten może być zaimplementowany na układzie.

1.2 Przetwarzanie obrazu

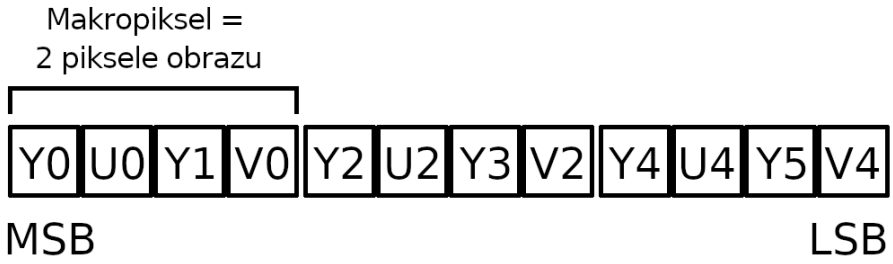
Cyfrowe przetwarzanie obrazu jest problemem wymagającym dużych mocy obliczeniowych ze względu na ilość danych do przetworzenia. Nieskompresowany kolorowy obraz z pikselami w formacie RGB (po 8 bitów na kolor) o wysokości 720 pikseli i szerokości 1280 pikseli to 22118400 bitów ($\approx$ 2,5MB). Obraz przetwarzany w czasie rzeczywistym, na przykład z kamery, zwielokrotnia tę liczbę o liczbę klatek na sekundę (przy trzydziestu klatkach na sekundę liczba danych rośnie do około 79 megabajtów na sekundę). Należy również pamiętać, że dane są dwuwymiarowe co jest ważne przy problemach związanych z rozpoznawaniem wzorców, klasyfikacją przedmiotów na obrazie, filtrowania w celu rozmazania lub wyostrzenia obrazów itp.

1.2.1 Formaty pikseli

Jest wiele modeli przestrzeni barw (a co za tym idzie, sposobów kodowania pikseli) między innymi:

- RGB, używany w aparatach, skanerach, telewizorach
- CMYK, używany w druku wielobarwnym
- HSV
- YUV

Składowe dwóch ostatnich przestrzeni barw oddzielają informację o jasności od informacji o kolorach. Model barw YUV składa się z kanału luminacji Y i kanałów kodujących barwę U oraz V, są to kolejno składowa niebieska i składowa czerwona. W projekcie użyty jest format pikseli YUY2 (znany też pod nazwą YUYV), w którym na dwa piksele przypadają 32 bity. Licząc od najstarszego bitu, pierwsze osiem bitów przypada na Y0 (patrz rys. 1), to jest luminacja pierwszego piksela; następne osiem bitów na U0; kolejne osiem bitów to luminacja drugiego piksela; pozostałe bity to składowa czerwona V0. Dla obydwóch pikseli składowe U i V są wspólne. Co istotne w projekcie łatwo oddzielić luminację, która jest używana w przetwarzaniu obrazu.



Rysunek 1: Schemat formatu pikseli YUV2

1.3 Sieci neuronowe

Sztuczna sieć neuronowa (SSN) jest modelem zdolnym do odwzorowania złożonych funkcji. Najprostsze sieci są zbudowane ze sztucznych neuronów, z których każdy posiada wiele wejść oraz jedno wyjście, które może być połączone z wejściami wielu innych neuronów. Każde z wejść neuronu jest związane ze znalezioną w procesie trenowania wagą. Wartość wyjścia to obliczony wynik funkcji aktywacji z sumy ważonych wejść. Sieć może mieć wiele warstw neuronów ukrytych, których wejściami są wyjścia neuronów z poprzedniej warstwy.

Sieci neuronowe są stosowane w problemach związanych z predykcją, klasyfikacją, przetwarzaniem i analizowaniem danych. Do ich zastosowania nie jest potrzebna znajomość algorytmu rozwiązania danego problemu. Obliczenia w sieciach są wykonywane równolegle w każdej warstwie, dzięki czemu implementacja sieci na układzie FPGA może działać wielokrotnie szybciej niż na CPU, pomimo niższej częstotliwości układu.

2 Opis projektu

2.1 Zarys projektu

Celem projektu jest implementacja systemu do rozpoznawania cyfr w czasie rzeczywistym. Cel zrealizowano poprzez implementację wtyczki GStreamera, wykorzystującej sieć neuronową na układzie Xilinx Zynq MPSoC oraz stworzenie odpowiedniego potoku danych korzystając z bibliotek GStreamer. Zadaniami spoczywającymi na innych elementach potoku jest obsługa kamery, kadrowanie i skalowanie obrazu oraz wyświetlenie go na końcowym urządzeniu.

2.2 Platforma

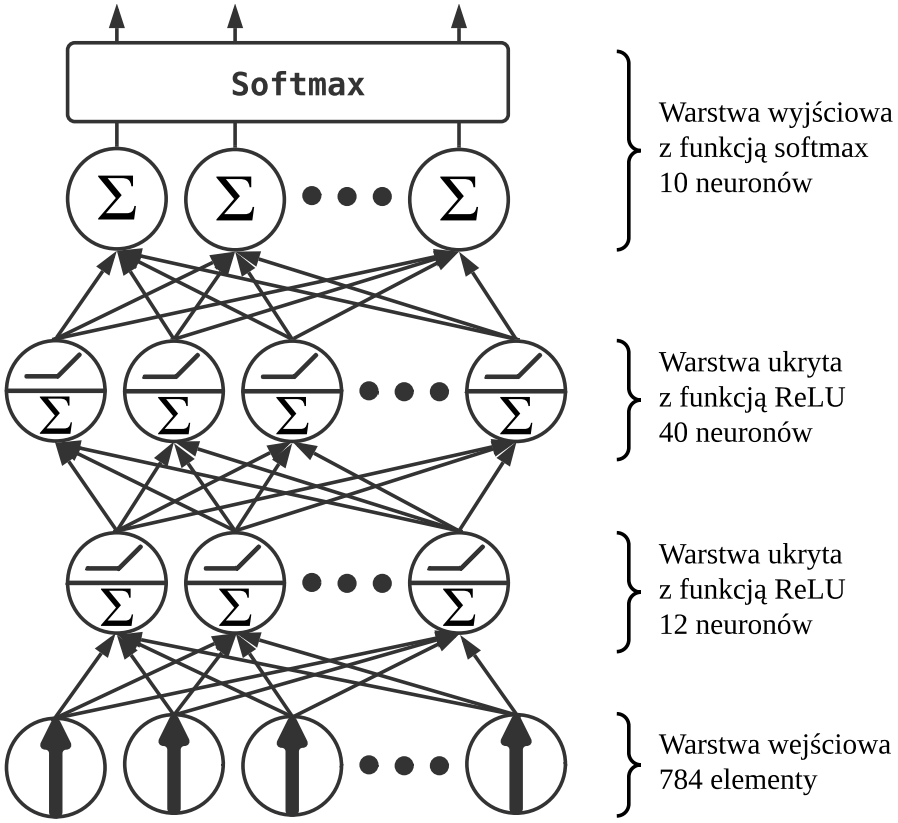
Rysunek 2: Stanowisko testowe

Sprzęt wykorzystany w projekcie to Xilinx Zynq UltraScale+ MPSoC ZCU104. Na jednym układzie znajduje się czterordzeniowy procesor ARM Cortex-A53, dwurdzeniowy procesor ARM Cortex-R5, układ graficzny Mali-400 oraz zasoby FPGA. Całość projektu została oparta o platformę Xilinx reVISION[2]. Przetwarzane dane dostarczane są z kamery USB, która była dołączona w zestawie z płytą Zynq. Urządzeniem końcowym jest telewizor połączony przewodem HDMI z płytą.

2.3 Sieć neuronowa

Architektura sieci została dobrana uwzględniając dostępne zasoby programowalnej logiki na płycie, a także możliwości sprzętu na którym dokonywana była jej synteza. Dla problemu klasyfikowania obrazów dobrze nadają się sieci splotowe (konwolucyjne, ang. convolutional neural networks — CNN), których przykładem jest popularna sieć LeNet-5.[3] Architektura ta zawiera zarówno w pełni połączone warstwy oraz warstwy splotowe i łączące. Niestety, z powodu ograniczeń sprzętowych, w pracy nie została użyta ta architektura.

Model wykorzystany w projekcie posiada 2 warstwy ukryte, posiadające kolejno 12 i 40 neuronów aktywowanych funkcją ReLU oraz warstwy wyjściowej złożonej z 10 neuronów z funkcją aktywacji softmax.



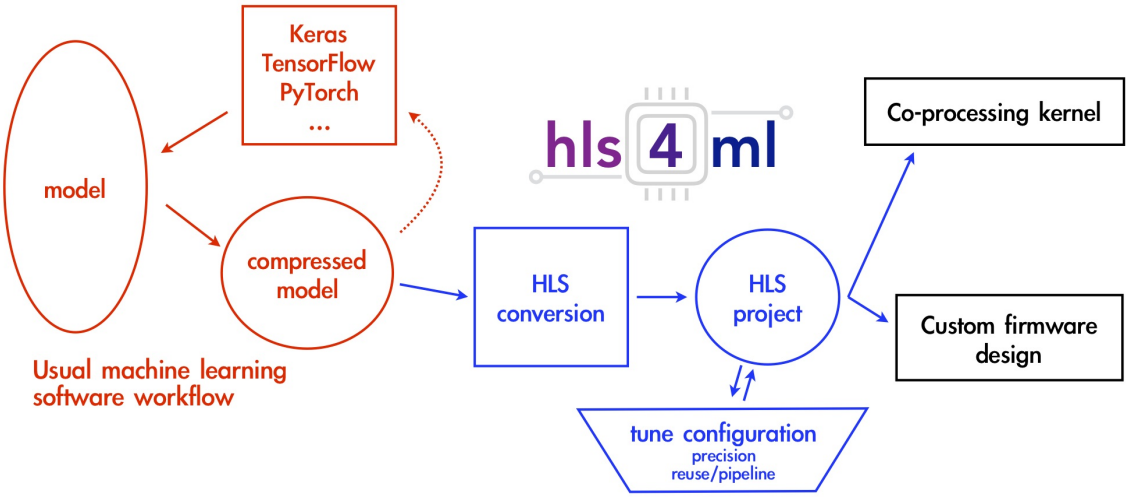
Rysunek 3: Schemat użytej architektury.

Do stworzenia sieci wykorzystano bibliotekę TensorFlow. Sieć uczona była na danych z bazy MNIST[4] składającej się łącznie z 70000 przykładów cyfr na obrazach o wielkości 28×28 pikseli, z których każdy przedstawiony jest jako wartość od 0 (kolor czarny) do 255 (kolor biały). Próbki zostały podzielone na zbiór uczący, liczący 60000 próbek, oraz zbiór do testów z pozostałych 10000 cyfr. Cyfry składają się z białych pikseli, tło jest czarne. Dokonano prób trenowania sieci przetworzonymi danymi, w których kolory były odwrócone, natomiast wytrenowane modele w trakcie testów nie przekraczały progu czterdziestu procent dobrze zaklasyfikowanych obrazów.

2.4 hls4ml

2.4.1 Idea hls4ml

Celem projektu hls4ml jest wygenerowanie kodu C++ na podstawie zapisanego modelu z TensorFlow. Przykładowy schemat pracy z hls4ml przedstawiony jest na rysunku 4. Czerwona część schematu pokazuje ogólną organizację pracy przy projektowaniu odpowiedniego modelu uczenia maszynowego. Niebieska część należy do hls4ml, który tłumaczy dostarczony model z wagami do syntetyzowalnego kodu, który następnie można włączyć do większego projektu lub zaimplementować jako samodzielną część na FPGA. Generowany projekt jest parametryzowany przez plik konfiguracyjny yml zawierający ścieżkę do pliku z zapisanym modelem wraz z wagami, typy danych kodujące wartości wag, nazwę docelowego układu FPGA oraz parametry optymalizacji dotyczące zużycia zasobów — większa ilość zasobów oznacza zrównoleglenie większej części obliczeń.

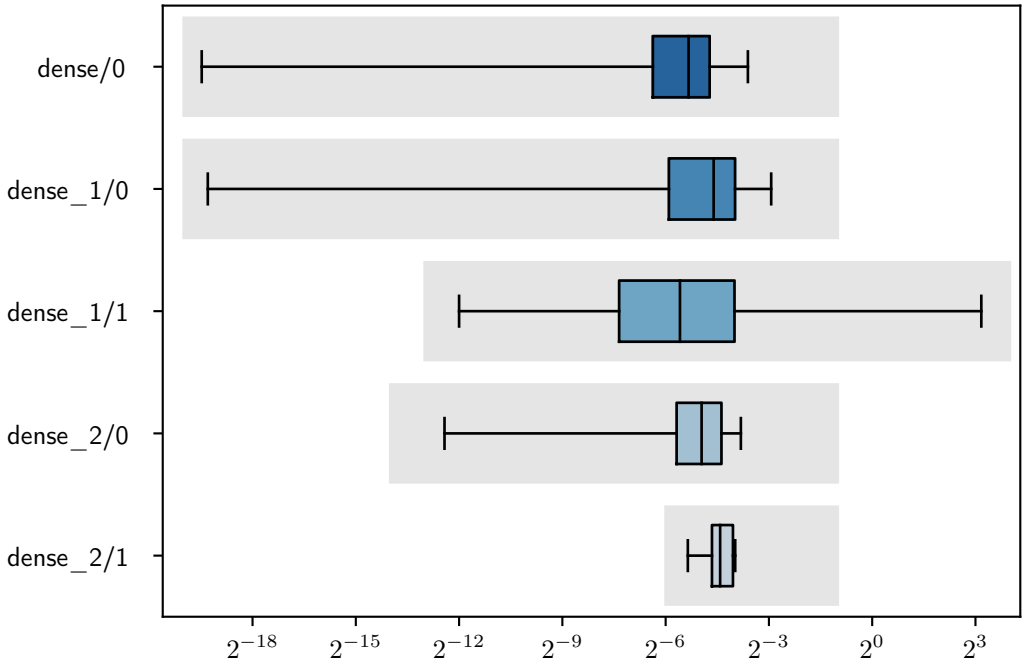


Rysunek 4: Schemat pracy z hls4ml.
Dokumentacja hls4ml, https://fastmachinelearning.org/hls4ml/CONCEPTS.html

2.4.2 Precyzja danych

Typ danych używany w przekonwertowanym modelu to duże liczby stałoprzecinkowe (`ap_fixed`) oraz liczby całkowite (`ap_int`). Precyzję obu typów można ustalić do jednego bita. Obliczenia przeprowadzane na liczbach o mniejszej precyzji umożliwiają większe zrównoleglenie obliczeń, natomiast zbyt niska precyzja może poskutkować bezużytecznością zsyntetyzowanej sieci. Aby odpowiednio dobrać precyzję wag, skorzystano z pythonowej biblioteki hls4ml.profiling.

Program korzystający z funkcji dostarczanych przez tę bibliotekę analizuje plik konfiguracyjny yml oraz model z pliku h5. Wynikiem działania programu jest wykres (patrz rys. 5) przedstawiający rozkład wartości wag każdej z warstw modelu otrzymanych w procesie trenowania. Szare pole w tle wykresu przedstawia zakres wartości, które obejmowane są przez precyzję określoną w pliku konfiguracyjnym. Dobrym punktem początkowym jest wybranie takiej liczby bitów dla każdej z warstw, która obejmuje wszystkie możliwe wagi. Dalsze ustalanie precyzji można wykonać w trakcie analizy wyników symulacji.



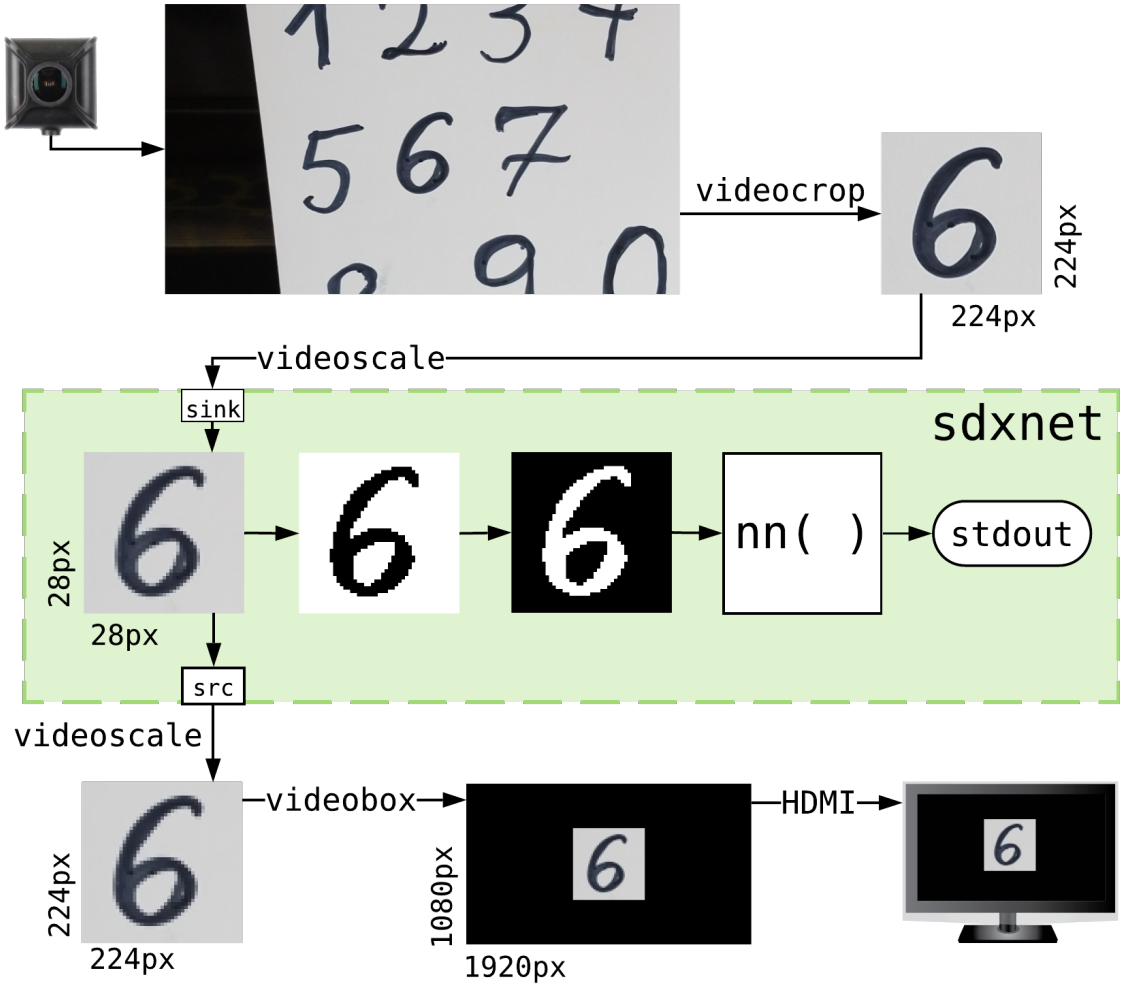
Rysunek 5: Rozkład wartości wag użytego modelu

2.5 GStreamer

Zsyntetyzowana sieć jest częścią projektu. Potrzebne jest również dostarczenie danych do sieci oraz przedstawienie wyniku. Do tego celu skorzystano z frameworka GStreamer, dzięki któremu można tworzyć grafy z komponentów (pluginów, elementów) przetwarzających media, zarówno audio jak i video. Każdy z elementów grafu składa się z co najmniej jednego źródła (source), lub ujścia (sink), może mieć również wiele wejść i wyjść. W grafie pierwszy element nie może mieć wejść, natomiast konieczne jest aby posiadał co najmniej jedno wyjście. Poprawnie przygotowany graf nie powinien mieć komponentów oferujących źródło, które nie są z niczym połączone. Pluginy mają ujednolicony interfejs, dzięki czemu można w łatwy sposób włączyć do grafu własny element. Wtyczki charakteryzują się pewnymi własnościami, znanymi jako „caps". Określają one jakie media jest w stanie przetworzyć dana wtyczka (na przykład format pikseli, maksymalny rozmiar obrazu). Łączone ze sobą elementy dokonują negocjacji parametrów mediów, takich jak rozdzielczość obrazu, format pikseli, ilość klatek na sekundę oraz innych. Wszystkie wtyczki wypisane niżej (poza xlnxvideosink, xlnxvideosrc od Xilinx oraz sdxnet, będącego własną implementacją) są dostępne wraz z instalacją GStreamera.

2.5.1 xlnxvideosrc i xlnxvideosink

Są to pluginy dostarczone przez firmę Xilinx wraz z platformą reVISION. Obydwa korzystają biblioteki Xilinx `video_lib`. Pierwszy z nich ułatwia odczytywanie danych ze źródeł, dla których potrzebne byłyby dodatkowe działania. Są to między innymi kamera USB (użyta w projekcie), HDMI, MIPI CSI (sprzętowy interfejs do transmisji obrazów i wideo). Sam element zbudowany jest w oparciu o element v4l2src[5, s.33], dostępny w standardowej instalacji GStreamera. Xlnxvideosink również jest oparty o inny element



Rysunek 6: Schemat logiczny przetwarzania obrazu

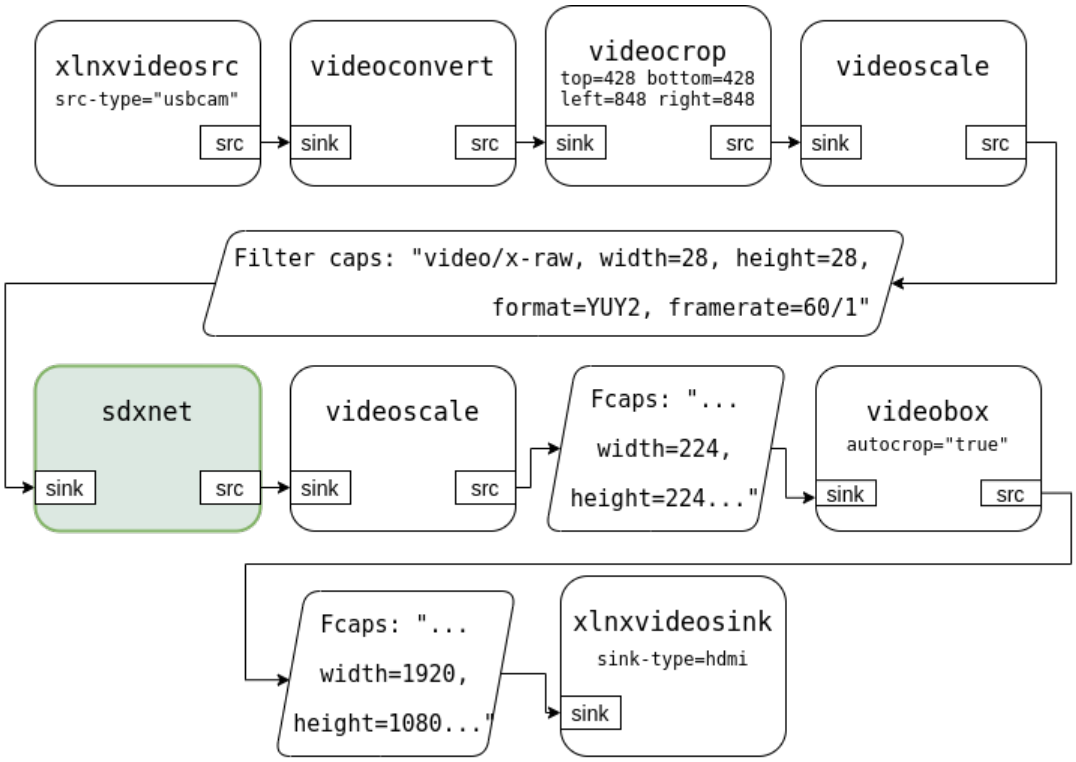
— kmssink[5, s.33]. Zapewnia odpowiednią konfigurację połączenia z wyświetlaczami podłączonymi przez HDMI oraz DisplayPort.

2.5.2 videoconvert

Element mający za zadanie dostosować wszystkie parametry obrazu tak, aby móc połączyć ze sobą dwa niekompatybilne pod względem „caps" elementy. Ta niekompatybilność może być spowodowana na przykład tym, że dwie wtyczki potrzebują innego formatu pikseli i jednocześnie nie oferują możliwości konwersji z jednego formatu na inny.

2.5.3 videocrop

Wtyczka służąca do wykadrowania obrazu w zdefiniowanym obszarze. Wykorzystana została aby otrzymać obraz o tej samej długości i szerokości wynoszącej 224 (co jest ośmiokrotnością 28, czyli długością boku obrazów, którymi wytrenowana została sieć) wycięty ze środka wideo o rozmiarze 1920×1080.



Rysunek 7: Schemat grafu. Na zielono plugin z siecią neuronową

2.5.4 videoscale

Skaluje obraz do wynegocjowanych pomiędzy sąsiadującymi elementami parametrów, przy czym pierwsza próba negocjacji to ta sama wielkość obrazu przy ujściu jak i w źródle, aby skalowanie nie było potrzebne.

2.5.5 sdxnet

Sdxnet to wtyczka wykorzystująca sieć neuronową do rozpoznania cyfr znajdujących się na obrazie przez nią przechodzącym. Element ten został zaimplementowany na potrzeby tego projektu.

2.5.6 Filter caps

Element precyzujący parametry obrazu, które wymuszają dostosowanie się poprzedniego elementu — na przykład videoscale. Zapisuje się go w postaci ciągu znaków ujętych w cudzysłów.

2.5.7 videobox

Oferuje możliwość osadzenia obrazu w tym o innym rozmiarze wypełniając pozostałą przestrzeń ramką w wybranym kolorze. Własność autocrop oznacza automatyczne obliczenie wielkości ramek na podstawie parametrów określonych przez kolejny element tak, aby obraz przychodzący do videobox był wycentrowany a ramki były tej samej wielkości.

2.5.8 fpsdisplaysink

Wtyczka typu sink (mająca tylko ujście), która jako parametr pobiera inną wtyczkę tego typu, np. xlnxvideosink, zastępując w grafie tamtą. Jej użycie pozwala na sprawdzenie liczby klatek na sekundę wyświetlanego obrazu.

2.6 Używanie sieci

2.6.1 Generowanie projektu

Architekturę sieci wraz z wagami zapisano do pliku h5. Stworzono plik konfiguracyjny hls4ml. Następnie na jego podstawie wygenerowano projekt z przekonwertowaną siecią. Wśród wygenerowanych plików znajduje się również kod służący do symulacji działania projektu. Przygotowane zostały pliki z danymi testującymi sieć — 10000 przetworzonych przykładów z bazy MNIST tak, aby cyfry były koloru czarnego, tło białego. Zakres wartości wynosi od 0 do 255.

2.6.2 Funkcja

Cała sieć jest przedstawiona jako jedna funkcja o nazwie `nn`. Parametrami tej funkcji są dwie tablice: `input[]`, do której są zapisywane są dane do przetworzenia, oraz `output[]`, do której funkcja zapisuje obliczone predykcje.

```
1 void nn(
2     unsigned char input[784],
3     float         output[10]
4 );
```

Listing 1: Nagłówek funkcji

2.6.3 Interfejs

Aby móc korzystać z sieci w aplikacji uruchamianej na procesorze ARM zadeklarowano użycie interfejsu IO AXI-4 Lite.

2.6.4 Dostosowanie sieci

W celu poprawienia wyników działania sieci dokonano pewnych usprawnień. Sieć została wytrenowana oryginalnymi danymi, w których piksele tworzące cyfry mają wartości równe 255 lub tej wartości bliskie, a piksele białego są przedtawione jako 0. Ponadto rzeczywiste dane z kamery mogą być zaszumione, przedstawione obiekty zacienione, a same cyfry mogą nie być idealnie czarne.

Przy każdym wywołaniu funkcji sieci dokonywana jest transformacja danych poprzez kod pokazany na listingu 2. Wartość każdego piksela jest zamieniana na wartość 255 lub 0, zależnie od początkowej jego wartości — dla wartości mniejszych od 140 (kolor szary lub ciemniejszy) przypisany jest kolor biały, wartość 255. Dla pikseli jasnych (od 140 w górę) przypisywana wartość to 0, kolor czarny.

W ten sposób dokonuje się zarówno odpowiedniego przetworzenia danych uwzględniającego sposób wytrenowania modelu, jak również uwydatnienia cyfry oraz pozbycia się szumów obrazu i jasnych cieni.

```
for(int i=0; i<784; i++) {
#pragma HLS pipeline II=1
        input1[i] = (input[i] < 140) ? 255 : 0;
}
```

Listing 2: Transformacja danych.

- `input[]` — tablica z danymi (parametr funkcji)
- `input1[]` — dane przetwarzane przez sieć

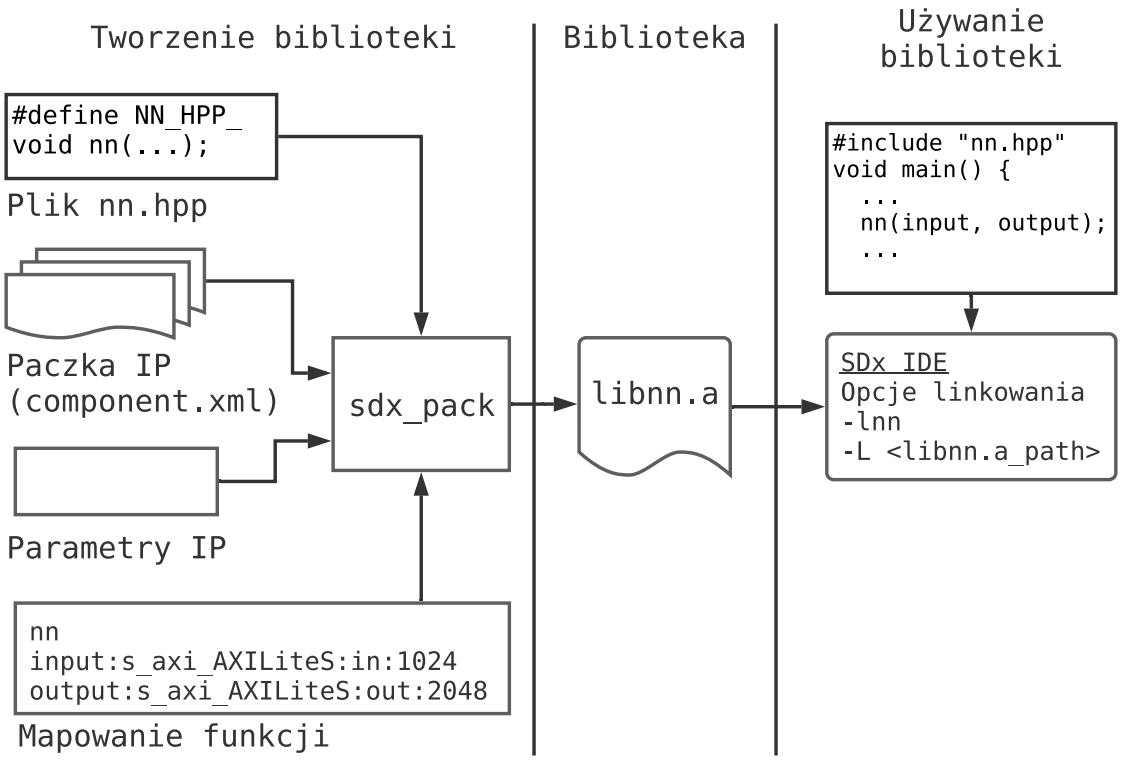
2.6.5 Tworzenie biblioteki

Zsyntetyzowany moduł sieci został wyeksportowany w środowisku Vivado HLS do paczki IP (Intelectual Property). Następnie poprzez narzędzie `sdx_pack` utworzono statyczną bibliotekę gotową do wykorzystania w innym projekcie w środowisku SDSoC (Software-Defined System on Chip, IDE do pisania aplikacji lub bibliotek uruchamianych na platformach Xilinx MPSoC).

Wywołując narzędzie `sdx_pack` należy podać plik nagłówkowy funkcji, docelową nazwę biblioteki, ścieżkę do pliku `component.xml` wygenerowanego podczas eksportu do paczki IP, mapowanie parametrów funkcji na porty modułu, protokół kontroli modułu, odpowiedni zegar oraz informacje dotyczące docelowej platformy: nazwę jej rodziny, procesora oraz systemu.

```
sdx_pack -header nn.hpp -lib libnn.a \
         -func nn -map input=s_axi_AXILiteS:in:1024 \
                  -map output=s_axi_AXILiteS:out:2048 \
         -func-end \
         -ip ip/component.xml \
         -control ap_ctrl_hs=s_axi_AXILiteS:0 \
         -primary-clk ap_clk=13.333 \
         -target-family zynquplus \
         -target-cpu cortex-a53 \
         -target-os linux
```

Listing 3: Narzędzie sdx_pack



Rysunek 8: Schemat tworzenia biblioteki
Za *SDSoC Environment User Guide: C-Callable IP Libraries*, https://china.xilinx.com/html_docs/xilinx2018_3/sdsoc_doc/c-callable-libraries-zah1504034388097.html

2.7 Implementacja pluginu sdxnet

Dalsza część implementacji składa się z dwóch części. Pierwsza z nich to `neuralnet`, biblioteka statyczna, która odpowiada za inicjalizację i zwolnienie pamięci dla danych, z których korzystają funkcje akcelerowane sprzętowo, wyciągnięcie danych dotyczących luminacji z obrazu, wywołanie funkcji `nn` (patrz rozdział 2.6.2) oraz za utworzenie obrazu wychodzącego z pluginu. Druga część to biblioteka dzielona `gstsdxnet`, korzystająca z funkcji z biblioteki `neuralnet`. Biblioteka ta jest rozpoznawana jako plugin GStreamer, można więc z niej korzystać przy budowaniu grafów za pomocą GStreamera. Schemat wykonywania kodu przedstawiony jest na stronie 16.

2.7.1 neuralnet

Dane obrazu (kanał luminacji i kanał chrominacji) przechowywane są w obiektach klasy `xf::Mat` będącej częścią bibliotek xfOpenCV. Określona jest wysokość oraz szerokość obrazu, typ pikseli kodujących obraz (w tym przypadku ośmiobitowe bez znaku z jednym kanałem pikseli). Pamięć dla tych obiektów oraz dla tablicy przechowującej predykcje sieci alokowana jest przez funkcję `net_init_sds`, a zwalniana przez funkcję `net_deinit_sds`. Dla każdej klatki obrazu z poziomu `gstsdxnet` wywoływane są kolejno trzy funkcje: `net_sds_read`, `net_sds_predict` oraz `net_sds_write`.

Pierwsza z nich rozkodowuje przy pomocy masek i przesunięć bitowych dane z pikseli i zapisuje je do dwóch kanałów (obiektów `xf::Mat`). Ostatnia, również używając tych samych bitowych operacji, łączy obydwa kanały tworząc wyjściowy obraz, przy

czym dla kanału luminacji są to dokładnie te same dane, które zostały odczytane przez `net_sds_read`, natomiast wszystkie bajty kanału chrominacji mają wartość 128, aby wyjściowy obraz był w skali szarości (dzięki czemu obraz na monitorze jest bardziej zbliżony do tego rzeczywiście przetwarzanego). Obydwie funkcje korzystają z funkcji akcelerowanych sprzętowo aby zrównoleglić i przyspieszyć w ten sposób proces. Funkcja `net_sds_predict` wywołuje funkcję `nn` oraz kopiuje otrzymane predykcje do tablicy otrzymanej z `gstsdxnet`.

Przy kompilacji biblioteki należy dołączyć bibliotekę statyczną powstałą w procesie opisanym w rozdziale 2.6.5.

2.7.2 gstsdxnet

Framework GStreamer jest w stanie rozpoznać bibliotekę `gstsdxnet` dzięki napisaniu funkcji spełniających wymagania stawiane przez framework. Obejmują one inicjalizację pluginu (m.in. alokacja pamięci, zdefiniowanie możliwych parametrów mediów na wejściu i na wyjściu, zapisanie metadanych), funkcję przetwarzającą klatki obrazu oraz te odpowiadające za prawidłowe zakończenie pracy wtyczki. Funkcja inicjalizująca plugin uruchamia też funkcję inicjalizującą z biblioteki `neuralnet`. Funkcja przetwarzająca obraz (`gst_sdx_process_frames` przygotowuje dane, które przekazuje do odpowiednich funkcji z `neuralnet`. Dla co dwudziestej piątej klatki obrazu na standardowe wyjście zapisywany jest wynik predykcji w postaci pary liczb (przewidywana cyfra, prawdopodobieństwo).

Kod wtyczki oparto o kod przykładowej wtyczki dostarczonej wraz z platformą reVISION 2018-3 przez Xilinx.

Inicjalizacja pluginu

```
gst_sdx_net_set_caps(...) {
   ...

   net_init_sds(height,
      width, pixel_form,
      data_pointer)

   ...
}
```

```
net_init_sds(..., data_pointer) {
   struct * data = malloc(...)
   data->luma = new xf::Mat...
   ...
   *data_pointer = data
}
```

Przetwarzanie klatki obrazu

```
gst.._process_frames(
   *in_frame, *out_frame,
   ...) {
   ...
   net_sds_read(
      in_frame_data,
      data_pointer, ...)
```

```
net_sds_read(*in_frame_data,
   *data, ...) {
   ...
   read_net_input(
      in_frame_data,
      data->luma,
      data->uv, ...)
}
```

read_net_input

```
   net_sds_predict(
      predictions_pointer,
      data_pointer)
```

```
net_sds_predict(*predictions,
   *data) {
   ...
   nn(data->luma, predictions)
   ..
}
```

nn

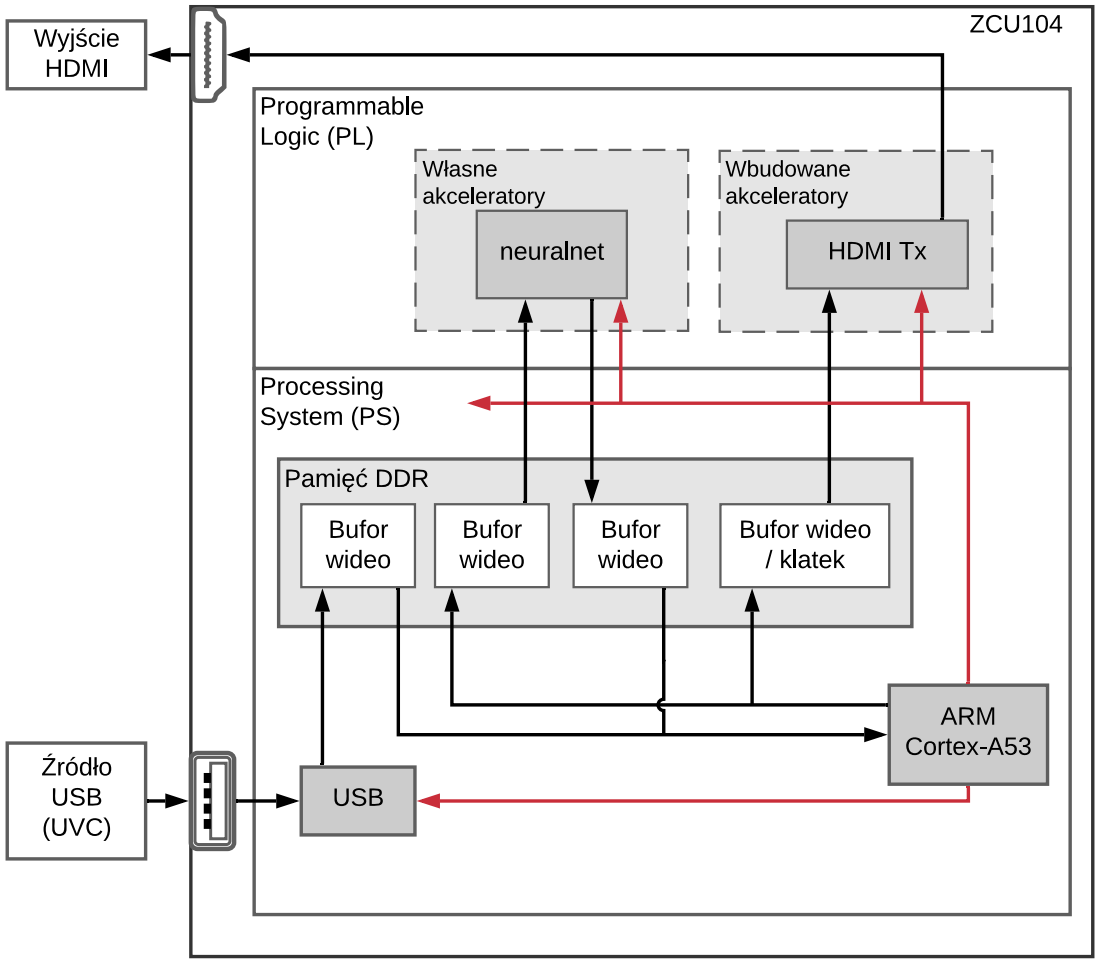
```
   digit, prob =
   get_max(predictions_pointer)

   printf("%d %f", digit, prob)

   net_sds_write(
      out_frame_data,
      data_pointer, ...)

   ...

}
```

```
net_sds_write(*out_frame_data,
   *data, ...) {
   ...
   write_net_output(
      data->luma,
      data->uv,
      out_frame_data...)
}
```

write_net_output

2.8 Uruchamianie

Aby uruchomić pipeline przedstawiony na rysunku 7 można skorzystać z narzędzia `gst-launch-1.0`. Narzędzie ułatwia i przyspiesza uruchamianie różnych strumieni przetwarzających media, ponieważ nie ma potrzeby zaprogramowania aplikacji. Narzędzie jako argumenty wywołania programu pobiera nazwy pluginów oddzielone znakiem wykrzyknika w kolejności w jakiej mają wystąpić w grafie. Skrypt przedstawiono na listingu 4 na stronie 18.



Rysunek 9: Uproszczony schemat przedstawiający działanie projektu na sprzęcie. Czarnymi strzałkami oznaczono pomiędzy jakimi elementami następuje wymiana danych, na czerwono kontrolę procesora

```
#!/bin/bash

fps=60/1
fmt=YUY2
whd=1920
hhd=1080

wnn=28
hnn=28

wbox=224
hbox=224

src="usbcam"

sink=hdmi
plane=30

export GST_DEBUG="sdxnet:6"

gst-launch-1.0 \
  xlnxvideosrc src-type="${src}" io-mode="dmabuf" ! \
  videoconvert ! \
  videocrop top=428 bottom=428 left=848 right=848 ! \
  videoscale ! \
  "video/x-raw, width=${wnn}, height=${hnn}, format=${fmt}" ! \
  sdxnet ! \
  videoscale ! \
  "video/x-raw, width=${wbox}, height=${hbox}" ! \
  videobox autocrop=true ! \
  "video/x-raw, width=${whd}, height=${hhd}" ! \
  xlnxvideosink sink-type=${sink} \
    plane-id=${plane} fullscreen-overlay=true
```

Listing 4: Skrypt uruchamiający cały projekt

3 Wyniki i dyskusja

W tym rozdziale przedstawione są wyniki prezentujące dokładność predykcji dokonywanych przez model na różnych etapach implementacji projektu zaczynając od testach w Pythonie, a kończąc na sprawdzeniu działania w pełni zaimplementowanego projektu.

3.1 Ewaulacja modelu

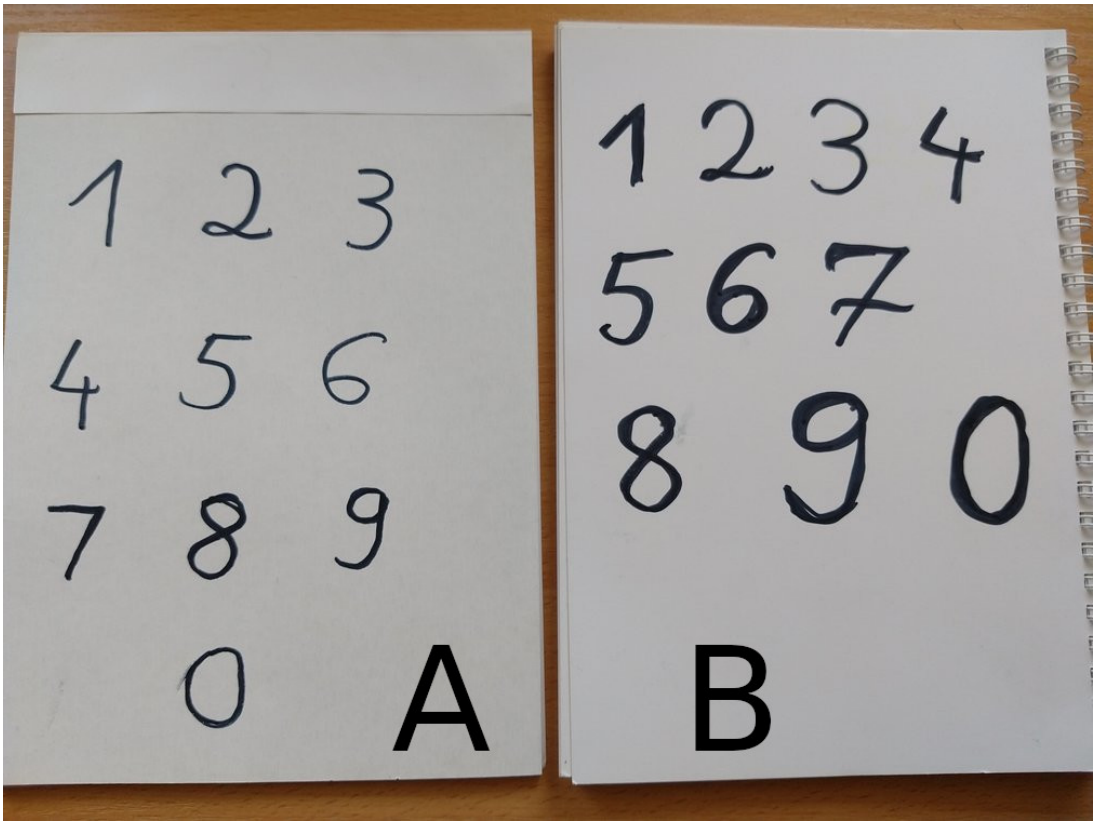
Do testów w Pythonie użyto 10000 próbek z bazy MNIST, którymi nie wytrenowano modelu. Osiągnięta dokładność wynosi 92,22%.

3.2 Symulacja

Kolejne testy przeprowadzono po konwersji modelu sieci do kodu C++ przez hls4ml. Danymi testowymi były te same próbki z bazy MNIST. Liczba dobrze sklasyfikowanych cyfr spadła — osiągnięto 90,52% dokładności.

3.3 Dane rzeczywiste

Kompletny projekt przetestowano na zbiorze liczącym 20 cyfr (patrz rys. 10). Cyfry zapisano odręcznie czarnym markerem na białym papierze. Dla każdej z nich dokonano pomiarów, a uzyskane wyniki zapisano w tabeli 1.



Rysunek 10: Przygotowane zestawy danych testowych

3.3.1 Tabela z wynikami

Dla każdej z cyfr wykonano kilkadziesiąt odczytów. W tabeli 1 podane są najistotniejsze z nich. Pierwsza kolumna oznacza rzeczywistą cyfrę. W kolumnach „zestaw A" i „zestaw B" podane wyniki dla cyfr z zestawów przedstawionych na rysunku 10. Każda komórka zawiera maksymalną odczytaną wartość prawdopodobieństwa przynależności do danej klasy (lub „nie rozpoznano" jeśli sieć ani razu nie rozpoznała poprawnie cyfry) oraz wszystkie wystąpienia błędnych predykcji wraz z odpowiadającym im największymi odczytami prawdopodobieństwa (o ile takie wystąpiły). Ze względu na użyty typ danych w ostatniej warstwie sieci wyniki otrzymano z dokładnością do 6,25 punkta procentowego.

Cyfra	zestaw A	zestaw B
0	97,5%	97,5% „3" 67,5% „2" 62,5%
1	nie rozpoznano „7" 81,25%	92,5% „7" 62,5%
2	100% „3" 50%	100%
3	100%	100%
4	92,5% „9" 92,5%	do 37,5% różne cyfry do 60%
5	62,5% „2" 43,75% „3" 37,5%	87,5% „6" 56,25% „3" 31,25%
6	87,5%	100% „2" 62,5%
7	nie rozpoznano „6" 93,75% „2" 62,5%	100%
8	93,75%	93,75% „3" 75% „9" 43,75%
9	nie rozpoznano „3" 93,75% „8" 87,5%	nie rozpoznano „3" 87,5% „2" 56,25%

Tabela 1: Wyniki dla danych rzeczywistych z kamery

3.3.2 Omówienie wyników

Dla 14 spośród 20 cyfr otrzymano wyniki z prawdopodobieństwem wynoszącym co najmniej 87,5%. Dla cyfr „2" i „3" uzyskano wiele odczytów ze stuprocentowym prawdopodobieństwem dla obydwu zestawów. Cyfry „0", „6" i „8" również były w większości

odczytów poprawnie kwalifikowane. Małe problemy nastąpiły przy próbach odczytu cyfr „1", „4" i „7", w których sieć nie była w stanie poprawnie określić klasy, natomiast poprawnie odczytywała tę cyfrę z drugiego zestawu. Jedynie cyfra „9" nie została w żadnym z przypadków rozpoznana, ponadto błędnie odczytywana jako cyfra „3" z obu zestawów.

Każda próba pomiaru cyfry dawała wiele wyników w zależności od położenia cyfry na obrazie, jej grubości, kąta pod jakim jest nagrywana. Pomiary starano robić się tak, aby środek cyfry pokrywał się ze środkiem obrazu, a także żeby cyfra nie stykała się z jego krawędzią. Dla wszystkich próbek sprawdzano różne odległości, przesunięcia cyfry względem środka obrazu i kąty. Zauważono, że przy kilku próbkach, szczególnie dla cyfry „5", najmniejsze poruszenie kamery wynikające z niestabilności ręki wpływało w znacznym stopniu na predykcję.

3.4 Dyskusja

Na każdym z etapów implementacji można zauważyć spadek dokładności sieci. Po przetłumaczeniu modelu sieci przez hls4ml jest to głównie spowodowane zmniejszeniem precyzji danych na jakich sieć dokonuje obliczeń. Na ostatnim etapie spadek związany jest z możliwymi szumami obrazu, niewystarczająco dobrym nakierowaniem kamery na cyfrę, złym oświetleniem lub innymi zewnętrznymi przyczynami. Sieć o tej stosunkowo małej architekturze może mieć problem z rozpoznawaniem cyfr o takim kształcie i położeniu na obrazie, z którym nie miała do czynienia w trakcie trenowania. Możliwym rozwiązaniem tych problemów mogłoby być zastosowanie sieci splotowej (konwolucyjnej), na przykład LeNet-5, która nie tylko osiąga zdecydowanie wyższy wynik w trakcie ewaluacji ale również lepiej radzi sobie z generalizowaniem kształtów znajdujących się na obrazie. W przeprowadzonym teście architektury LeNet-5 osiągnięto wynik 99,18% dobrze rozpoznanych cyfr, jednak z wcześniej wspomnianych powodów nie wykorzystano takiej architektury.

Podsumowanie

Celem pracy był projekt, który pozwala na rozpoznawanie cyfr w czasie rzeczywistym, korzystając z modelu sieci neuronowej zaimplementowanej na układzie FPGA. Całość można określić jako GStreamer pipeline, spośród którego elementów można wyróżnić wtyczkę do rozpoznawania cyfr, korzystającą z sieci neuronowej, wtyczki odpowiadające za wstępne przetworzenie obrazu oraz te, które umożliwiają wyświetlenie obrazu na urządzeniu zewnętrznym.

W rozdziale 1 została przedstawiona teoria przydatna do zrozumienia projektu. Opisana została architektura FPGA w rozdziale 1.1, informacje o przetwarzaniu obrazu (1.2) w tym opis użytego formatu pikseli oraz krótki opis jak działają sieci neuronowe (1.3).

Przedstawiono również sam projekt zaczynając od ogólnego zarysu (2.1), pokazania platformy z której korzystano (2.2) i prezentacji architektury sieci (2.3). Następnie przybliżono frameworki hls4ml (2.4) i GStreamer (2.5). Został opisany i przedstawiony również sam GStreamer pipeline w rozdziałach 2.5.1–2.5.8.

Przybliżony został proces implementacji modelu sieci na FPGA w rozdziale 2.6, gdzie opisano jak dostosować kod będący wynikiem działania hls4ml do poprawnego działania z resztą projektu oraz w jaki sposób utworzyć z niej bibliotekę dzieloną, z której można korzystać w projektach SDSoC.

Ostatnie podrozdziały rozdziału 2 dotyczą wykorzystania utworzonej biblioteki dzielonej w implementacji wtyczki GStreamer. Proces tworzenia został podzielony na dwie części (neuralnet, patrz 2.7.1 i gstsdxnet — 2.7.2), aby w logiczny sposób oddzielić od siebie potrzebne funkcjonalności.

Na samym końcu zaprezentowano otrzymane wyniki na różnych etapach implementacji sieci neuronowej: jako model w Pythonie, w trakcie symulacji komputerowej przed syntezą oraz po pełnej implementacji, wykorzystując własne dane testowe (patrz rys. 10).

W implementacji projektu jest wiele rzeczy, które można poprawić lub w jakiś sposób ulepszyć. Wśród nich jest wykorzystanie innej architektury sieci, aby otrzymać większą skuteczność rozpoznawania cyfr, przeniesienie części odpowiedzialnej za wstępne przetworzenie obrazu do funkcji akcelerowanych sprzętowo w celu poprawienia wydajności, inny sposób prezentacji wyniku, na przykład na wyświetlanym obrazie, kończąc na samym sposobie informowania użytkownika jaka część obrazu będzie klasyfikowana przez sieć — wizualne oznaczenie fragmentu obrazu zamiast wykadrowania reszty. Na wszystkie wyżej wymienione poprawy zabrakło czasu, możliwości sprzętowych lub wiedzy dotyczącej platformy reVISION.

W repozytorium BitBucket[6] udostępnione zostały pliki źródłowe pozwalając na odtworzenie projektu. Udostępniono również stosowne instrukcje.

Literatura

[1] I. Kuon, R. Tessier i J. Rose, *FPGA Architecture: Survey and Challenges, Foundations and Trends® in Electronics Design Automation.* Tom 2, 2007

[2] Xilinx, platforma reVISION w repozytorium GitHub https://github.com/Xilinx/Embedded-Reference-Platforms-User-Guide/blob/2018.3/README.md (dostęp 18.08.2020)

[3] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, *Gradient-Based Learning Applied to Document Recognition.* 1998

[4] Y. LeCun, C. Cortes, C. Burges, Baza danych MNIST http://yann.lecun.com/exdb/mnist/ (dostęp 18.08.2020)

[5] Xilinx, *Zynq UltraScale+ MPSoC Base Targeted Reference Design, User Guide.* UG1221 (v2018.3), 2018

[6] *Repozytorium NN-Revision na BitBucket z plikami źródłowymi* https://bitbucket.org/fpgafais/nn-revision/src/master/ (dostęp 20.08.2020)