



AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
Wydział Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej

Praca dyplomowa

Detekcja obiektów z wykorzystaniem głębokich sieci neuronowych zrealizowana na wbudowanej platformie obliczeniowej

Object detection using deep neural networks implemented on an embedded computing platform

Autor: *Michał Machura*
Kierunek studiów: *Automatyka i Robotyka*
Opiekun pracy: *dr inż. Tomasz Kryjak*

Kraków, 2021

Streszczenie

W pracy przedstawiono pełen proces projektowania sprzętowej implementacji systemu detekcji opartego o głębokie sieci neuronowe na potrzeby konkursu *2021 DAC SDC*. Celem było zaproponowanie rozwiązania o dużej dokładności, wysokiej przepustowości oraz o niewielkim zużyciu energii. Na wstępnie omówiono wymagania oraz założenia konkursu, docelową platformę – *Avnet Ultra96 V2*, a także dokonano przeglądu dostępnych rozwiązań klasycznych oraz opartych o sieci neuronowe, w szczególności rozwiązań energooszczędnich. Następnie przedstawiono proces projektowania architektury sieci, wraz z treningiem i kwantyzacją oraz opisano implementację akceleratora sprzętowego, jak również próby optymalizacji zużycia energii. W rezultacie uzyskano dokładność detekcji 0.7015 mierzoną współczynnikiem *IoU*, przy przepustowości *72.7 fps* oraz zużyciu energii *2739J* dla 52500 obrazów.

Słowa kluczowe: głębokie konwolucyjne sieci neuronowe, system wizyjny, detekcja obiektów, kwantyzowane sieci neuronowe, akcelerator sprzętowy

Abstract

This work presents the full process of designing hardware implementation of a detecton system based on deep neural networks for the *2021 DAC SDC* competition. The aim was to design a solution with high accuracy, high throughput and low energy consumption. The requirements and assumptions of the competition and the target platform - *Avnet Ultra96 V2* were initially discussed, as well as the available classic solutions and solutions based on neural networks, especially energy-saving solutions, were reviewed. Next, the process of designing the network architecture is presented, along with training and quantization, and the implementation of a hardware accelerator as well as attempts to optimize energy consumption are described. The obtained result was the detection accuracy of 0.7015, measured with the *IoU* coefficient, with a throughput of *72.7 fps* and energy consumption of *2739J* for 52500 images.

Keywords: Deep Convolutional Neural Networks, vision system, object detection, Quantized Neural Networks, hardware accelerator

Spis treści

1. Wprowadzenie	9
1.1. Cel pracy	10
1.2. Zawartość pracy	10
2. Analiza wymagań konkursu DAC SDC 2021	11
2.1. Wymagania	11
2.2. Platforma sprzętowa	13
2.3. Narzędzia	14
3. Detekcja obiektów	17
3.1. Zadanie detekcji	17
3.2. Metody klasyczne	19
3.3. Sieci neuronowe	21
3.3.1. Wstęp do sztucznych sieci neuronowych	21
3.3.2. R-CNN	24
3.3.3. SSD	25
3.3.4. YOLO	28
3.4. Przegląd rozwiązań energooszczędnych	31
3.4.1. Konwolucja separowalna	31
3.4.2. Sieci kwantyzowane	32
3.4.3. Sieci binarne	34
3.4.4. Ultra_net	34
3.4.5. SkyNet	35
4. Badania wstępne	37
4.1. Architektura wstępna	37
4.2. Architektura rozgałęziona	39
4.3. Architektura Resnet18	40
4.4. LittleNet	42

5. Implementacja programowa oraz sprzętowa wybranej architektury sieci	49
5.1. Implementacja programowa	49
5.2. Implementacja sprzętowa	50
5.2.1. Warstwa wejściowa	52
5.2.2. Warstwa <i>depthwise (DW)</i>	54
5.2.3. Warstwa <i>pointwise (PW)</i>	55
5.2.4. Sterowanie akceleracją	56
5.3. Podsumowanie	58
6. Ewaluacja sprzętowa i optymalizacja	59
6.1. Ewaluacja	59
6.2. Optymalizacja	60
7. Podsumowanie	63
A. Dodatek A	69

1. Wprowadzenie

Rozwój technologii takich jak pojazdy autonomiczne, roboty humanoidalne, systemy nadzorcze czy systemy automatycznej kontroli wymagają stosowania szeroko rozumianych systemów percepji o wysokiej skuteczności. Możliwe jest to miedzy innymi poprzez realizację zadania detekcji wybranych typów obiektów na podstawie danych z czujników takich jak kamery, radar czy *LiDAR*. Otrzymane w rezultacie parametry obiektu, takie jak położenie czy wymiary, pozwalają na podjęcie stosownej akcji np. ominiecie przeszkody czy dotarcie do wybranego celu. We wspomnianych systemach, często wymagane jest też działanie w czasie rzeczywistym, przy ograniczonym budżecie energetycznym.

Rozwiązańami cechującymi się wysoką skutecznością są algorytmy sztucznej inteligencji, a w szczególności sieci neuronowe. Jednak charakteryzują się on również znaczną złożonością obliczeniową. W przypadku systemów stacjonarnych problem ten można rozwiązać poprzez zastosowanie urządzeń zapewniających dużą moc obliczeniową, takich jak wydajne karty graficzne (GPU, ang. *Graphics Processing Unit*). Jednak takie podejście nie zawsze jest możliwe dla zastosowań mobilnych – pojazdów autonomicznych, robotów, dronów – gdzie dostępny budżet energetyczny jest ograniczony. W takich sytuacjach jedną z możliwości jest zastosowanie układów *FPGA* (ang. *field-programmable gate array*) czy *SoC* (ang. *System on Chip*) oraz tzw. sieci kwantyzowanych (ang. *Quantized Neural Networks*). Rozwiązania te pozwalają osiągać zarówno wysoką skuteczność, niewielkie zużycie energii jak i wysoką przepustowość pozwalającą na zastosowanie w systemach czasu rzeczywistego. Zastosowanie nie ogranicza się jedynie do urządzeń mobilnych. Możliwe jest również stosowanie w serwerowniach pozwalając nie tylko przyspieszyć obliczenia, lecz również zredukować zużycie energii.

Oba wspomniane zagadnienia detekcji oraz akceleracji są na tyle istotne, iż społeczność naukowa organizuje konkursy, takie jak *DAC SDC* (ang. *2021 Design automation Conference System Design Contest*) czy *LPVC* (ang. *Low Power Vision Challenge*).

1.1. Cel pracy

Celem niniejszej pracy jest implementacja głębokiej sieci neuronowej na wbudowanej platformie obliczeniowej – układzie *Zynq UltraScale+ MPSoC*. System jest opracowywany na potrzeby konkursu *2021 DAC SDC* (ang. *2021 Design automation Conference System Design Contest*). Zadaniem jest detekcja obiektów na zdjęciach zarejestrowanych przez drona. Wymagane jest osiągnięcie wysokiej przepustowości oraz skuteczności, przy jednoczesnym niewielkim zużyciu energii.

1.2. Zawartość pracy

W pracy przedstawiono zagadnienia związane z uczeniem sieci neuronowych, a także ich implementacją. Z uwagi na uczestnictwo w konkursie w rozdziale 2 przedstawiono szczegółowe wymagania i założenia dotyczące m.in. implementacji i oceny rozwiązania. Przedstawiono także opis docelowej platformy obliczeniowej oraz omówiono narzędzia wspomagające implementację sprzętową sieci neuronowych. W rozdziale 3 zaprezentowano przegląd metod detekcji klasycznych oraz opartych o sieci neuronowe, w szczególności rozwiązania energooszczędne. Badania nad architekturą oraz proces uczenia i kwantyzacji zostały opisane w rozdziale 4. Rozdział 5 przedstawia implementację opracowanej architektury zarówno programową jak i sprzętową. Uzyskane rezultaty oraz przeprowadzoną optymalizację omówiono w rozdziale 6. W rozdziale 7 podsumowano pracę, przedstawiono wnioski z niej płynące oraz możliwe kontynuacje prac.

2. Analiza wymagań konkursu DAC SDC 2021

Opracowywany system powstaje na potrzeby konkursu *Design Automation Conference System Design Contest 2021* (DAC SDC). Rozpatrywanym problemem jest detekcja obiektów za pomocą sztucznej sieci neuronowej. Realizacja zadania ma zostać przeprowadzona na wbudowanej platformie obliczeniowej typu *SBC* (ang. *Single Board Computer*) wyposażonej w logikę rekonfigurowalną. W niniejszym rozdziale zostaną przedstawione założenia oraz wymagania stawiane wobec opracowywanego systemu detekcji, opisana docelowa platforma obliczeniowa, a także dostępne narzędzia wspomagające docelową implementację.

2.1. Wymagania

Opracowywany system powstaje na potrzeby konkursu, zatem rozwiążanie powinno spełniać stawiane założenia oraz wymagania dotyczące realizowanego zadania i jego implementacji. Zadaniem jest detekcja pojedynczych obiektów za pomocą sieci neuronowych na sekwencjach zarejestrowanych przez drona. Organizatorzy dostarczają zbiór treningowy składający się z obrazów o wymiarach 640 pikseli szerokości na 360 pikseli wysokości. Zbiór składa się z ponad 90 tysięcy obrazów, wchodzących w skład 95 sekwencji podzielonych na 12 kategorii. Na rysunku 2.1 przedstawiono przykładowe obrazy znajdujące się w zbiorze uczącym wraz z zaznaczonymi obiektami.

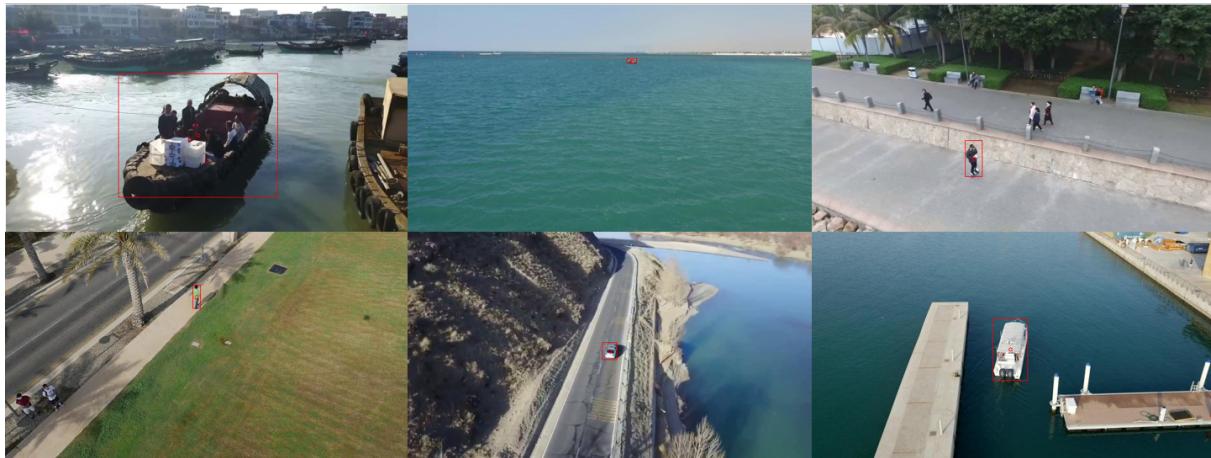
Samo rozwiążanie poddawane jest ocenie danej wzorem (2.4). Uzyskanie wyższej wartości skutkuje uzyskaniem wyższego miejsca w rankingu, z czego wynika, iż funkcję *Score* należy maksymalizować.

$$E_{Score} = \log_2(E) \quad (2.1)$$

$$IoU_{Score} = \max(0.1, \text{ReLU}(1 - 5\text{ReLU}(0.7 - IoU))) \quad (2.2)$$

$$FPS_{Score} = \text{ReLU}(1 - \text{ReLU}(1 - \frac{FPS}{30})) \quad (2.3)$$

$$Score = \frac{10^2}{E_{Score}} IoU_{Score} FPS_{Score} \quad (2.4)$$



Rys. 2.1. Przykładowe obrazy zbioru uczącego wraz z zaznaczonymi obiektami detekcji. Źródło [1].

Ocenie poddawana jest zarówno dokładność detekcji mierzona współczynnikiem IoU (ang. *Intersection over Union*), lecz również przepustowość FPS mierzona poprzez liczbę przetworzonych obrazów na sekundę. Sam współczynnik IoU definiowany jest jako stosunek powierzchni (liczby pikseli) części wspólnej obszaru referencyjnego X_{ref} oraz obszaru wyznaczonego X_{pred} do łącznej powierzchni obu obszarów. Wyrażone to zostało za pomocą wzoru (2.5).

$$IoU = \frac{|X_{ref} \cap X_{pred}|}{|X_{ref} \cup X_{pred}|} \quad (2.5)$$

Ponadto mierzona jest również całkowita energia E zużyta przez układ – wyznaczana jako iloczynu czasu przetwarzania oraz średniej mocy układu (w czasie przetwarzania). Chwilowe wartości mocy są odczytywane z regulatora mocy z częstotliwością próbkowania $20Hz$. Zauważno pomiar energii oraz czasu musi być realizowany przez opracowaną aplikację. Wstępna analiza pozwala stwierdzić, iż uzyskanie wartości IoU oraz FPS poniżej wartości progowych skutkuje zastosowanie kary, natomiast zwiększanie tych wartości powyżej wartości progowych nie przynosi dodatkowych korzyści (przynajmniej w sposób bezpośredni).

Wśród wymagań znajdują się również informacje odnośnie dostarczenia wymaganych plików. Aplikacja sterująca ma być zrealizowana w postaci notatnika *Jupyter*. Dodatkowo należy dostarczyć również pliki **.hwh* (plik opisu diagramu blokowego logiki programowej) oraz **.bit* (plik konfiguracji logiki programowej) Ponadto należy dołączyć inne niezbędne do prawnego działania aplikacji pliki. Finalne rozwiązanie musi być dostępne w formie otwarto-źródłowej.

2.2. Platforma sprzętowa

Ewaluacja wytrenowanej sieci neuronowej jest przeprowadzana na platformie obliczeniowej *Avnet Ultra96-V2* [2]. Jest to płytka rozwojowa wyposażona w układ *Xilinx Zynq UltraScale+ MPSoC ZU3EG A484*, 2 GB pamięci LPDDR4, slot kart microSD, moduł Wi-Fi, a także porty USB 3.0, porty IO oraz port mini-display. Układ działa pod kontrolą systemu operacyjnego *PetaLinux* wraz z uruchomionym serwerem *Jupyter* (pochodzących z dystrybucji projektu *Pynq*[3]). Sam układ *Xilinx Zynq UltraScale+ MPSoC ZU3EG A484* jest układem typu SoC. Zawiera czterordzeniowy procesor ARM Cortex-A53 MPCore, dwurdzeniowy procesor Arm Cortex-R5F MPCore oraz procesor graficzny Mali-400 MP2 [4]. Ponadto układ wyposażony jest w logikę programowalną połączoną z systemem procesorowym magistralą AXI. Logika programowalna zawiera: 154 tys. *System Logic Cells (SLC)*, 141 tys. przerzutników (ang. *Flip-Flops, FF*) oraz 71 tys. *LUT* (ang. Look-Up Table) zawartych w blokach *CLB* (ang. *Configurable Logic Block*), 360 *DSP slice*, 1.8 MB pamięci *distributed RAM*, a także 7.6 MB pamięci zawartej w 216 blokach *BRAM*.

SLC jest to miara określająca możliwości czy liczbę zasobów umożliwiającą porównanie układów logiki programowalnej. Jeden *SLC* stanowi 1 *LUT* pozwalający na realizację 6-o elementowej funkcji logicznej, 2 przerzutniki oraz dodatkowe elementy logiczne t.j. multiplekser.

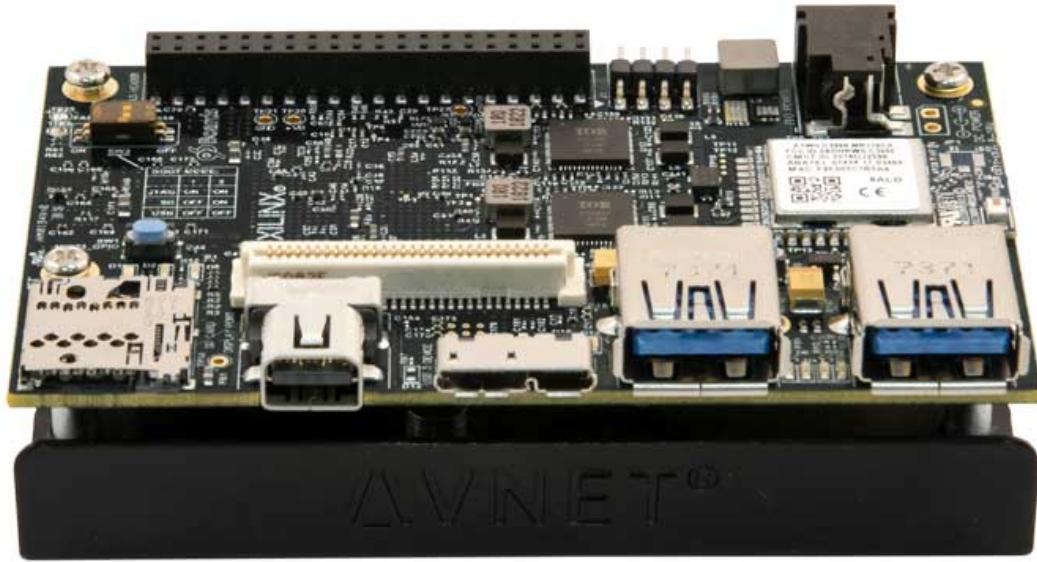
Bloki *CLB* [5] dzielą się na dwa typy:

- *SLICEL* – jest zbudowany z 8 *LUT*, 16 przerzutników, 8-bitowego łańcucha przeniesień (ang. *carry chain*) oraz 7 multiplekserów. Elementy *LUT* możliwe są do konfiguracji pozwalającej na realizację dowolnej funkcji logicznej o 7-u, 8-u lub 9-u wejściach lub wybranych funkcji o 55-u wejściach. Uzyskane rezultaty mogą zostać zapamiętane w pamięci przerzutników (FF – ang. *Flip Flop*).
- *SLICEM* – stanowi rozbudowany *SLICEL* o dodatkowe elementy pamięci możliwe do konfiguracji w postaci 64 bitowej pamięci *Distributed RAM* lub 32 bitowego rejestru przesuwnego.

DSP (ang. *Digital Signal Processor*) [6] stanowią elementy obliczeniowe pozwalające na realizację, operacji arytmetycznych czy logicznych. Charakteryzują się niewielkim zużyciem energii oraz dużą elastycznością. Możliwe są konfiguracje m.in. jako sumator, mnożarka czy nawet akumulator iloczynów sygnałów wejściowych.

Blok *BRAM* (ang. *Block RAM*) [bram] stanowi 36Kbit pamięci o 2 portach do zapisu i 2 portach do odczytu. Możliwa jest konfiguracja jako 2 niezależne pamięci do 18Kbit, a także jako kolejka *FIFO*. Dodatkowo dostępny jest tryb uśpienia pamięci pozwalając na dynamiczne zarządzanie mocą.

Rysunek 2.2 przedstawia platformę docelową.



Rys. 2.2. Płytki Avnet Ultra96 V2. Źródło [2].

2.3. Narzędzia

Uzyskanie przepustowości 30fps , pozwalającej na osiągnięcie wysokiej oceny rozwiązania, może być trudne lub niemożliwe przy użyciu jedynie części procesorowej nawet dla modeli o niskiej złożoności obliczeniowej. Wymagana jest tutaj dodatkowa akceleracja wykorzystująca logikę programowalną. W tym celu możliwe jest wykorzystanie istniejących narzędzi wspomagających sprzętową implementację sieci neuronowych, a także procesu ich uczenia.

Uczenie sieci z przeznaczeniem do sprzętowej akceleracji zazwyczaj wiąże się z wytrenowaniem modelu zmiennoprzecinkowego, następnie poddaniu go kwantyzacji oraz finalnie implementacji sprzętowej. Etap wytrenowania modelu zmiennoprzecinkowego może zostać przeprowadzony z użyciem dowolnych narzędzi. Korzystając z formatu nie wspieranego wymagana może być zmiana reprezentacji. Następnym etapem jest kwantyzacja wag oraz wyników pośrednich sieci, gdzie następuje przybliżenie liczb zmiennoprzecinkowych liczbami stałoprzecinkowymi lub binarnymi. Zazwyczaj wyniki sieci kwantyzowanej różnią się od wyników modelu zmiennoprzecinkowego, przez co wymagane jest wznowienie procesu uczenia. Istnieje też możliwość pominięcia początkowego treningu modelu zmiennoprzecinkowego. Model kwantyzowany możliwy jest do sprzętowej implementacji z wykorzystaniem wybranych narzędzi.

Poniżej wymieniono wybrane narzędzia wspomagające sprzętową implementację sieci neuronowych

- *Brevitas*[7] – biblioteka dla framework'a *PyTorch* [8], przeznaczona do wspomagania kwantyzacji i treningu QNN, rozwijana przez *Xilinx*. Umożliwia trenowanie modeli kwantyzowanych, zmiennoprzecinkowych oraz mieszanych. Proces kwantyzacji wymaga zbudowania modelu z predefiniowanych warstw przystosowanych do kwantyzacji. Metoda kwantyzacji jest podawana jako parametr inicjalizacyjny, za pośrednictwem nazwy klasy *kwantyzatora* – predefiniowanej lub własnej implementacji. Pozwala to na zastosowanie dowolnej techniki kwantyzacji, czyniąc narzędzie bardzo wygodnym – w szczególności w środowisku naukowym.
- *FINN*[9] – framework rozwijany przez *Xilinx* pozwalający na implementację sprzętową QNN w architekturze przetwarzania potokowego. Wymaga użycia modelu kwantyzowanego w formacie *ONNX* (ang. *Open Neural Network Exchange*) uzyskanego np. z *Brevitas*. Narzędzie dokonuje implementacji za pośrednictwem *HLS*. Oprócz opisu sprzętowego generowana jest również część programowa pozwalająca na sterowanie procesem akceleracji. W pewnym stopniu możliwe jest również uproszczenie modelu np. łącząc kolejne operacje z użyciem stałych.
- *QKeras*[10] – biblioteka dla framework'a *TensorFlow*[11] pozwalająca na kwantyzację i trening QNN. Narzędzie mniej elastyczne niż *Brevitas*, lecz posiadające prosty i przyjazny interfejs. Możliwa jest również automatyczna konwersja modelu zmiennoprzecinkowego do modelu zapisanego z użyciem kwantyzowanych odpowiedników.
- *hls4ml*[12] – biblioteka dokonująca konwersji modelu QNN (*QKeras*) do zapisu z użyciem języka *HLS*. Posiada prosty interfejs. Możliwe jest m.in. ustalenie parametrów ponownego użycia zasobów, czy optymalizacja wybranych elementów modelu.
- *Vitis AI*[13] – środowisko pozwalające na przeprowadzenie kwantyzacji, optymalizacji architektury sieci oraz sterowania procesem akceleracji poprzez stosowne biblioteki C++/Python. Jest to środowisko rozwijane przez firmę *Xilinx*. Wykorzystywana jest tutaj akceleracja z wykorzystaniem generycznego, konfigurowalnego koprocesora DPU (ang. *Deep Processing Unit*).

Na etapie analizy problemu uwzględniono zarówno wymagania konkursowe, dostępną platformę sprzętową, a także narzędzia wspomagające sprzętową implementację. Głównym celem jest implementacja szybkiego i energooszczędnego rozwiązania, pozwalającego na dokładną detekcję pojedynczych obiektów. Zadanie jest realizowane w oparciu o sprzętową akcelerację wykorzystującą logikę programowalną płytki *Avnet Ultra96 V2*. Dostępnych jest wiele narzędzi usprawniających proces kwantyzacji, jak również implementacji sprzętowej. W edycji 2021 konkursu zmieniono w stosunku do lat poprzednich funkcję oceny (2.4). Liniowa zależność

zużycia energii została zastąpiona logarytmiczną (2.1). Zmniejszono wagę związaną z przepustowością oraz dokładnością, a także zastosowano dodatkową “zewnętrzną” funkcję *ReLU*. Ponadto funkcja oceny jakości (2.2) dla $iou < 0.52$ przyjmuje wartość 0.1. W poprzednich edycjach dla $iou < 0.5$ funkcja “zerowała” ostateczna ocenę. Wymóg udostępnienia rozwiązania w formie otwarto-źródłowej pozwala na przegląd rozwiązań z edycji poprzedzających, a tym samym na ich dalszy rozwój.

3. Detekcja obiektów

W poprzednim rozdziale przedstawiono wymagania konkursu DAC SDC 2021 oraz jego główne zadanie – detekcję obiektów. W niniejszym rozdziale przedstawione zostaną metody detekcji – klasyczne oraz bazujące na sieciach neuronowych, a także architektury sieci stosowane w rozwiązańach energoszczędnnych, w tym sieci kwantyzowane (ang. *Quantized Neural Networks, QNN*) oraz binarne (ang. *Binary Neural Networks, BNN*).

3.1. Zadanie detekcji

Analizę rozważanego zadania należy rozpocząć od jego sformułowania. Zakładając, iż dany jest obraz $F(y, x)$ o szerokości W i wysokości H określony na zbiorze

$$D = \{(y, x) \in \mathbb{N}^2 : 0 \leq x < W \wedge 0 \leq y < H\} \quad (3.1)$$

stanowiącym zbiór pikseli obrazu (par współrzędnych) (y, x) oraz $P \subseteq D$ oznaczające zbiór pikseli poszukiwanego obiektu, należy znaleźć x_{min} , x_{max} , y_{min} oraz y_{max} takie, że spełnione są zależności (3.2)-(3.5). Na rysunku 3.1 przedstawiono graficzną interpretację poniższych równań.

$$x_{min} = \min_{j \in \mathbb{N} \wedge j < W} \{j : \exists i \in \mathbb{N} \wedge i < H \wedge (i, j) \in P\} \quad (3.2)$$

$$x_{max} = \max_{j \in \mathbb{N} \wedge j < W} \{j : \exists i \in \mathbb{N} \wedge i < H \wedge (i, j) \in P\} \quad (3.3)$$

$$y_{min} = \min_{i \in \mathbb{N} \wedge i < H} \{i : \exists j \in \mathbb{N} \wedge j < W \wedge (i, j) \in P\} \quad (3.4)$$

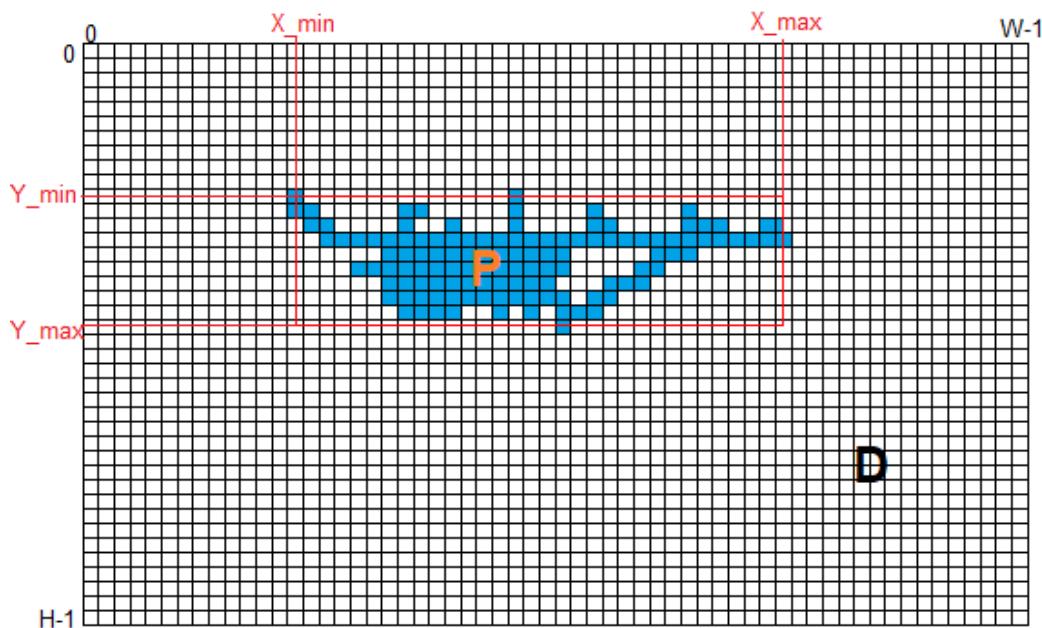
$$y_{max} = \max_{i \in \mathbb{N} \wedge i < H} \{i : \exists j \in \mathbb{N} \wedge j < W \wedge (i, j) \in P\} \quad (3.5)$$

Innymi słowy należy znaleźć parametry:

$$bbox = (x_{min}, x_{max}, y_{min}, y_{max}) \quad (3.6)$$

najmniejszego prostokąta opisanego (ang. *bounding box*) na poszukiwanym obiekcie. Możliwa jest również inna reprezentacja

$$bbox = (x_c, y_c, w, h) \quad (3.7)$$



Rys. 3.1. Graficzna interpretacja zadania detekcji.

gdzie kolejne zmienne to współrzędne środka oraz wymiary obiektu. Obie reprezentacje związane są poprzez równania (3.8)-(3.11).

$$x_c = \frac{x_{min} + x_{max}}{2} \quad (3.8)$$

$$y_c = \frac{y_{min} + y_{max}}{2} \quad (3.9)$$

$$w = x_{max} - x_{min} \quad (3.10)$$

$$h = y_{max} - y_{min} \quad (3.11)$$

Przedstawione sformułowanie zadania tyczy się detekcji pojedynczego obiektu z założeniem, iż obiekt znajduje się na obrazie. W przypadku ogólnym należy znaleźć n bounding box, przy czym n jest zmienne i zależne od rozpatrywanego obrazu. Bardzo często zadanie detekcji łączone z zadaniem klasyfikacji, wówczas oprócz położenia i wymiarów obiektu należy podać również jego typ/klasę.

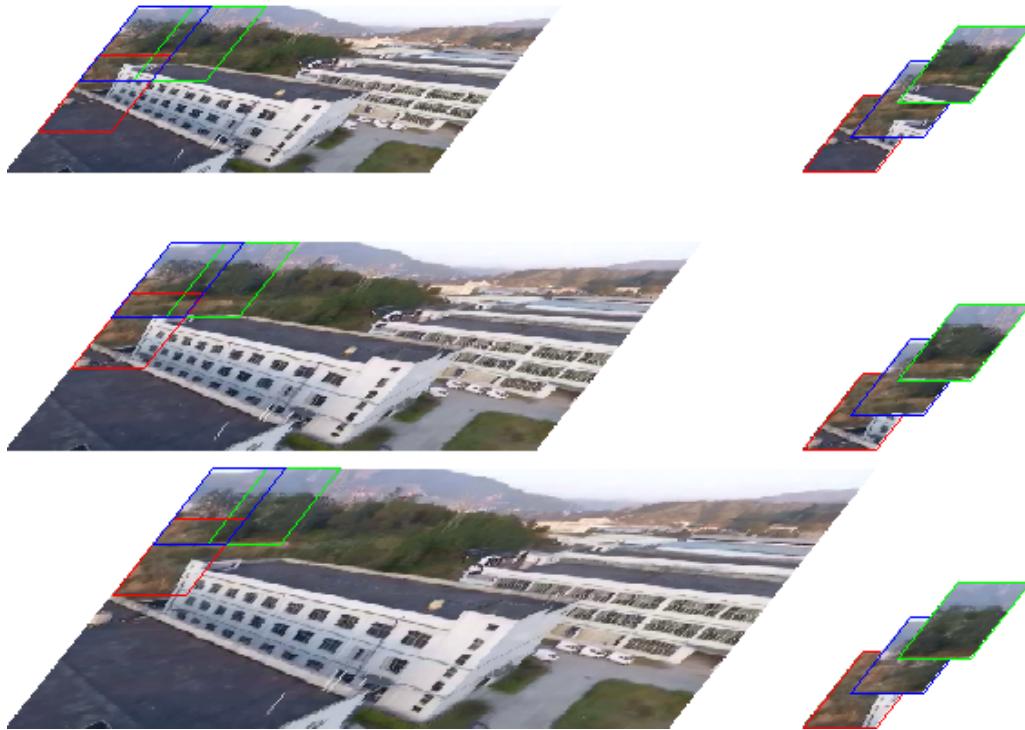
3.2. Metody klasyczne

W prostych przypadkach zadanie detekcji może sprowadzić się do znalezienia wymiarów obiektów na obrazie binarnym. Sytuacja tak może mieć miejsce, gdy poszukiwany obiekt znacznie odróżnia się od tła. Wówczas możliwe jest wskazanie pikseli należących do rozpatrywanego obiektu, poprzez np. binaryzację na podstawie wybranych składowych przestrzeni barwnej. Możliwe są również rozwiązania wykorzystujące technikę okna przesuwnego, ekstrakcji wybranych cech wraz z zadaniem klasyfikacji. Wówczas mamy do czynienia z tzw. metodami klasycznymi. Znalezienie parametrów $bbox$ możliwe wówczas jest z użyciem algorytmów wizji komputerowej lub uczenia maszynowego.

Wspomniana technika okna przesuwnego jest realizowana poprzez analizę (klasyfikację) fragmentów obrazu (zwanych oknem) o określonych wymiarach (d_w, d_h). Fragmenty te stanowią otoczenie wybranych punktów obrazu. Wybierane punty stanowią węzły siatki oddalone od siebie o określoną wartość kroku (s_w, s_h). Można to utożsamiać z “przesuwaniem” okna w płaszczyźnie obrazu. W przedstawionej technice rozmiar okna jest stały co powoduje, że obiekty o wymiarach innych, niż rozpatrywane w problemie klasyfikacji mogą nie zostać poprawnie wykryte. Z tego powodu wymagana jest analiza w wielu skalach. Do tego celu wykorzystuje się tzw. piramidę obrazów, gdzie każdy kolejny obraz stanowi przeskalowaną wersję obrazu pierwotnego, który jest poddawany analizie z wykorzystaniem okna przesuwnego. W przypadku wykrycia obiektu w danej skali oraz położeniu okna, następuje przetransformowanie parametrów detekcji do skali oryginalnej. Dokładność algorytmów wykorzystujących okno przesuwne jest zależna od kroku przesunięcia okna, a także liczby rozpatrywanych skali. Na rysunku 3.2 przedstawiono piramidę obrazów oraz technikę okna przesuwnego.

Jednym z często stosowanych podejść jest algorytm *Viola-Jones*[14]. Wykorzystywane jest tutaj okno przesuwne oraz tzw. cechy *Haar'a*. Reprezentują one układy jasności obrazu takie jak krawędzie, linie czy narożniki. Z obszaru definiowanego przez okno przesuwne ekstrahowane są cechy poddawane klasyfikacji metodą *AdaBoost*. Polega to na klasyfikacji z użyciem kaskady tzw. słabych klasyfikatorów. W przypadku, gdy jeden klasyfikator nie stwierdza wykrycia obiektu, pozostałe etapy klasyfikacji mogą zostać pominięte dla rozpatrywanego okna. Obiekt jest uznawany za wykryty jedynie, gdy dla wszystkie klasyfikatory stwierdzą obecność obiektu.

Innym przykładem może być podejście *HOG+SVM*[15] (ang. *Histogram of Oriented Gradients + Support Vector Machine*). Wykorzystywany jest tutaj podział okna detekcji na komórki, w których wyznaczane są histogramy orientacji gradientu. Histogramy są następnie normalizowane wewnątrz bloków składających się z kilku komórek. Uzyskiwany jest wówczas wektor

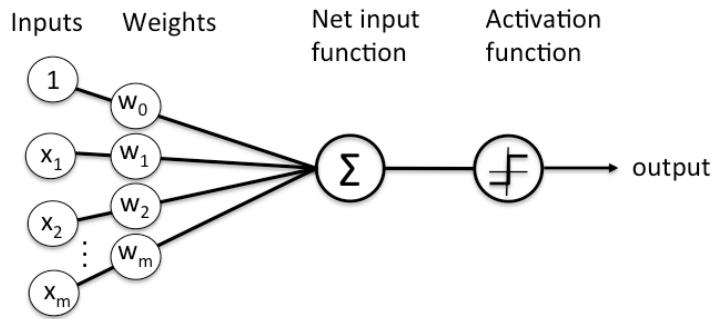


Rys. 3.2. Piramida obrazów wraz z zaznaczonymi wybranymi oknami oraz wyekstrahowane okna przesuwne dla danej skali. Obraz pochodzi ze zbioru uczącego dostarczonego przez organizatorów konkursu [1].

cech opisujący obszar obrazu wyznaczany przez okno przesuwne. Cechy te poddawane są klasyfikacji poprzez SVM, której wynik daje informację o obecności lub braku rozpatrywanego typu obiektu.

Metodą bazującą na cechach *HOG* jest *DPM* (ang. *Deformable Part-based Model*) [16]. Pojęcie to bazuje na detekcji prostych obiektów, które składają się na obiekty o większym stopniu skomplikowania. Wykorzystywane jest tutaj dopasowanie modelów obiektów składowych do mapy cech *HOG* (lub dowolnych innych). Otrzymywane są w ten sposób mapy prawdopodobieństwa wystąpienia danych klas. Następnie wyniki te są analizowane poprzez mieszanie modeli (ang. *Mixture models*) obiektów złożonych. Ponadto metoda ułatwia również detekcję obiektów deformowalnych tzn. mogących zmieniać kształt w perspektywie czasu np. człowiek, u którego położenie, orientacja i kształt części ciała może się zmieniać w czasie.

Wspomniane metody w większości operują na technice okna przesuwnego, co ma wpływ na jakość detekcji. Wykorzystywane są tutaj techniki klasyfikacji na podstawie wybranych cech reprezentujących zawartość okna. Możliwa jest również detekcja poprzez wykorzystanie obrazu binarnego reprezentującego obiekt detekcji. Stosując metody klasyczne zazwyczaj wymagany jest etap przetwarzania wstępного składający się na filtrację, wyrównanie histogramu czy przekształcenia morfologiczne.



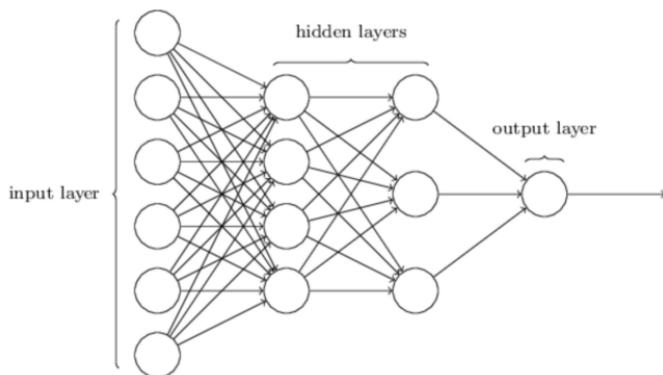
Rys. 3.3. Schemat neuronu. Źródło: [18].

3.3. Sieci neuronowe

Dotychczas wymienione metody wymagały wcześniejszej ekstrakcji określonych cech bądź mapy segmentacji. Stosując sztuczne sieci neuronowe (ang. *Artificial Neural Networks*) możliwe jest połączenie etapów przetwarzania wstępnego, ekstrakcji cech, klasyfikacji jak również określenia położenia i wymiarów obiektu. Pozwala to zarówno na względne uproszczenie systemu detekcji, lecz często również na osiągnięcie lepszych rezultatów. Odbywa się to jednak kosztem zwiększenia złożoności obliczeniowej oraz wymaga złożonego procesu uczenia z użyciem dużych zbiorów danych. W podrozdziale zostaną opisane wybrane metody detekcji wykorzystujące sieci neuronowe, a także wybrane architektury z przeznaczeniem sprzętowej implementacji.

3.3.1. Wstęp do sztucznych sieci neuronowych

Sztuczne sieci neuronowe stanowią algorytmy przetwarzania danych inspirowane biologicznymi procesami zachodzącymi w układzie nerwowym [17]. Wykorzystywane jest tutaj pojęcie tzw. sztucznego neuronu jako odpowiednika biologicznej komórki nerwowej. Neurony stanowią funkcje przetwarzające podane sygnały wejściowe oraz zwracające rezultat w postaci sygnałów wyjściowych. Schemat neuronu przedstawiono na rysunku 3.3. Przedstawiony neuron realizuje operację iloczynu skalarnego pomiędzy wektorem wag w oraz wektorem sygnałów wejściowych x (wliczając stałą 1 stanowiącą sygnał bias). Uzyskany rezultat jest najczęściej poddawany aktywacji wybraną funkcją nieliniową. Większość neuronów posiada parametry zwane wagami. Mogą również zawierać elementy pamięci. Neurony grupuje się w tzw. warstwy (ang. *layers*), które następnie odpowiednio łączy się ze sobą tworząc strukturę sztucznej sieci neuronowej (rysunek 3.4). Dane wejściowe sieci są propagowane przez kolejne warstwy, aż do warstw stanowiących wyjście sieci. Otrzymany rezultat jest zależny od parametrów neuronów poszczególnych warstw.



Rys. 3.4. Schemat sieci neuronowej. Źródło: [19].

Sieć neuronowa posiada zdolność uczenia się tzn. zmian wartości parametrów na podstawie kolejno przedstawianych wzorców uczących wraz z prezentowaniem referencyjnych rezultatów wyjściowych. Proces ten nazywa się uczeniem sieci neuronowej. Do tego celu definiuje się funkcję błędu wyjść, a następnie odpowiednio aktualizuje się wagie z wykorzystaniem tzw. propagacji wstecznej błędu. Jednokrotne przedstawienie zbioru uczącego nazywane jest epoką, w trakcie której następuje aktualizacja wag modelu sieci neuronowej. Ponadto warstwy nie posiadające bezpośredniego połączenia z wyjściem sieci nazywane są warstwami ukrytymi. Sieć która posiada przynajmniej jedną warstwę ukrytą (realizującą operacje nielinowe) nazywana jest głęboką siecią neuronową (ang. *Deep Neural Network*). Sieć głęboka zyskuje zdolność wykształcania cech w procesie uczenia przez warstwy ukryte. Każda kolejna warstwa ukryta wykształca bardziej ogólne cechy. Dokładniejsze omówienie sztucznych sieci neuronowych, nawiązanie do biologicznych odpowiedników, wybrane własności oraz metody uczenia można znaleźć [17] oraz [20]. Opisano tam przede wszystkim warstwy neuronów realizujących połączenia typu “każdy z każdym”, nazywane warstwami gęsto połączonymi (ang. *Densely Connected*) lub w pełni połączonymi (ang. *Fully Connected, FC*). Każdy neuron realizuje tam operacje iloczynu skalarnego wektora sygnałów wejściowych z wektorem wag, niezależnie dla każdego neuronu.

Zastosowanie głębokich sieci neuronowych wykorzystujących warstwy *FC* dla problemów przetwarzania danych tego samego typu o znaczących rozmiarach, takich jak obrazy (a także sygnały dźwiękowe, czy serie danych) skutkuje znaczącym rozbudowaniem architektury, wzrostem złożoności obliczeniowej, a także trudnościami w procesie uczenia. Możliwe jest wówczas zastosowanie tzw. konwolucyjnych sieci neuronowych (ang. *Convolutional Neural Networks, CNN*). Definiują one sposób przetwarzania danych w każdej warstwie w sposób identyczny dla każdego neuronu wyznaczającego daną cechę. Pozwala to na uproszczenie obliczeń, a także na zwiększenie zdolności generalizowania danych. Podstawą jest operacja konwolucji realizowana w przestrzeni cech (lub obrazu dla warstwy wejściowej) z wykorzystaniem maski konwolucji

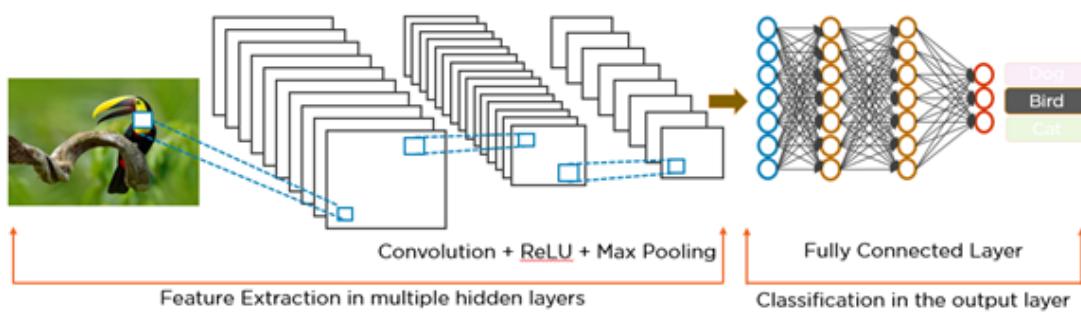
o zadanym rozmiarze otoczenia oraz kształcie (elementy maski mogą zostać “rozsunięte” tzw. *dilation*). Ponadto możliwe jest wykonanie operacji konwolucji z krokiem większym od 1, co skutkuje zmniejszeniem rozmiaru wynikowej mapy cech. Za rodzaj wyznaczanych cech odpowiedzialny jest typ warstwy oraz ewentualne wagi filtrów. Wśród najczęściej stosowanych typów warstw konwolucyjnych wymienić można:

- warstwa konwolucyjna – realizuje standardowe operacje konwolucji Ch_{out} filtrami z wagami dobieranymi w procesie uczenia. Wymiary filtrów zależą od rozmiaru (S_x, S_y) rozpatrywanego kontekstu oraz od liczby kanałów wejściowych Ch_{in} .
- warstwa próbująca – realizowane jest próbkowanie otoczenia definiowanego przez rozmiar maski. Często stosowanymi są warstwy *Max Pooling*, *Min Pooling* czy *Average Pooling* wykonujących kolejno operacje filtru maksymalnego, minimalnego oraz uśredniającego. Najczęściej przyjmowany jest krok konwolucji równy rozmiarowi maski, co skutkuje analizą niepokrywających się obszarów oraz redukcją wymiaru mapy cech.
- warstwa normalizacyjna – wykonywana jest operacja normalizacji kanałów wejściowych z wykorzystaniem średniej kroczej \bar{x}_r oraz wariancji kroczej σ_r , aktualizowanych w trakcie przedstawiania kolejnych wzorców uczących zgrupowanych w tzw. *batch*. Ponadto warstwa posiada dodatkowe wagi przekształcenia aficznego a, b wyznaczane w procesie uczenia. Równanie (3.12) przedstawia wykonywaną operację dla danego kanału o wartościach x oraz dającej rezultat y . Ponadto wykorzystywana jest dodatkowa stała ϵ zapobiegająca dzieleniu przez 0.

$$y = a \frac{x - \bar{x}_r}{\sqrt{\sigma_r + \epsilon}} + b \quad (3.12)$$

- warstwa aktywacji – stanowi zastosowanie wybranej funkcji aktywacji np. *ReLU*, funkcji logistycznej czy *softmax* (*sm*). Często operacje aktywacji uznaje się jako realizowaną bezpośrednio w warstwie konwolucyjnej czy *FC*.

Na rysunku 3.5 przedstawiono przykładową architekturę sieci złożoną z wybranych warstw konwolucyjnych oraz *FC*. Wykorzystanie warstw *FC* sprawia, iż wymagany jest stały rozmiar obrazu wejściowego. W sytuacji gdy architektura sieci składa się tylko z warstw konwolucyjnych rozmiar danych wejściowych jest niezależny od architektury. Architekturę taką nazywa się w pełni konwolucyjną (ang. *Fully Convolutional Neural Network, FCNN*). Warto również wspomnieć o technice transferu wiedzy (ang. *trasfer learning*). Polega ona na inicjalizacji wag wagami pochodząymi od sieci już nauczonej (zazwyczaj na dużym zbiorze danych takim jak np. *ImageNet*).



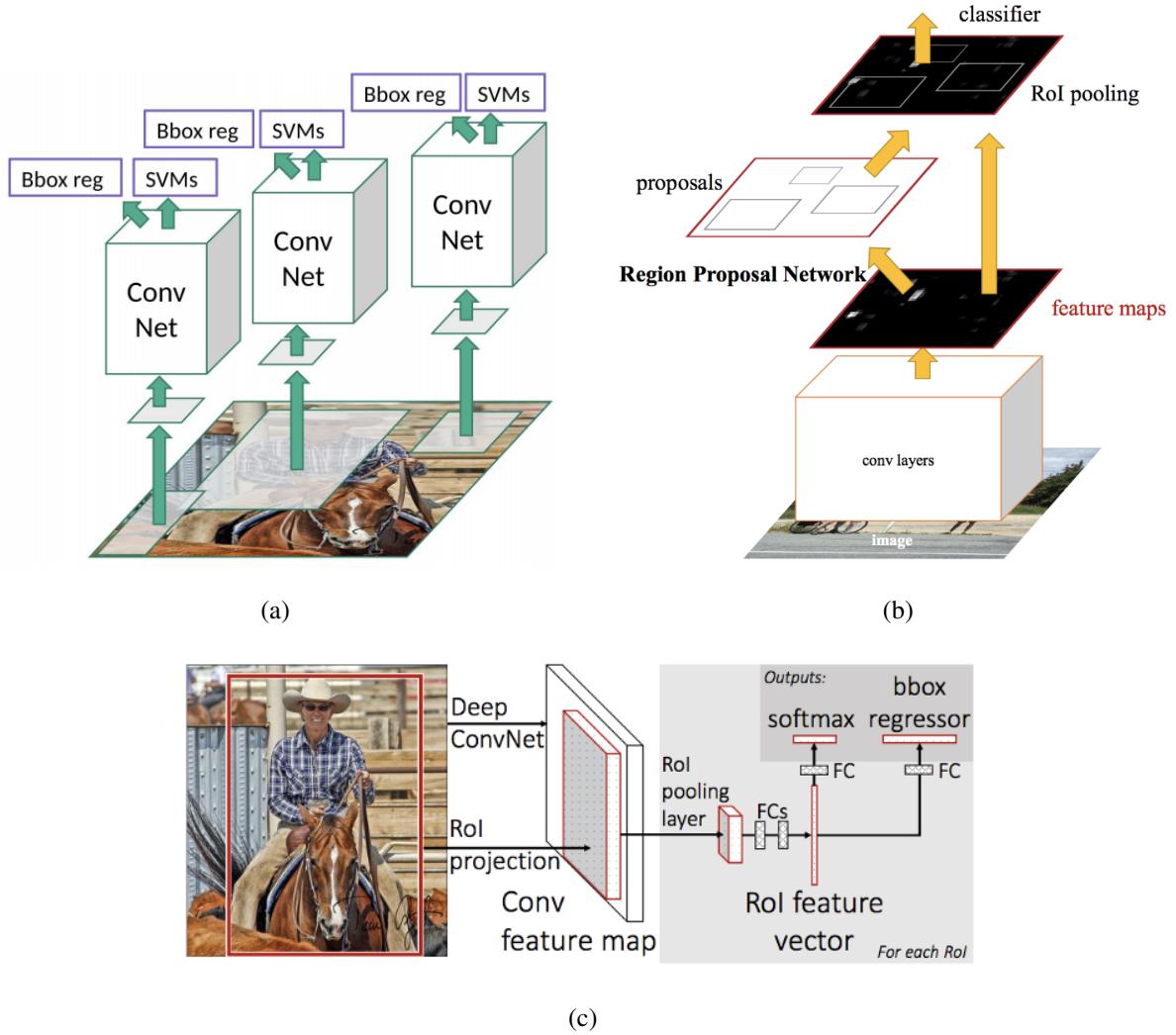
Rys. 3.5. Przykładowa architektura sieci CNN wykorzystująca warstwy konwolucji, *Max Pooling*, *FC* oraz funkcje *ReLU* przystosowane do zadania klasyfikacji obrazu. Źródło: [21].

3.3.2. R-CNN

We wcześniej wspomnianych metodach klasycznych wykorzystywano technikę okna przesuwnego oraz skalowanie obrazu. W metodzie *R-CNN* (ang. *Regions with CNN features*) [22] analizowane są jedynie wybrane regiony obrazu. Wybór regionów jest dokonywany za pomocą wybranego algorytmu np. przeszukiwania selektywnego (ang. *Selective Search*) [23]. Następny etap polega na ekstrakcji cech z uzyskanych wcześniej obszarów. Wymagane jest aby wektory cech posiadały ustalony rozmiar. Z tego powodu proponowane obszary są skalowane do ustalonego rozmiaru. Ekstrakcja cech odbywa się z użyciem przetrenowanej (np. na zbiorze [24]) sieci CNN. Możliwe jest wykorzystanie dowolnych architektur CNN np. *AlexNet* [25], *VGG-16* [26]. Ostatni etap to klasyfikacja oraz regresja. W tym celu należy przeprowadzić trening dowolnego klasyfikatora np. sieć *FC* lub *SVM* stwierdzającego obecność lub brak rozpatrywanego typu obiektu w rozpatrywanym obszarze. W przypadku wykrycia obiektu znane są jego przybliżone położenie oraz wymiary. Celem zwiększenia dokładności detekcji dokonywane są estymaty położenia i wymiarów, poprzez regresję liniową.

Etap ekstrakcji cech wymagał przetworzenia każdego obszaru obrazu przez całą sieć. Możliwe jest również wstępne przetworzenie całego obrazu tylko przez warstwy konwolucyjne. Uzyskiwane są w ten sposób mapy cech dla całego obrazu. Następnie fragmenty map odpowiadające wejściowym proponowanym regionom, po odpowiednim przeskalowaniu do ustalonego wymiaru, poddawane są klasyfikacji oraz regresji parametrów. Do tego celu używana jest sieci neuronowa wykorzystująca warstwy *FC*. Pozwala to na zwiększenie szybkości działania systemu. Wspomniane rozwinięcie metody nazywane jest *Fast R-CNN* [27].

Obie wspomniane metody bazowały na proponowanych poprzez wybrany algorytm regionach obrazu. Etap ten można zastąpić użyciem do tego celu wytrenowanej sieci neuronowej nazywanej *Region Proposal Network* (RPN). Ponadto sieć ta może bazować na mapach cech



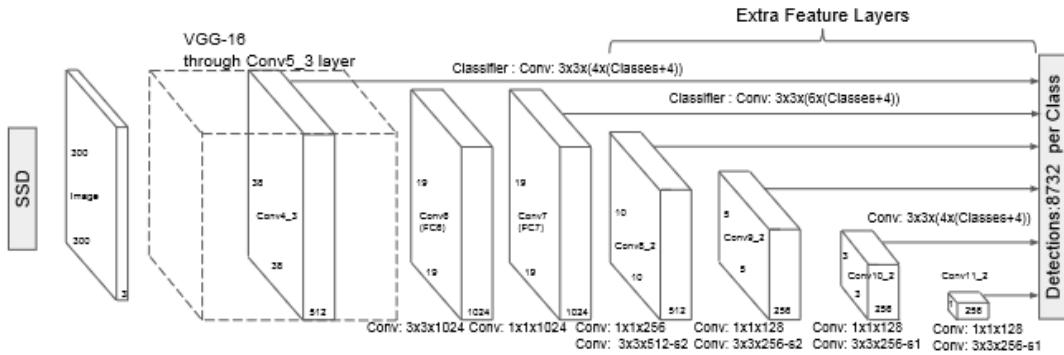
Rys. 3.6. Architektury R-CNN 3.6a, Faster R-CNN 3.6b oraz Fast R-CNN 3.6c.

Źródło: [29].

wykorzystywanych do ekstrakcji wektorów cech stałymiarowych dla proponowanych regionów. Rozwiązanie to nazywane jest *Faster R-CNN* [28].

3.3.3. SSD

Systemy detekcji bazujące na rozwiążaniu *R-CNN* wymagają dwuetapowego przetwarzania obrazu. W architekturze *Single Shot Detector* (SSD) [30] detekcja obiektów wykonywana jest jednoetapowo. Wykorzystywana do tego celu jest jednokierunkowa sieć w pełni konwolucyjna (ang. *Fully Convolutional Neural Network, FCNN*) stanowiąca tzw. sieć bazową (ang. *base network*) przetrenowaną do zadania klasyfikacji (bez "głowy" klasyfikatora). Za ostatnią warstwą sieci bazowej dołączane są dodatkowe warstwy konwolucyjne oraz *Max pooling* celem zmniejszenia rozmiaru map cech. Do wybranych warstw pośrednich przyłączane są kolejne



Rys. 3.7. Architektura SSD. Do sieci bazowej dołączone są kolejne warstwy wraz rozgałęzieniami. Źródło [30].

warstwy tworząc rozgałęzienia. Każde rozgałęzienie zakończone jest warstwą konwolucyjną, której filtry realizują zadania klasyfikacji wykrycia obiektu, typu oraz regresji jego parametrów. Na rysunku 3.7 przedstawiono architekturę SSD, gdzie jako sieć bazową wykorzystano VGG-16. Estymacja wymiarów odbywa się względem ustalonych wymiarów (d_w, d_h) nazywanych *default box* (dla każdego zastawu filtrów detekcja + klasyfikacja może zostać dobrany inny wymiar (d_w, d_h)). Położenie obiektu jest wyznaczane w oparciu o rezultaty odpowiednich filtrów (f_x, f_y, f_w, f_h), lecz także o ich położenie (i, j) w siatce (ang. *grid*) wynikającej z wymiarów warstwy (d_N, d_M) znormalizowane do wymiarów *default box*. Ostatecznie detekcja parametrów obiektu daje się przedstawić za pomocą równań (3.13)-(3.16) ((W, H) oznaczają wymiary obrazu wejściowego).

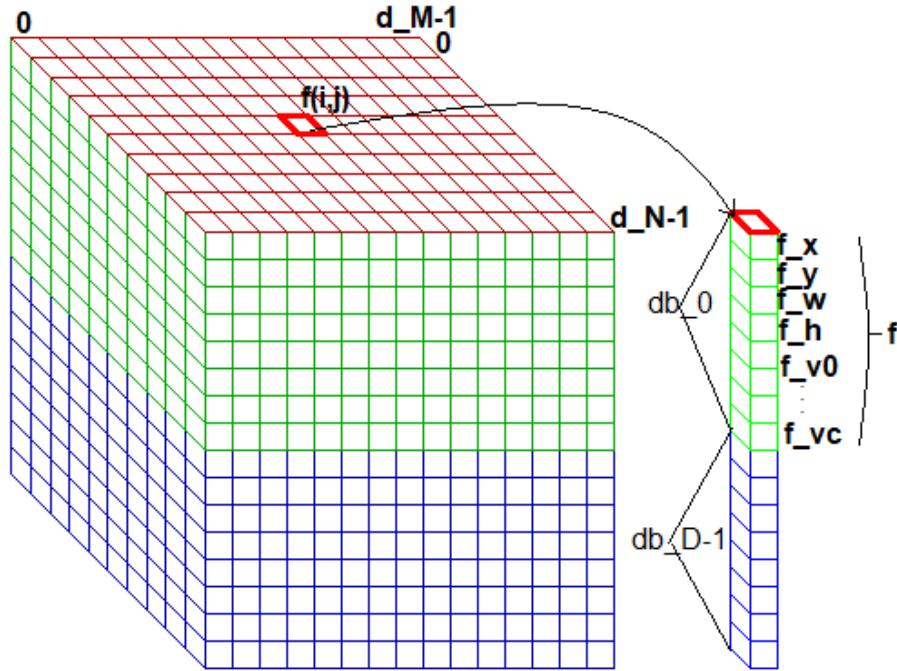
$$x_s = \left(\frac{j}{d_N} + 0.5 \right) W + d_w f_x(i, j) \quad (3.13)$$

$$y_s = \left(\frac{i}{d_M} + 0.5 \right) H + d_h f_y(i, j) \quad (3.14)$$

$$w = d_w e^{f_w(i, j)} \quad (3.15)$$

$$h = d_h e^{f_h(i, j)} \quad (3.16)$$

Trening sieci odbywa się w oparciu o sumę błędu klasyfikacji oraz błędu regresji. Błąd regresji L_{reg} jest reprezentowany przez sumę błędów bezwzględnych znormalizowanego przesunięcia względem środka w siatce oraz wykładnika eksponencjalnego przeskalowania względem wymiarów *default box*. Równanie (3.17) przedstawia funkcję błędu dla pojedynczego *default*



Rys. 3.8. Siatka detekcji dla D *default box*. Poprzez db oznaczono grupę filtrów przypisanych do konkretnego *defult box*. Filtry $v0, \dots vc$ oznaczają kolejne klasy obiektów.

box dla referencyjnych parametrów $i, j, x_{ref}, y_{ref}, w_{ref}$ oraz h_{ref} . Poprzez f oznaczono rezultaty filtracji wcześniej wspomnianymi filtrami.

$$\begin{aligned} L_{reg}(f, d, i, j, x_{ref}, y_{ref}, w_{ref}, h_{ref}) = & |f_x(i, j) - \frac{x_{ref} \frac{d_N}{W} - j - 0.5}{d_w}| \\ & + |f_y(i, j) - \frac{y_{ref} \frac{d_M}{H} - i - 0.5}{d_h}| \\ & + |f_w(i, j) - \log\left(\frac{w_{ref}}{d_w}\right)| \\ & + |f_h(i, j) - \log\left(\frac{h_{ref}}{d_h}\right)| \end{aligned} \quad (3.17)$$

Błąd klasyfikacji L_{class} jest obliczany przy pomocy funkcji danej wzorem (3.18). Poprzez f oznaczono zbiór filtrów realizujących zadanie klasyfikacji, C oznacza referencyjną mapę klas dla danego *default box* d , gdzie każdy element mapy oznacza numer klasy. Należy zauważyć, iż dla elementów, dla których nie powinno nastąpić wykrycie, również przypisana jest klasa oznaczająca brak wykrycia obiektu.

$$L_{class}(f, d, C) = - \sum_{ii}^{d_M} \sum_{jj}^{d_N} \log(sm(f_v))_{C(ii,jj)}(ii, jj) \quad (3.18)$$

Przedstawione równania stanowią składowe błędu całej sieci L , przy czym są one sumowane dla każdego *default box* oraz dla każdego zestawu parametrów referencyjnych. W równaniu (3.19) poprzez D oznaczono zbiór *default box* d . Przyjmując za do dd jako wybrany *default box*, $f^d d$ stanowi zestaw filtrów przypisanych dd , $C^d d$ jest maską referencyjną klas dla dd . Poprzez R oznaczono zbiór parametrów referencyjnych r opisujących dany obiekt.

$$L(f, D, C, R) = \sum_{d \in D} L_{class}(f^d, d, C^d) + \sum_{r \in R} L_{reg}(f^{r_d}, r_d, r_i, r_j, r_x, r_y, r_w, r_h) \quad (3.19)$$

W przedstawionych równaniach filtr wykrycia obiektu nie występuje jawnie, lecz pod postacią klasy “niewykrycia obiektu”. Architektura *SSD* znaczco upraszcza implementację zadania detekcji, poprzez jednoetapowe przetwarzanie obrazu.

3.3.4. YOLO

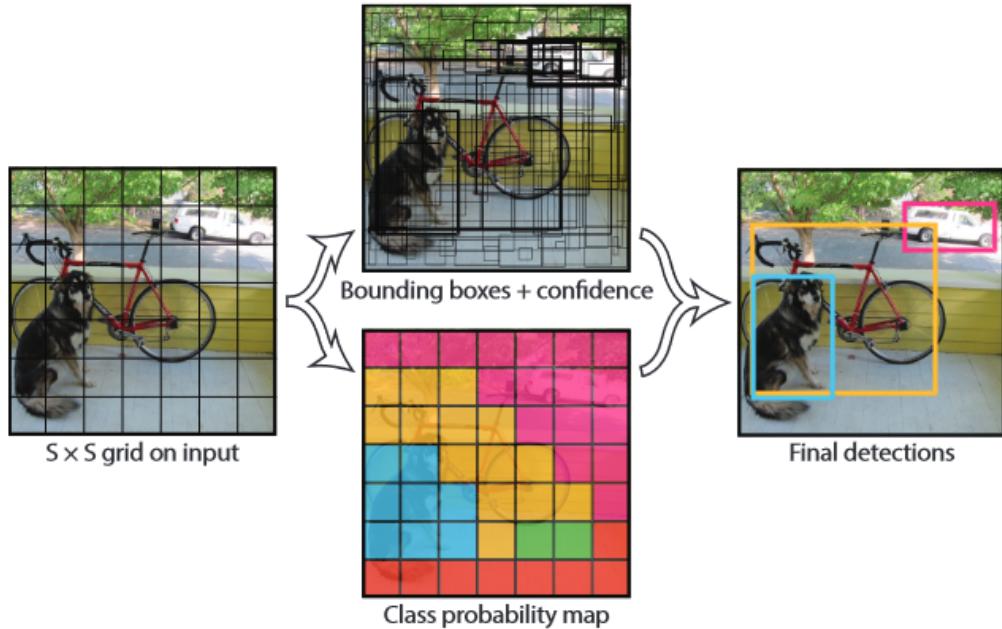
Poprzednikiem architektury *SSD* w detekcji jednoetapowej była architektura *YOLOv1* (ang. *You Only Look Ones*) [31]. Architektura ta wykorzystywała podział obrazu na siatkę o określonych wymiarach odpowiadających wymiarom ostatniej warstwy typu *Fully Connected (FC)*. W przypadku architektury *SSD* kolejne filtry stanowiły klasyfikację oraz regresję, dla *YOLOv1* tę rolę spełniają odpowiednie neurony warstwy *FC* poprzez stosowną interpretację dla wymiaru $BxSxSx(5 + C)$, gdzie B to liczba estymowanych *bounding box*, S wymiar siatki, C liczba rozpatrywanych klas. Stała wartość 5 oznacza liczbę neuronów spełniających rolę wykrycia obiektu oraz estymacji parametrów obiektu. Trening architektury sprowadzał się do minimalizacji ważonej sumy kwadratów błędów dla położenia, błędów pierwiastków wymiarów, błędów klasy oraz błędów wykrycia obiektu. Ponadto błędy te były wyznaczane tylko dla elementów siatki, w których występował (częściowo lub w pełni) obiekt referencyjny. Na rysunku 3.9 zilustrowano metodę detekcji opartą o architekturę *YOLO*.

Wykorzystanie warstw *FC* w *YOLOv1* uniemożliwiło realizację przetwarzania w wielu skalach. Z tego powodu w następnych wersjach *YOLOv2* [32] oraz *YOLOv3* [33] również odrzucono warstwę *FC* na rzecz warstw konwolucyjnych z filtrami f o wymiarach 1x1 uzyskując architekturę *FCNN*. Wyznaczenie parametrów obiektów zostało przedstawione poprzez równania (3.20)-(3.23). Poprzez (a_w, a_h) oznaczono wymiary tzw. *anchor box*, stanowiącego odpowiednik *default box* w *SSD*. Wymiary siatki oznaczono poprzez (a_N, a_M) . Natomiast (a_x, a_y) oznaczają położenie indeksy elementu siatki, dla którego wyznaczane są parametry.

$$x_c = (\sigma(f_x) + a_x) \frac{W}{a_N} \quad (3.20)$$

$$y_c = (\sigma(f_y) + a_y) \frac{H}{a_M} \quad (3.21)$$

$$w = a_w e^{f_w} \quad (3.22)$$



Rys. 3.9. Metoda detekcji oparta o architekturę YOLO. Źródło: [31]

$$h = a_h e^{f_h} \quad (3.23)$$

Zastosowanie funkcji logistycznej w estymacji położenia pozwala na osiągnięcie większej stabilności regresji [32], czy korelacji pomiędzy wykryciem obiektu w siatce, a predykcją jego położenia. Linowa estymacja odchyłki położenia pozwalała na "przesunięcie" obiektu do elementu siatki o niskiej wartości wykrycia. Wiązało by się to co najmniej z brakiem spójności pomiędzy wykryciem obiektu, a jego położeniem. Powodem tego był brak ograniczenia przyjmowanych wartości poszczególnych neuronów. Ponadto referencyjne wartości wykrywalności obiektu dla YOLOv2 stanowią wartość współczynnika IoU pomiędzy obiektem referencyjnym, a rozpatrywanym *anchor box* (stanowiącym odpowiednik *default box* w SSD). Dla YOLOv3 wykrywalność obiektu (autorzy używają sformułowania *objectness score* [33]) jest zrealizowana jako regresja logistyczna. Z tego powodu wartości referencyjne przyjmują wartość 1 w przypadku, gdy wartość IoU jest większa niż ustalona wartość progowa oraz 0 w przeciwnym przypadku.

Kolejna wersja YOLOv4[34] znaczco poprawia rezultaty względem wersji poprzednich, lecz jest również znacznie bardziej skomplikowana. Trening YOLOv3 zakładał minimalizację sumy kwadratów błędów dla części regresyjnej. W wersji YOLOv4 autorzy[34] zalecają użycie funkcji błędu reprezentowanej przez *CIoU* [35] danej równaniem (3.26). Wspomniane są również inne funkcje bazujące na metryce *IoU* takie jak *GIoU*[36] (3.24) oraz *DIoU* [35] (3.25).

$$GIoU(A, B) = 1 - IoU(A, B) + giou_{part}(A, B) \quad (3.24)$$

$$DIoU(A, B) = 1 - IoU(A, B) + diou_{part}(A, B) \quad (3.25)$$

$$CIoU(A, B) = 1 - IoU(A, B) + diou_{part}(A, B) + ciou_{part}(A, B) \quad (3.26)$$

$$giou_{part}(A, B) = \frac{|C \setminus (A \cup B)|}{|C|} \quad (3.27)$$

$$diou_{part}(A, B) = \frac{(A_{x_c} - B_{x_c})^2 + (A_{y_c} - B_{y_c})^2}{C_w^2 + C_h^2} \quad (3.28)$$

$$v(A, B) = \frac{4}{\pi^2} (\arctan(\frac{A_w}{A_h}) - \arctan(\frac{B_w}{B_h}))^2 \quad (3.29)$$

$$\alpha(A, B) = \frac{v(A, B)}{1 - IoU(A, B) + v(A, B)} \quad (3.30)$$

$$ciou_{part}(A, B) = \alpha(A, B)v(A, B) \quad (3.31)$$

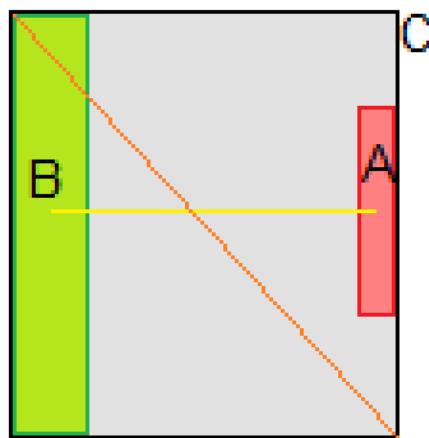
Oznaczenia:

- A, B, C – bounding box
- C – najmniejszy bounding box taki, że $A, B \subseteq C$

Analizując powyższe wzory można dostrzec, iż gdy:

- obiekty nie posiadają części wspólnej $IoU = 0$,
- obiekty zawierają się w sobie $giou_{part} = 0$,
- środek obiektów znajduje się w tym samym punkcie $diou_{part} = 0$,
- $ciou_{part}$ ma wpływ tylko na błąd wynikający z wymiarów obiektu.
- $ciou_{part}$ dla obiektów o tym samym stosunku wymiarów, lecz o różnych wymiarach przybiera wartość 0.

Funkcja $DIoU$ minimalizuje odstęp pomiędzy obiektami, $CIoU$ rozwija tę funkcję o skalowanie wymiarów, natomiast $GIoU$ pozwala na ograniczenie obszaru poza obiektami referencyjnym oraz wyznaczonym z odpowiedzi sieci. Na rysunku 3.10 przedstawiono graficzne interpretacje błędów.



Rys. 3.10. Graficzna reprezentacja $GIoU$, $DIoU$ dla obiektów A (czerwony) oraz B (zielony) o tym samym stosunku wymiarów ($ciou_{part} = 0$). Poprzez C (czarny) oznaczono najmniejszy *bounding box* zawierający obiekty A i B . $diou_{parts}$ to stosunek długości linii żółtej do pomarańczowej. $giou_{part}$ to stosunek szarego pola powierzchni ($C \setminus (A)$) do pola powierzchni C .

3.4. Przegląd rozwiązań energooszczędnych

Dotychczas omawiane metody detekcji nie były projektowane z myślą o optymalizacji zużycia energii czy łatwości implementacji sprzętowej. Ze względu na to, iż celem konkursu DAC SDC 2021 jest sprzętowa implementacja w układzie FPGA, warto również rozpatrzyć rozwiązania pozwalające nie tylko osiągnąć wysoką, dokładność, lecz również zmniejszyć zużycie energii i zwiększyć przepustowość.

3.4.1. Konwolucja separowalna

Jednym z częstych rozwiązań przeznaczonych dla urządzeń mobilnych o małej mocy obliczeniowej jest zastosowanie konwolucji separowalnych (ang. *Separable Convolution*). Wykonywane są tu kolejno po sobie dwa typy konwolucji *Depthwise (DW)* oraz *Pointwise (PW)*. Konwolucja *DW* polega na niezależnej filtracji poszczególnych kanałów (głębii) (podobnie jak ma to miejsce w przypadku konwolucji dla obrazu w skali szarości). Maska konwolucji *DW* posiada jedynie dwa wymiary przez co możliwa jest analiza kontekstu jedynie w obrębie jednego kanału. Konwolucja *PW* stanowi szczególny przypadek pełnej konwolucji (standardowej) z filtrem o wymiarach 1×1 (stosowanym np. w warstwach *YOLO*). Pozwala to uprościć obliczenia ze względu na brak analizowania kontekstu / otoczenia rozpatrywanego punktu (ang. *point*) maski cech. Maska konwolucji *PW* posiada jeden wymiar równy liczbie kanałów wejściowych. Połączenie obu konwolucji pozwala na aproksymację pełnej konwolucji zachowując własność

analizy kontekstu pomiędzy kanałami, przy zmniejszeniu liczby obliczeń. Ponadto po konwolucji *DW* możliwe jest zastosowanie dodatkowych nieliniowości. Pierwszą architekturą, w której zastosowanie konwolucji separowalnych przyniosło olbrzymi sukces był *GoogLeNet*[37]. Dla zastosowań mobilnych tego typu konwolucje zostały wprowadzone w architekturze *MobileNet*[38]. Zastosowanie konwolucji separowalnych pozwala zarówno na ograniczenie pamięci wymaganej na przechowywanie modelu, jak również na zmniejszenie liczby wymaganych obliczeń. Rysunki 3.11b oraz 3.11c przedstawiają schemat konwolucji *DW* oraz *PW*.

3.4.2. Sieci kwantyzowane

Obliczenia zmiennoprzecinkowe pozwalają na osiągnięcie wysokiej dokładności obliczeń wraz z szerokim zakresem zmienności. Czas obliczeń zmiennoprzecinkowych jest zazwyczaj dłuższy w porównaniu do obliczeń całkowitoliczbowych. Algorytmy sieci neuronowych w większości przypadków są projektowane do implementacji z wykorzystaniem wspomnianego typu obliczeń. Jednakże nie zawsze wymagana jest wysoka precyzaja obliczeń. Wówczas możliwe jest zastosowanie tzw. kwantyzowanych sieci neuronowych (ang. *Quantized Neural Networks, QNN*) [39] możliwych do implementacji z wykorzystaniem obliczeń całkowitoliczbowych (ang. *integer*) czy stałoprzecinkowych (ang. *fixedpoint*). Aby jednak możliwa była tak implementacja wymagane jest przeprowadzeniu kwantyzacji. Proces polega na odpowiednim przetransformowaniu wag oraz wyników pośrednich tak, aby odwzorować obliczenia o ograniczonej precyzyji. Równanie (3.37) przedstawia kwantyzację stałoprzecinkową. Funkcja Q wykonuje operację kwantyzacji stałoprzecinkowej zmiennej zmiennoprzecinkowej x , dla zapisu na b bitach z bitem znaku ($s = 1$) lub bez ($s = 0$), przy wykorzystaniu i bitów na część całkowitą i znak oraz z metodą przybliżenia do liczby całkowitej n .

$$\min_{int}(b, s) = -s2^{b-s} \quad (3.32)$$

$$\max_{int}(b, s) = 2^{b-s} - 1 \quad (3.33)$$

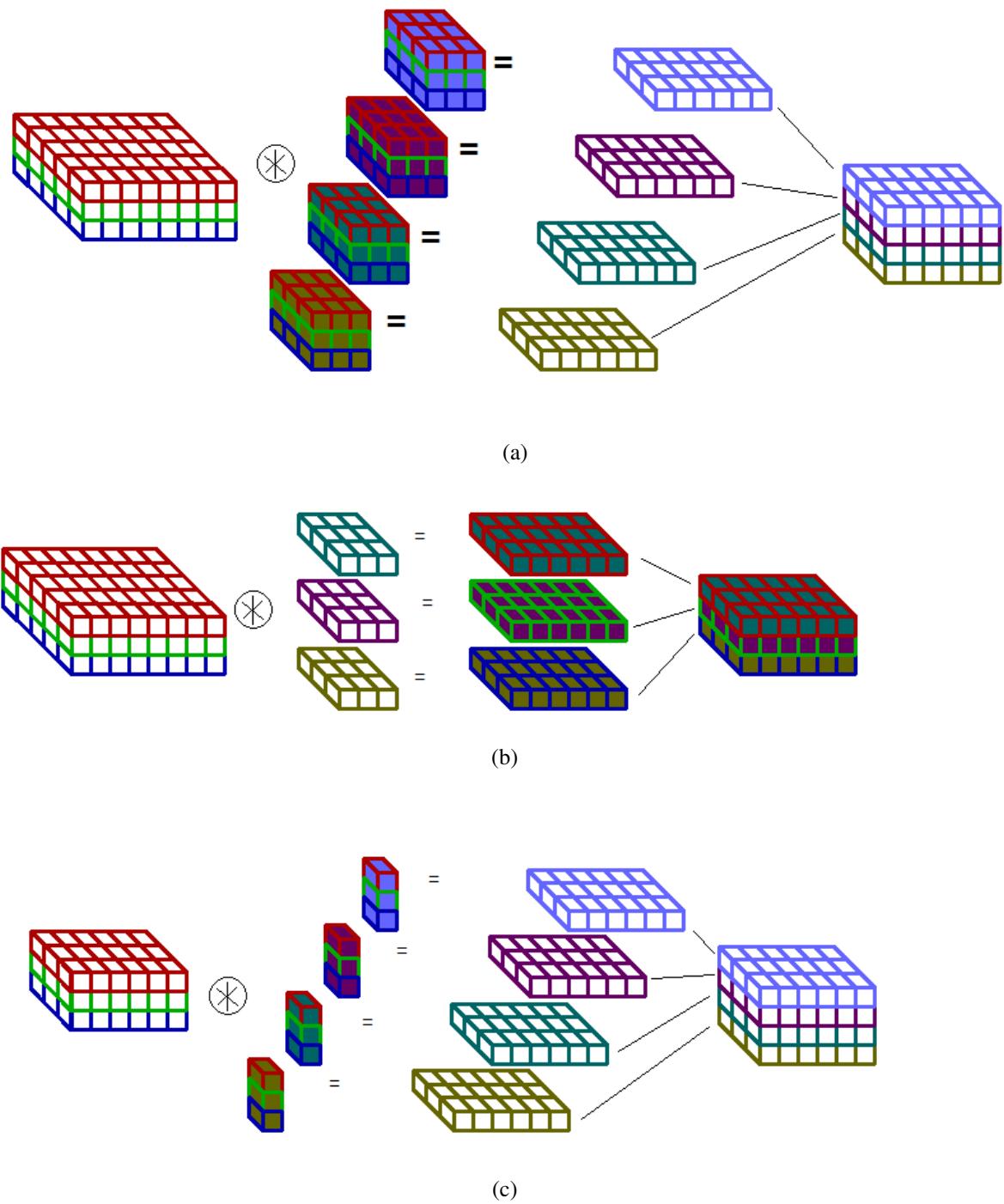
$$\text{limit}(x_{int}, b, s) = \min(\max(x, \min_{int}(b, s)), \max_{int}(b, s)) \quad (3.34)$$

$$\text{scale}(b, i) = \frac{1}{2^{b-i}} \quad (3.35)$$

$$\text{int}(x, b, i, s, n) = \text{limit}\left(n\left(\frac{x}{\text{scale}(b, i)}\right), b, s\right) \quad (3.36)$$

$$Q(x, b, i, s, n) = \text{int}(x, b, i, s, n) \text{ scale}(b, i) \quad (3.37)$$

Funkcja Q zwraca rezultat w postaci przybliżenia do liczby stałoprzecinkowej. Jako metody przybliżenia możliwe jest użycie zaokrąglenia w dół (podłoga, ang. *floor*), w górę (sufit, ang. *ceil*) lub do najbliższej wartości całkowitej (ang. *round*). Model sieci z przeprowadzoną kwantyzacją wymaga dalszego uczenia, ze względu możliwą utratę dokładności lub ograniczenie wartości do zakresu liczb stałoprzecinkowych. Uzyskanie zapisu z wykorzystaniem liczb



Rys. 3.11. Schemat konwolucji pełnej 3.11a, *depthwise* 3.11b oraz *pointwise* 3.11c.

całkowitych (celem ewaluacji sprzętowej lub programowej modelu) jest możliwe poprzez równanie (3.36).

3.4.3. Sieci binarne

Przedstawiona dotychczas kwantyzacja wykorzystywała więcej niż 1 bit do zapisu danych. Przy kwantyzacji 1 bitowej uzyskiwana jest tzw. binarna sieć neuronowa (ang. *Binary Neural Network, BNN*) [39]. W sieciach binarnych odpowiednikiem operacji mnożenia jest operacja *xnor*, natomiast sumowania operacja zliczania bitów. Aby dokonać kwantyzacji binarnej Q_b zmiennej x wykorzystuje się funkcję *sign*. Otrzymywana wartość pochodzi ze zbioru $\{-1, 1\}$.

$$Q_b(x) = \text{sign}(x) \quad (3.38)$$

Kwantyzację binarną można również zrealizować w sposób stochastyczny [39]. Przedstawiono poprzez równanie (3.39). Poprzez r oznaczono zmienną losową z przedziału $[0; 1]$. Funkcja σ_h oznacza funkcję *hard sigmoid*.

$$Q_{bs}(x) = \begin{cases} 1 & r < \sigma_h(x) \\ -1 & \text{otherwise} \end{cases} \quad (3.39)$$

Celem dalszego procesu uczenia wykorzystuje się dodatkowy parametr skalujący rezultat kwantyzacji uzyskiwany w sposób analityczny [40] lub stanowiący jeden z parametrów procesu uczenia [41]. Dodatkowo w [39] zaleca się ograniczenie wartości wag i wyników pośrednich do przedziału $[-1; 1]$

Sieci binarne pozwalają na znaczącą redukcję pamięci potrzebnej do przechowywania wag, a także znacząco upraszczają obliczenia.

3.4.4. Ultra_net

Mając na uwadze iż jednym z warunków konkursu jest udostępnienie pełnego rozwiązania w formie otwarto-źródłowej, możliwa jest analiza rozwiązań z edycji poprzedzających rozpatrywaną. Zwycięski zespół poprzedniej edycji (2020) wykorzystał architekturę sieci *Ultra_net* [42]. Jest to architektura *FCNN* zawierająca 9 warstw kowolucyjnych, warstwy normalizujące, warstwy *Max Pooling* oraz warstwę *YOLOv3* [33]. Ponadto warstwy konwolucyjne zawierały co najwyżej 64 filtry o wymiarach kontekstu 3x3. Sprzętowa akceleracja została zrealizowana z użyciem języka *HLS* (ang. *High Level Synthesis*). Wykorzystano tutaj 4 bitową kwantyzację wag konwolucji. Dla warstw normalizacyjnych kwantyzacja była zmienna (stała dla warstwy, lecz zależna od jej położenia). Rozwiązanie osiągnęło dokładność detekcji 0.656 współczynnika *IoU*, przepustowość 212.726 *fps* oraz zużyło 1641.1 *J* energii.

3.4.5. SkyNet

Innym rozwiązaniem jest architektura *SkyNet* [43]. Przez 3 ostatnie edycje architektura ta osiągała jedne z najlepszych wyników. Jest to również sieć typu *FCNN*, wyróżniająca się zastosowaniem konwolucji seperowalnych oraz funkcji aktywacji *ReLU6*. W każdym bloku konwolucji separowalnej, po każdej warstwie konwolucji *DW* oraz *pPW* występuje warstwa normalizacji z funkcją aktywacji. Ostatnią warstwę sieci stanowi warstwa *YOLOv3* (podobnie jak w przypadku *Ultra_net*). Liczba filtrów w warstwach oraz położenie warstw *Max Pooling* zostały zoptymalizowane z użyciem algorytmu *PSO* (ang. *Particle Swarm Optimization*). Implementacja została zrealizowana z użyciem języka *HLS*. Wykorzystano tutaj dwa akceleratory - po jednym dla każdego typu konwolucji. Każda kolejna warstwa była obliczana sekwencyjnie tzn. w jednej chwili czasu obliczana była tylko jedna warstwa. Rozwiązanie osiągnęło dokładność detekcji 0.716 *IoU*, przepustowość 25.05 *fps* oraz zużyło 7260 *J* energii dla implementacji na płytce *Avnet Ultra96 VI*.

Architektura *SkyNet* osiąga lepszą jakość detekcji, niż architektura *Ultra_net*. Jednakże zrealizowana implementacja charakteryzuje się stosunkowo niską przepustowością oraz wysokim zużyciem energii.

Zadanie detekcji sprowadza się do znalezienia położenia obiektu oraz jego wymiarów, a często również określenia typu obiektu (klasyfikacja). Rozwiązanie problemu możliwe jest z wykorzystaniem zarówno metod klasycznych i okna przesuwnego czy wyznaczeniu rozmiarów obiektu na podstawie obrazu binarnego. Uczestnictwo w konkursie *DAC SDC 2021* wymagała zastosowania do tego celu sztucznych sieci neuronowych. Detekcja obiektów możliwa jest poprzez wieloetapowe przetwarzanie obrazu wykorzystujące metody bazujące na *R-CNN*. Konkurencyjnym podejściem jest detekcja jednoetapowa wykorzystująca sieci *SSD* czy *YOLO*. Dla sprzętowej implementacji dominuje jednak podejście wykorzystujące architekturę *YOLOv3* wraz z odpowiednią kwantyzacją wag oraz wyników pośrednich. W rozwiązaniach mobilnych stosowanie konwolucji seperowalnych pozwala ograniczyć wymaganą liczbę obliczeń. Ponadto zastosowanie błędu bazującego na metryce *IoU* w trakcie procesu uczenia pozwala osiągnąć lepsze rezultaty oraz szybszą zbieżność, niż dla błędów kwadratowego czy bezwzględnego.

4. Badania wstępne

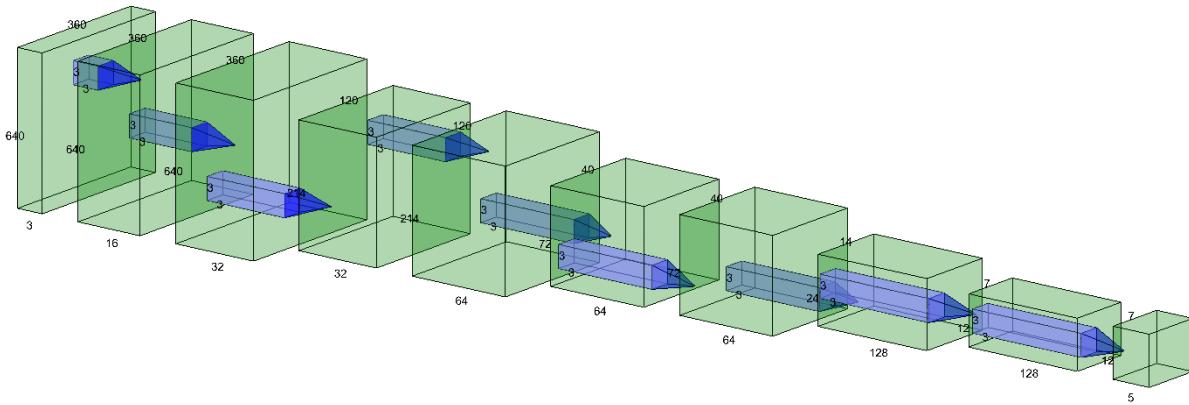
Analiza rozpatrywanego problemu detekcji pozwoliła zaznaczyć przestrzeń rozważanych rozwiązań do wybranych typów warstw, architektur czy metod detekcji. Ponadto, na tej podstawie możliwe jest zaproponowanie własnego rozwiązania. W niniejszym rozdziale zostaną przedstawione propozycje architektur sieci, proces ich uczenia, a także kwantyzacji.

4.1. Architektura wstępna

Wspomniane w rozdziale 3.4.1 konwolucje separowalne pozwalają zredukować liczbę parametrów sieci, a także wymaganą moc obliczeniową. Przykładowo sieć *SkyNet* (wersji bez połączenia typu *bypass*) wykorzystująca konwolucje separowalne posiada ponad 300 tys. parametrów. Analogiczna architektura wykorzystująca pełną konwolucję posiada ponad 2.6 miliona parametrów. Tak znaczna redukcja (również złożoności obliczeniowej) skłania do zastosowania konwolucji separowalnych w przypadku akceleracji sprzętowej. Postanowiono również sprawdzić czy możliwa jest dalsza redukcja liczby parametrów pozwalająca na uzyskanie satysfakcyjnego wyniku detekcji. W tym celu rozważono architekturę sieci o 9 warstwach konwolucji separowalnej (z bias) z funkcją aktywacji *ReLU* oraz 4 warstwami *Max Pooling* umieszczonymi po wybranych konwolucjach separowalnych. Ostatnia warstwa stanowi warstwę bazującą na warstwach *YOLOv1* oraz *YOLOv2* (rozdział 3.3.4). Na rysunku 4.1 przedstawiono graficzną reprezentację architektury. Jako wejście sieci pozostawiono oryginalne wymiary obrazów dostępnego w zbiorze treningowym.

Ze względu, iż rozpatrywany problem to detekcja bez klasyfikacji (przy czym wykrywane powinny być tylko obiekty należące do jednej z 12 klas), pominięto w funkcji błędu (4.8) część związaną z klasyfikacją. Obliczenie błędów poszczególnych neuronów jest realizowane poprzez porównanie z maską referencyjną wyjść y_{ref} (o wymiarach $(5, d_h, d_w)$) z odpowiadającymi wartościami dla kanałów. Równania (4.1)-(4.7) prezentują sposób wyznaczenia wartości referencyjnych dla elementów poszczególnych kanałów oraz wyznaczenie położenia obiektu w siatce.

Poprzez x_c, y_c, w, h oznaczono parametry obiektu referencyjnego, a_w oraz a_h odpowiednio szerokość i wysokość *anchor box*, $W = 640$ oraz $H = 360$ wymiary obrazu wejściowego,



Rys. 4.1. Wstępna architektura sieci dla zadania detekcji, wykorzystująca konwolucje separowalne oraz warstwę podobną do *YOLOv1*. Architektura posiada niespełna 43 tys. parametrów.

$d_w = 12$ oraz $d_h = 7$ wymiary siatki detekcji. Przyjęto $a_w = 16$ oraz $a_h = 16$ jako wartość łączną wartości kroku sieci (ang. *stride*) wynikającą z liczby warstw *Max Pooling*.

$$y_{ref0,i,j} = \begin{cases} 1, & \text{if } i = row \wedge j = col \\ 0, & \text{otherwise} \end{cases} \quad (4.1)$$

$$y_{ref1,i,j} = \begin{cases} x_c \frac{d_w}{W} - col - 0.5, & \text{if } i = row \wedge j = col \\ 0, & \text{otherwise} \end{cases} \quad (4.2)$$

$$y_{ref2,i,j} = \begin{cases} y_c \frac{d_h}{H} - row - 0.5, & \text{if } i = row \wedge j = col \\ 0, & \text{otherwise} \end{cases} \quad (4.3)$$

$$y_{ref3,i,j} = \begin{cases} \log(\frac{w}{a_w}) & \text{if } i = row \wedge j = col \\ 0, & \text{otherwise} \end{cases} \quad (4.4)$$

$$y_{ref4,i,j} = \begin{cases} \log\left(\frac{h}{a_h}\right) & \text{if } i = \text{row} \wedge j = \text{col} \\ 0, & \text{otherwise} \end{cases} \quad (4.5)$$

$$col = \lfloor x_c \frac{d_w}{W} - 0.5 \rceil \quad (4.6)$$

$$row = \lfloor y_c \frac{d_h}{H} - 0.5 \rceil \quad (4.7)$$

Pierwszy kanał $y_{ref0,:}$ oznacza referencyjną wartość dla wykrycia obiektu. Zrezygnowano tutaj z wyznaczania wartości na podstawie IoU i przyjęto tylko jeden element maski za niezerowy. Kolejnym kanałom dla elementu na pozycji (row, col) przypisano odpowiednio

przetransformowane parametry obiektu referencyjnego. Kanały $y_{ref1:2,:,:}$ stanowią odchylenie środka obiektu od środka komórki (elementu (row, col)) siatki znormalizowane do wymiaru komórki siatki. Kolejne dwa kanały $y_{ref3:4,:,:}$ przedstawiają wykładnik przeskalowania eksponentonalnego wymiarów obiektu do wymiarów *anchor box*.

Błąd wykrycia obiektu został zdefiniowany jako entropia binarna *BCE* (ang. *Binary Cross Entropy*). Błąd przesunięcia od centrum siatki detekcji oraz estymacja wymiarów został zdefiniowany jako średni błąd bezwzględny *MAE* (ang. *Mean Absolute Error*). Funkcję błędu (reprezentowaną przez równanie (4.8)) dla całej sieci stanowi ważona suma błędów składowych ze współczynnikami λ_{obj} , λ_x oraz λ_{wh} dobieranymi w sposób eksperymentalny. Poprzez y_{pred} oznaczono odpowiedź sieci.

$$\begin{aligned} loss = & \lambda_{obj} BCE(\sigma(y_{pred0,:,:}), y_{ref0,:,:}) \\ & + \lambda_{xy} MAE(y_{pred1:2,:,:}, y_{ref1:2,:,:}) \\ & + \lambda_{wh} MAE(y_{pred3:4,:,:}, y_{ref3:4,:,:}) \end{aligned} \quad (4.8)$$

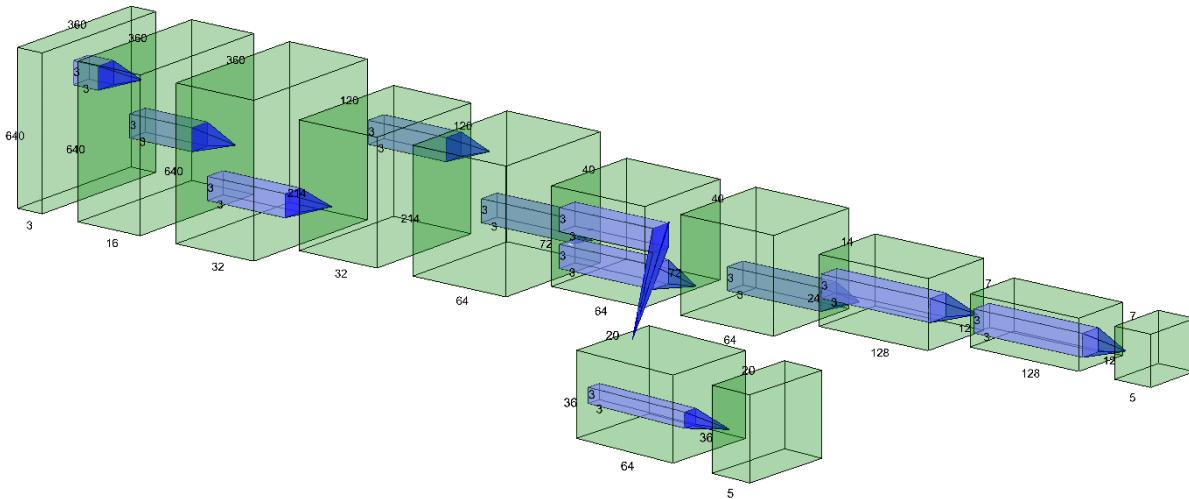
Przed rozpoczęciem procesu uczenia dokonano podziału dostępnego zbioru uczącego na trzy zbiory: uczący, walidacyjny oraz testowy w stosunku 81:9:10. Rozdzielenie danych odbywało się losowo wewnętrz każdej sekwencji. Uzyskane zbiory były stosowane dla wszystkich dalszych wariantów architektur i funkcji błędu.

Rozpoczynając trening sieci ustalone wszystkie parametry funkcji błędu (4.8) na wartość 1. Dla zbioru uczącego przeprowadzano augmentację danych z wykorzystaniem filtracji uśredniającej (konwolucja z maską o wymiarze SxS o wartościach $\frac{1}{S^2}$), zakłóceń addytywnych (dodanie białego szumu z zakresu $[-\sigma; \sigma]$ z ograniczeniem wartości do przedziału $[0.0; 1.0]$), przeskalowania (powiększenie lub pomniejszenie obrazu) czy też rotacji (obrót obrazu o kąt z zakresu $[-45 \text{ deg}; 45 \text{ deg}]$). Do minimalizacji błędu użyto algorytmu *Adadelta*. Dla pozostałych architektur również przeprowadzano augmentację danych i używano tego samego algorytmu optymalizacji.

Proces uczenia przerwano, gdy nie zauważono zmian wartości błędu oraz metryki *IoU* dla zbioru uczącego oraz walidacyjnego. Finalnie architektura osiągnęła wartość współczynnika $IoU = 0.47$ dla zbioru walidacyjnego. Uzyskany wynik był niesatisfakcyjny, zdecydowano się na odrzucenie rozpatrywanej wersji architektury w dalszych rozważaniach. W trakcie wizualizacji zauważono, iż sieć poprawnie wskazuje element siatki, lecz gorzej radzi sobie regresją parametrów. Przyczyną mogła być zbyt rzadka siatka detekcji.

4.2. Architektura rozgałęziona

Architektura *SSD* wykorzystuje połączenia pochodzące z warstw pośrednich celem poprawienia rezultatów detekcji. Ze względu na niewielką wartość współczynnika *IoU*, wcześniej

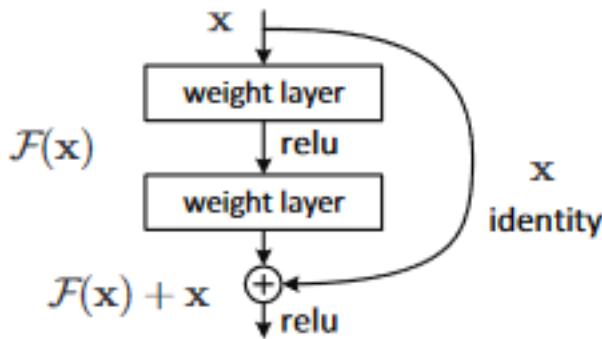


Rys. 4.2. Architektura rozgałęziona wykorzystująca konwolucje separowalne.

rozważaną architekturę zmodyfikowano o dodatkowe wyjście z sieci pochodzące z warstwy pośredniej zakończonego warstwą *YOLO* analogicznie jak w wersji nierożgałezionej (wymiary *anchor box* dla rozgałęzienia $a_w = 8$ oraz $a_h = 8$). Na rysunku 4.2 przedstawiono graficzną reprezentację architektury. Po przeprowadzeniu procesu uczenia sieci, z wykorzystaniem techniki transferu wiedzy z architektury pierwotnej, uzyskano pogorszenie wartości *IoU*. Z tego powodu odrzucono analizę również tej architektury. Przyczyną niepowodzenia może być zbyt mała liczba parametrów dla rozpatrywanego problemu. Ponadto w stosowanych funkcjach błędu dla części regresyjnej wymagano zerowych rezultatów (wartość referencyjną stanowiło 0) elementów siatki, do których nie przypisano obiektu (położenie różne od (row, col)). Zwiększa to stabilność działania sieci, jednakże wymaga również większej liczby pozytywnych detekcji z parametrami różnymi od wymiaru *anchor box* oraz położenia środka elementu siatki, co dla rozpatrywanego problemu detekcji pojedynczego obiektu nie jest możliwe.

4.3. Architektura Resnet18

Rezultaty dotychczas badanych architektur były niesatisfakcjonujące. Zdecydowano się zatem na zastosowanie rozwiązania o znacznie większej liczbie parametrów, a także wykorzystującego połączenia rezydualne (połączenia omijające wybrane następne warstwy przedstawione na rysunku 4.3) – *Resnet18*[44], tym samym sprawdzając czy bardziej rozbudowany model uzyska żądane rezultaty. W tym celu wykorzystano dwa pierwsze bloki rezydualne sieci *ResNet18* wraz z dodatkową warstwą konwolucyjną posiadającą 128 filtrów oraz warstwą *YOLOv3*. Celem przyspieszenia procesu uczenia wag bloków rezydualnych zostały zainicjalizowane wagami z sieci trenowanej na zbiorze *ImageNet*[24] (dostęp za pośrednictwem biblioteki *PyTorch*).



Rys. 4.3. Blok rezydualny. Przekształcenie identycznościowe omijające wybrane bloki konwolucji. Źródło: [44]

Wykorzystano tu trzy *anchor box* o wymiarach (szerokość x wysokość): 22×33 , 43×69 , 89×133 uzyskanych jako wynik algorytmu k-średnich. Uzyskana architektura podsiadała ponad 700 tys. parametrów. Wymiary obrazów wejściowych zredukowano do 360 pikseli szerokości oraz 180 pikseli wysokości. Funkcja błędu odpowiedzi sieci przyjmuje postać daną równaniem (4.9). Jako funkcję błędu regresji wybrano funkcję *GIoU* (3.24). Błąd wykrycia obiektu jest opisany przez funkcję entropii binarnej. Dla obu współczynników wagowych przyjęto wartość 1.

$$\text{loss}(v_{pred}, v_{ref}, A_{pred}, B_{ref}) = \lambda_{\text{validity}} \text{BCE}(v_{pred}, v_{ref}) + \lambda_{bbox} \sum_{a=0}^2 \text{GIoU}(A_{pred}(a), B_{ref}) \quad (4.9)$$

Poprzez v_{pred} oznaczono 3 kanały wykrycia obiektu dla każdego *anchor box*, v_{ref} stanowi maskę referencyjną. Wartość 1 znajduje się na pozycji (row, col) (wyznaczane z równań (4.7)-(4.6)) dla każdego *anchor box*. Predykcja parametrów obiektu dla elementu siatki (row, col) dla a -tego *bounding box* została oznaczona $A_{pred}(a)$ oraz wyznaczona za pomocą równań (3.20)-(3.23). Parametry obiektu referencyjnego oznaczono poprzez B_{ref} . Wyznaczany błąd jest obliczany względem grupy filtrów każdego z *anchor box* na tej samej pozycji w siatce.

Dla modelu zmienoprzecinkowego architektura osiągnęła wartość *IoU* ponad 0.8 dla zbioru walidacyjnego. Proces uczenia został przerwany podobnie jak poprzednio, gdy nie zauważano znaczących zmian błędu i metryki *IoU*.

Zdecydowano się na kwantyzację modelu z wykorzystaniem zapisu stałoprzecinkowego z wartościami dobranymi eksperymentalnie. Obraz wejściowy (znormalizowany do przedziału $[0.0; 1.0]$) został poddany kwantyzacji 8 bitowej bez znaku i bez części całkowitej. Wagi oraz wyjścia warstw pośrednich poddano kwantyzacji 8 bitowej ze znakiem i 2 bitami części całkowitej. Wartości te były dobrane eksperymentalnie. Ostatnie dwie warstwy poddano kwantyzacji 8 bitowej ze znakiem i 3 bitami części całkowitej (możliwość uzyskania skali przekształcającej *anchor box* do wymiarów obrazu). W przypadku każdej kwantyzacji zastosowano również

ograniczenie wartości wynikowej do limitów wynikających z zapisu w formacie stałoprzecinkowym. Tak dobrane parametry kwantyzacji mają również pełnić zadanie pewnego rodzaju regularizacji.

Po zakończeniu treningu sieci z kwantyzacją osiągnięto wartość $IoU = 0.72$. Uzyskana wartość pozwalałaby na osiągnięcie maksymalnej wartości oceny danej równaniem (2.2) (jeżeli uzyskano by podobny rezultat na zbiorze tajnym).

Rozważana architektura wydaje się być trudniejsza do sprzętowej implementacji ze względu na połączenia rezydualne oraz znaczne rozmiary. Jednakże analizując przeprowadzone procesy uczenia można zauważyc, iż dodanie warstw normalizujących oraz zastosowanie funkcji błędu bazującej na metryce IoU pozwala na osiągnięcie większych wartości IoU , przy mniejszej liczbie epok treningowych (dla poprzednich architektur liczba epok treningowych była znaczco większa, niż dla architektury rezydualnej).

4.4. LittleNet

Podsumowując wnioski z rozważanych dotychczas architektur można stwierdzić, iż:

- zastosowanie konwolucji separowalnych pozwala na znaczną redukcję liczby parametrów oraz złożoności obliczeniowej,
- warstwy normalizacji pozwalają zmniejszyć liczbę epok treningowych,
- funkcja błędu bazująca na matryce IoU pozwala osiągnąć lepsze wyniki.

Dodatkowo można zauważyc, iż konwolucja separowalna dla pierwszej warstwy posiada jedynie trzy filtry DW . Zatem następna warstwa PW dokonuje podziału w przestrzeni zaledwie trójwymiarowej. Może to skutkować utratą znaczącej części informacji już na początkowym etapie przetwarzania, a także redundancją filtrów PW (w przypadku większej ich liczby). Zastosowanie pełnej konwolucji w pierwszej warstwie mogłoby polepszyć jakość detekcji. Jednakże kownolucje pełne są bardziej złożone w implementacji sprzętowej. Z tego względu zdecydowano się na zastosowanie wielokrotnej filtracji DW . Wówczas każdy kanał wejściowy jest filtrowany nie jednym, a m niezależnymi filtrami. Pozwala to na zwiększenie liczby cech kontekstowych bezpośrednio ekstrahowanych z obrazu (czy też z każdego kanału) wraz z zachowaniem możliwości stosunkowo łatwej implementacji sprzętowej.

Proponowaną architekturę przedstawiono na rysunku 4.4. Projekt architektury powstał wykorzystując architektury *SkyNet* [43], poprzez dodanie dodatkowej warstwy konwolucji oraz eksperymentalnie dobranych wymiarów warstw, mając na uwadze redukcję liczby parametrów architektury wyjściowej. Sieć zawiera 7 bloków z konwolucją DW oraz 7 z konwolucją PW . Po każdej konwolucji występuje warstwa normalizacyjna oraz funkcja aktywacji *ReLU*. Pierwszy

blok zawiera warstwę DW z pięciokrotną filtracją kanałów, kolejne bloki zawierają już tylko dwukrotną filtrację kanałów. Po pierwszych 4 warstwach PW występuje warstwa *Max Pooling*. Ostatnią warstwą stanowi konwolucja PW stanowiąca warstwę *YOLOv3*. Wymiary *anchor box* pozostały takie same jak dla architektury wykorzystującej bloki rezydualne, lecz przeskalowane do wymiarów wejścia sieci. Proponowana architektura posiada niespełna 134 tys. parametrów, co stanowi znaczącą redukcję liczby parametrów w stosunku do *SkyNet* (powyżej 300 tys.) oraz *Ultra_net* (ponad 200 tys.). Architekturze nadano nazwę *LittleNet (LN)*.

Rozmiar wejścia sieci ustalono na 360 pikseli szerokości oraz 180 pikseli wysokości. Funkcja błędu odpowiedzi sieci (4.9) rozszerzono o regularyzację normą L_1 (średnia wartość bezwzględna) parametrów sieci p oraz uogólnienie błędu regresji funkcją IoU_{based} , uzyskując równanie (4.10).

$$\begin{aligned} loss(v_{pred}, v_{ref}, A_{pred}, B_{ref}, p) = & \lambda_{validity} BCE(v_{pred}, v_{ref}) \\ & + \lambda_{bbox} \sum_{a=0}^2 IoU_{based}(A_{pred}(a), B_{ref}) \\ & + \lambda_{reg} L_1(p) \end{aligned} \quad (4.10)$$

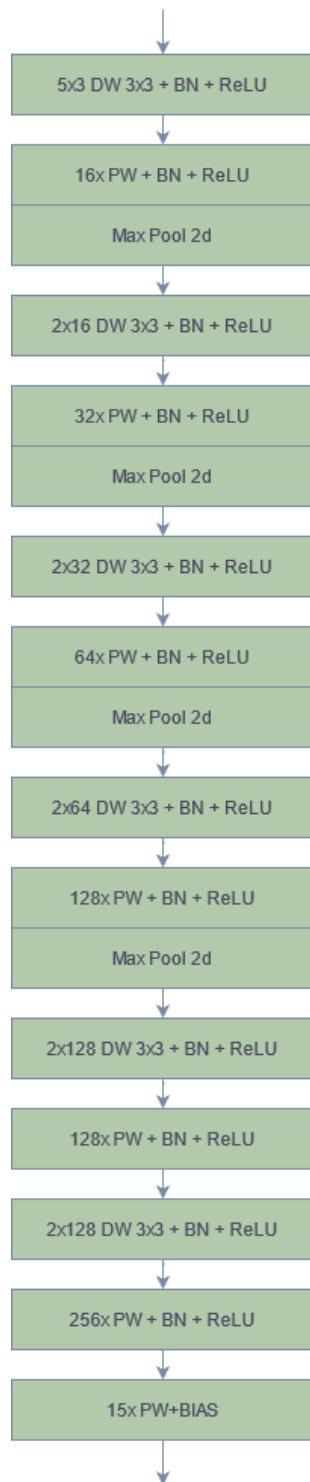
Za funkcję błędu regresji IoU_{based} przyjęto funkcję *GCIoU*(ang. *Generalized Complete IoU*) daną równaniem (4.11). Funkcja ta bazuje na *GIoU*[36], *DIoU*[35] oraz *CIoU*[35]. Połączenie powyższych funkcji pozwala na połączenie ich zalet funkcji wraz z eliminacją wad:

- $giou_{part}$ - bardziej bezpośredni wpływ na parametry oraz poprawne skalowanie (gdy $IoU = 0$) (3.27),
- $diou_{part}$ - centralizacji predykcji (3.28),
- $ciou_{part}$ - wzmacnione finalne skalowanie rozmiaru (3.31).

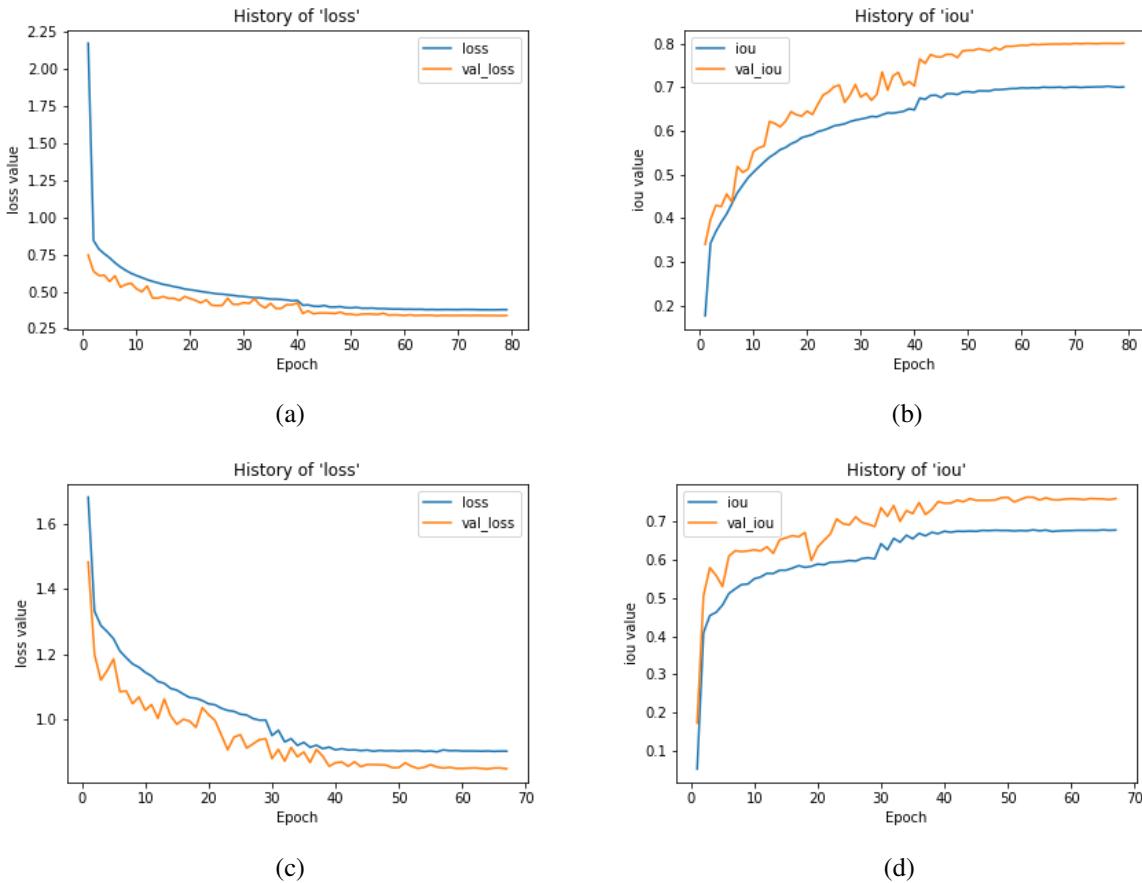
$$\begin{aligned} GCIoU(A, B) = & 1 - IoU(A, B) + giou_{part}(A, B) \\ & + diou_{part}(A, B) + ciou_{part}(A, B) \end{aligned} \quad (4.11)$$

Ustalono parametry wagowe na $\lambda_{validity} = 20$, $\lambda_{bbox} = 1$ oraz $\lambda_{reg} = 0.01$ (bazując na rozwiązaniu [42]). Krok ucący został początkowo ustalony jako $l_r(0) = 1$. W trakcie procesu uczenia podlegał on modyfikacji zgodnie z równaniami (4.12) oraz (4.13). Poprzez t oznaczono numer epoki uczącej, natomiast $loss(t)$ oznacza średnią wartość funkcji błędu na zbiorze uczącym po epoce t . Tak przeprowadzana zmiana kroku uczącego pozwala na dokonywanie większych zmian wag, gdy możliwe było zmniejszenie błędu oraz mniejsze zmiany, gdy wartość błędu wzrosła.

$$l_r(t+1) = \begin{cases} 1.3 * l_r(t), & \text{if } loss(t) < loss(t-1) \\ 0.5 * l_r(t), & \text{if } loss(t) > loss(t-1) \\ l_r(t), & \text{otherwise} \end{cases} \quad (4.12)$$



Rys. 4.4. Proponowana architektura sieci LittleNet.



Rys. 4.5. Przebiegi wartości błędu oraz metryki IoU (w zależności od epoki uczącej) dla modelu zmiennoprzecinkowego 4.5a i 4.5b oraz 4.5c i 4.5d.

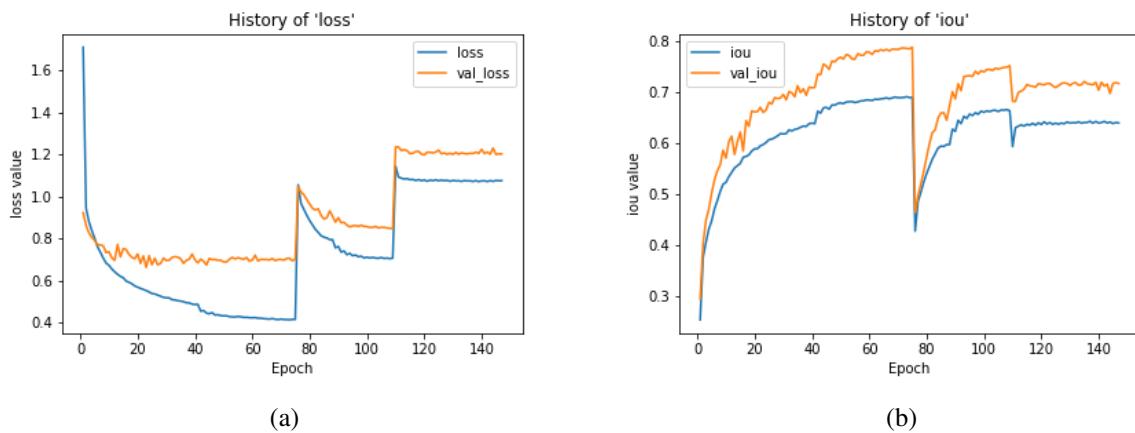
$$l_r(t+1) = \max(10^{-5}, \min(1, l_r(t+1))) \quad (4.13)$$

Po zakończeniu procesu uczenia modelu zmiennoprzecinkowego osiągnięto wartość $IoU = 0.8$ dla zbioru walidacyjnego oraz 0.7 dla zbioru uczącego. Powodem tak znacznej różnicy wartości metryki jest znaczny stopień augmentacji danych.

Następnie sieć poddano kwantyzacji. Dane wejściowe zostały poddane kwantyzacji 8 bitowej bez znaku i bez części całkowitej. Warstwy pośrednie poddano kwantyzacji 8 bitowej ze znakiem i 2 bitami części całkowitej. Ostatnią warstwę poddano kwantyzacji 8 bitowej ze znakiem i 3 bitami części całkowitej.

Uzyskano wartość $IoU = 0.76$ dla zbioru walidacyjnego oraz 0.67 dla zbioru uczącego. Na rysunkach 4.5a-4.5d przedstawiono przebiegi funkcji błędu oraz metryki IoU dla modelu zmiennoprzecinkowego oraz kwantyzowanego.

W trakcie realizacji implementacji sprzętowej, ustalono, iż dla obecnego rozmiaru wejścia sieci nie jest możliwe buforowanie wyników warstw pośrednich w pamięci BRAM. Z tego względu zdecydowano się na redukcję rozmiaru wejścia sieci do 200 pikseli szerokości oraz 100 pikseli wysokości. Rozpoczęto trening sieci z takimi samymi parametrami (wagi sieci zostały



Rys. 4.6. Przebiegi wartości błędu 4.6a oraz metryki IoU 4.6b (w zależności od epoki uczącej) dla modelu o rozmiarze wejścia 200x100 pikseli. Epoka 75 – rozpoczęcie kwantyzacji bez warstw normalizacyjnych. Epoka 109 – pełna kwantyzacja.

wylosowane). Uzyskano wartość $IoU = 0.78$ dla zbioru walidacyjnego oraz 0.69 dla zbioru uczącego.

Eksperymentalnie stwierdzono również, iż parametry warstw normalizacyjnych po przekształceniu do formy afinicznej uzyskują w większości przypadków wartości przekraczające zadany zakres wartości wynikający z dotychczasowej kwantyzacji. Względem poprzedniego modelu zmieniono liczbę bitów części całkowitej: 1 dla wyjścia oraz 3 dla wag warstw pośrednich. Od epoki 75 rozpoczęto kwantyzację z wyłączeniem warstw normalizujących. Następnie od epoki 109 kwantyzacji poddano również warstwy normalizujące. Dla modelu w pełni kwantyzowanego uzyskano wartość $IoU = 0.71$ dla zbioru walidacyjnego oraz 0.64 dla zbioru uczącego. Na rysunkach 4.6a-4.6b przedstawiono przebiegi funkcji błędu oraz metryki IoU dla modelu o zmniejszonych rozmiarach wejścia.

Poprzez analizę przeprowadzonych badań architektury stwierdzono, iż stosowanie konwolucji separowalnych posiada zaletę w postaci stosunkowo niewielkiej liczby parametrów, lecz również wadę wynikającą ze stosunkowo niewielkiego stopnia ekstrakcji cech z danych wejściowych (w szczególności w warstwach początkowych). Problem ten rozwiązano poprzez wielokrotną filtrację filtrami *depthwise*. Ponadto zastosowanie warstw normalizacyjnych pozwoliło na przyspieszenie procesu uczenia. Wykorzystując wnioski z przeprowadzonych badań zaprojektowano architekturę *LittleNet*. Jako funkcję błędu regresji zaproponowano własne rozwiązanie (*GCIoU*) bazującą na już istniejących. Osiągnięto dokładność $IoU = 0.71$ wraz z możliwą stosunkowo prostą implementacją sprzętową. Poza przedstawionymi przeprowadzono wiele innych eksperymentów, których rezultaty nie pozwalały osiągnąć większych wartości IoU . Na tej podstawie oraz na podstawie dotychczasowych wyników konkursu ($IoU \leq 0.74$) stwierdzono,

iż uzyskanie wartości wyższych jest stosunkowo trudne dla niewielkich architektur implementowanych sprzętowo. Z tego względu zaakceptowano uzyskany rezultat jako zadawalający.

5. Implementacja programowa oraz sprzętowa wybranej architektury sieci

Opracowanie odpowiedniej architektury sieci oprócz rozoważań teoretycznych, wymagało również wykonania szeregu eksperymentów obliczeniowych. Ponadto konieczna była implementacja sprzętowa modelu celem akceleracji obliczeń zaprojektowanej architektury na wybranej platformie obliczeniowej.

5.1. Implementacja programowa

Implementacja programowa została zrealizowana w ramach framework'a *PyTorch* [8]. Wykorzystano również bibliotekę *Brevitas* [7] do realizacji odpowiedniej metody kwantyzacji. Metyda implementacji kwantyzacji opiera się o klasę *ExtendedInjector*. Pozwala ona na "wstawienie" operacji kwantyzacji wag bezpośrednio przed zastosowaniem ich w obliczeniach. Ponadto biblioteka rozszerza predefiniowane w *PyTorch* warstwy o operacje kwantyzacji. W wersji kwantyzowanej dostępne są m.in. warstwa konwolucji, *Max Pooling* czy funkcja *ReLU*. Kwantyzowana warstwa normalizacji została zdefiniowana jako przekształcenie afiniczne bez aproksymacji średniej i wariancji. Wymagało to implementacji własnej warstwy dokonującej wyboru pomiędzy warstwą zmienoprzecinkową oraz jej wersją kwantyzowaną. Augmentacja danych została zrealizowana w oparciu o bibliotekę *OpenCV* [45]. Obliczenia były wykonywane na platformie *GoogleColab* [46] z wykorzystaniem *GPU*. Ze względu, iż dzienny czas użytkowania usługi jest ograniczony wymagane było przechowywanie stanu procesu ucznia w plikach zapisywanych poprzez usługę *GoogleDrive*.

Dla celu testowania wyników implementacji sprzętowej poszczególnych warstw został zaimplementowany dodatkowy model programowy (dwóch typów warstw). Wykorzystywał on obliczenia całkowitoliczbowe do wykonania operacji konwolucji *DW* oraz *PW*. Ponadto zaimplementowano także konwersję wag zmienoprzecinkowych do stałoprzecinkowych wymaganych do inicjalizacji pamięci ROM akceleratorów opisanych w dalszej części.

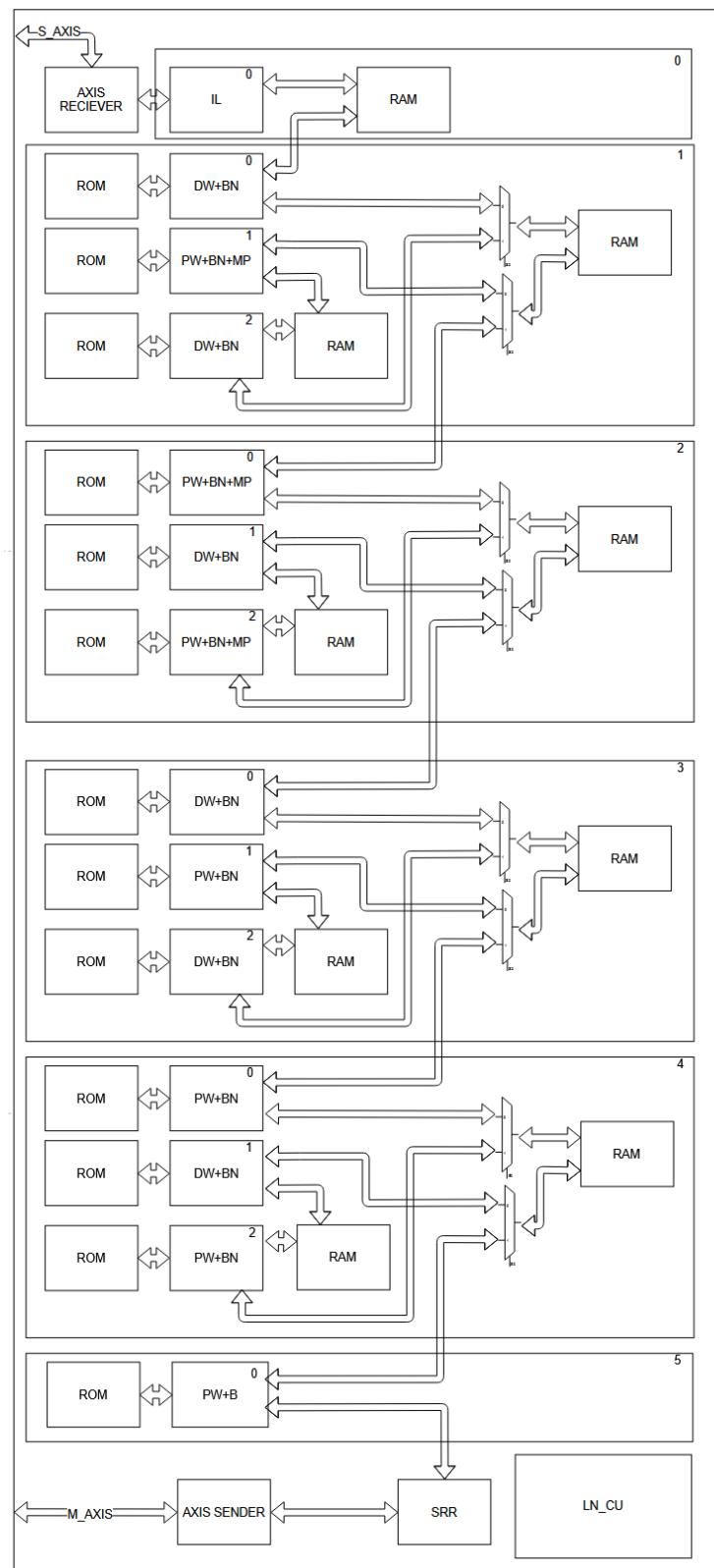
5.2. Implementacja sprzętowa

Wstępnie planowano implementacje sprzętową z wykorzystaniem narzędzia *FINN* 2.3. Jednakże ze względu na braki w dokumentacji (np. implementacja własnej metody kwantyzacji w *Brevitas*), a także brak możliwości akceleracji warstwy *BN* (przynajmniej bezpośrednio) zdecydowano się na implementację własną. Początkowo w języku wysokopoziomowym *HLS*, lecz ostatecznie w języku opisu sprzętu *System Verilog*. Akceleracja całej sieci stanowi architekturę potokową gruboziarnistą (ang. *coarse grained*). Schemat przedstawiono na rysunku 5.1. Na elementy wspomnianej architektury składają się:

- bloki komunikacji - realizują wymianę danych z *DMA* (ang. *Direct Memory Access*) [47] poprzez magistralą *AXI4-Stream* [48]. Blok *AXIS RECEIVER* odbiera i przekształca strumień danych do postaci wymaganej na dalszym etapie. Wysyłanie danych odbywa się poprzez blok *AXIS SENDER*.
- *IL* (ang. *Input Layer*) - warstwa wejściowa zmiana formatu danych wejściowych.
- *ROM*(ang. *Read-Only Memory*) - przechowywanie wag poszczególnych warstw.
- *RAM*(ang. *Random Access Memory*) - przechowywanie danych wejściowych oraz rezultatów poszczególnych akceleratorów.
- multipleksery - zarządzanie dostępem do pamięci.
- *DW, PW* - bloki akceleratorów warstw *depthwise* oraz *pointwise* w zadanych konfiguracjach (*BN, MP, B* - odpowiednio warstwa normalizacyjna, *Max Pooling*, bias).
- *SSR* - rejestr szeregowo-równoległy (ang. *Serial-Parallel Register*) realizuje odbiór danych z ostatniej warstwy.
- *LN_CU* (ang. *LittleNet Control Unit*) - logika sterująca procesem akceleracji.

Poszczególne akceleratory pogrupowane są po (co najwyżej) 3 w bloku. Na jeden blok przypisane są dwa bloki pamięci *RAM*, w tym jeden z dostępem multipleksowanym. Ma to na celu ograniczenie wymaganej łącznej pamięci, kosztem zmniejszenia przepustowości. Proces akceleracji odbywa się według schematu prezentowanego w tabeli 5.1. Zostało to zrealizowane w postaci maszyny stanów. W danym stanie 0-3 aktywne są akceleratory 0-2 z bloków 0-5, dla których występuje wartość 1. W tabeli 5.1 oznaczono również typy każdego akceleratora. Pozwala to uzyskać ciąg operacji przetwarzania danych tożsamy z architekturą sieci. W jednej chwili są aktywne akceleratory co czwartej warstwy. Jest to wymagane, aby udzielić dostępu do wybranych bloków pamięci kilku akceleratorom.

Wyróżniono 3 typy akceleratorów (dostępnych w kilku konfiguracjach):



Rys. 5.1. Schemat akceleracji architektury LittleNet. Uwzględniono jedynie najistotniejsze elementy.

Tabela 5.1. Schemat aktywacji akceleratorów zrealizowany jako maszyna stanów.

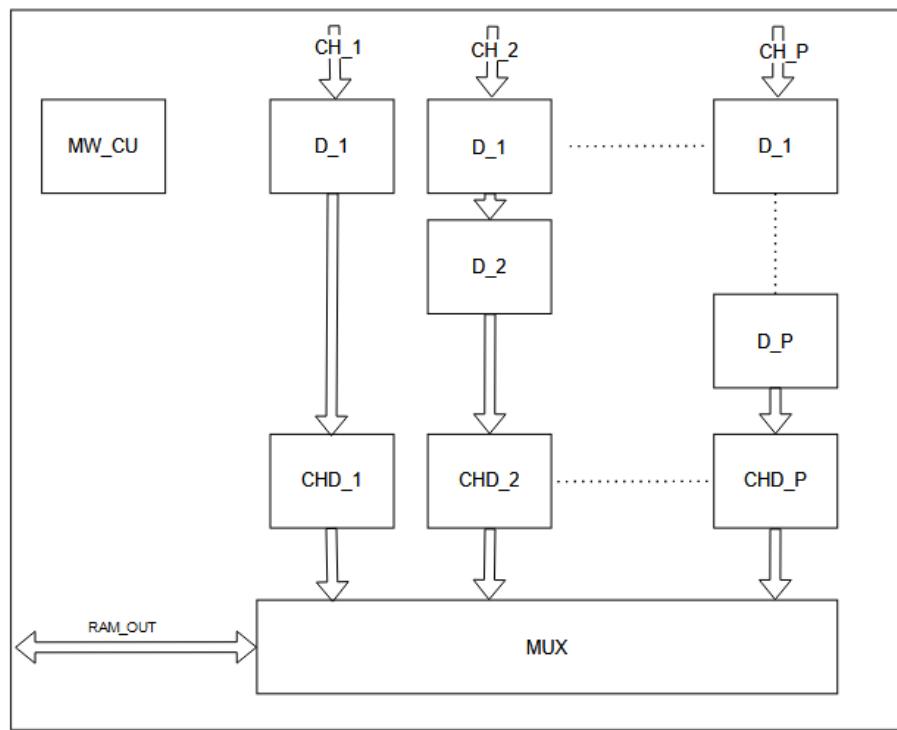
Blok	0	1			2			3			4			5	
Akcelerator	0	0	1	2	0	1	2	0	1	2	0	1	2	0	Następny Stan
Typ	IL	DW	PW	PW											
Stan															
0	1		1				1				1				1
1				1				1				1			2
2					1				1				1		3
3		1				1				1				1	0

- warstwa wejściowa (IL),
- warstwa *depthwise* (DW),
- warstwa *pointwise* (PW).

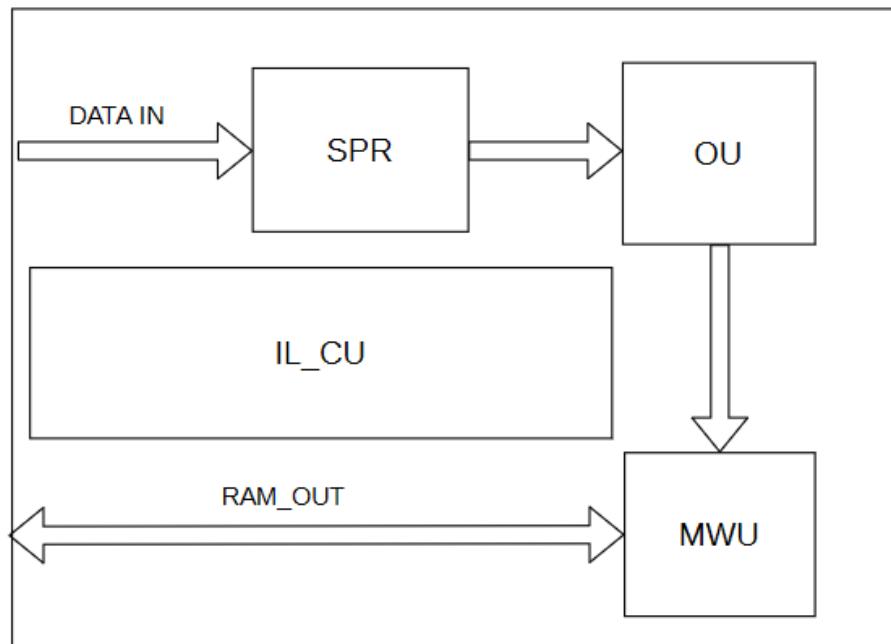
Akceleratory warstw *PW* pozwalają na równoległe wykonanie operacji. Jednakże wymagany jest zapis do jednej pamięci w sposób szeregowy. Do tego celu wykorzystywany jest moduł *MWU* (ang. *Memory Writer Unit*) przedstawiony na rysunku 5.2. Wykonuje on operacje analogiczne do rejestru równoległo-szeregowego (o P kanałach wejściowych). Równoległe kanały wejściowe $CH_i P$ są odpowiednio opóźniane. Każdy kanał CH_i przechodzi przez linię opóźniającą D o latencji równej indeksowi sygnału i . Do każdego kanału przypisany jest licznik adresu CHD_i . W przypadku, gdy opóźniona wartość kanału jest uznawana za ważną (ang. *valid*) następuje inkrementacja adresu. Wszystkie opóźnione kanały są multipleksowane przez moduł *MUX*, wybierając kolejne kanały. Wartość wybranego kanału wraz z opowiadającym adresem stanowią interfejs pamięci docelowej. Ponadto dodatkowa logika sterująca została przedstawiona poprzez blok *MW_CU*(ang. *Memory Writer Control Unit*).

5.2.1. Warstwa wejściowa

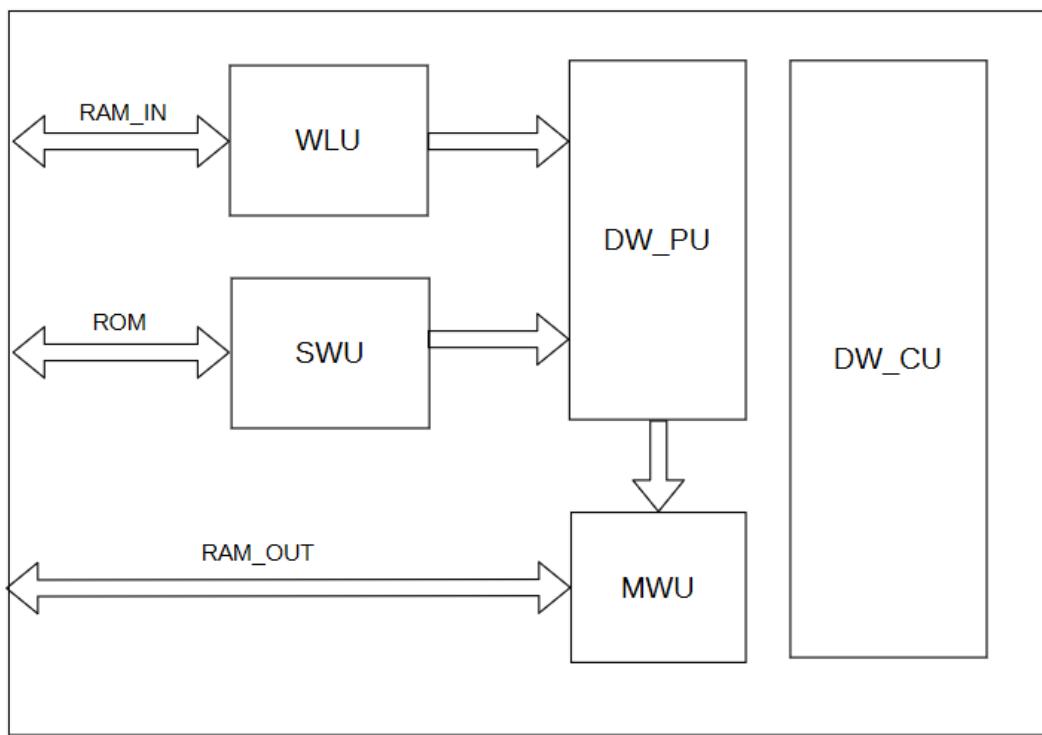
Warstwa wejściowa *IL* (ang. *Input Layer*) realizuje zadanie zmiany formatu danych. Schemat akceleratora został przedstawiony na rysunku 5.3. Dane wejściowe otrzymywane są w notacji *H-W-CH* (ang *Height-Width-Channel*). Na dalszym etapie przetwarzania wymagany jest format *CH-H-W*. Realizowane jest to poprzez buforowanie 3 kolejnych 32 bitowych pakietów danych w rejestrze szeregowo-równoległym *SPR*. Zgromadzone dane są rozdzielane na poszczególne kanały *R-G-B* (ang. *Red-Green-Blue*) składając się na 32 bitowe pakiety po jednym na każdy kanał. Uzyskany podział jest zapisywany do pamięci poprzez moduł *MWU* dla 3 kanałów.



Rys. 5.2. MWU – schemat generowania adresu dla przetwarzania wielokanałowego.



Rys. 5.3. Schemat warstwy wejściowej IL.

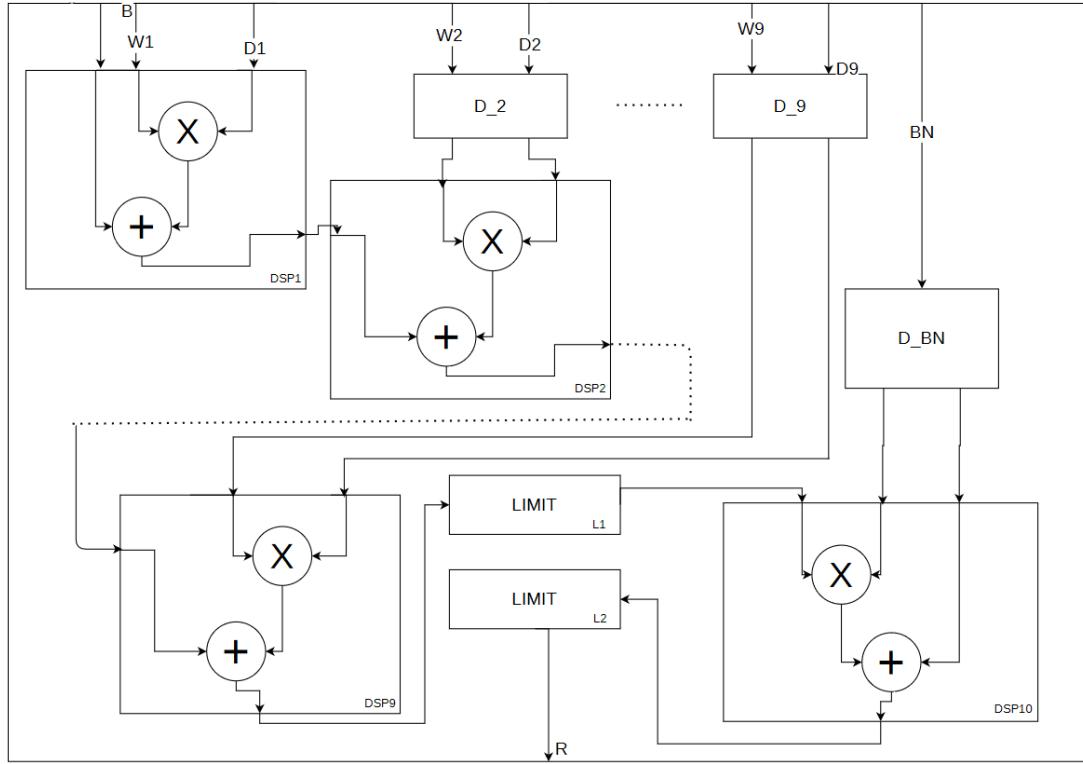


Rys. 5.4. Schemat akceleratora DW dostępnego w wielu konfiguracjach.

5.2.2. Warstwa *depthwise* (DW)

Akceleracja konwolucji DW jest realizowana razem z warstwą normalizującą. Na rysunku 5.4 przedstawiono schemat akceleratora DW. Obliczenia wykonywane są w architekturze po-tokowej drobnoziarnistej (ang. *fine grained*). W tym celu moduł SWU (ang. *Sliding Window Unit*) generuje strumień danych wraz z odpowiednimi opóźnieniami tak, aby uzyskać kontekst okna przesuwnego o wymiarach 3x3. Wykonanie operacji konwolucji wymaga wcześniejszego (przed rozpoczęciem strumieniowania każdego kanału) wczytania wag maski konwolucji. Operację tę wykonuje moduł WSL (ang. *Weights Loading Unit*). Sama operacja iloczynu skalarnego wektora wag i kontekstu jest wykonywana przez DW_{PU} (ang. *DepthWise Processing Unit*) przedstawiony na schemacie 5.5. Wykorzystano tutaj możliwość kaskadowego połączenia kolejnych 9 DSP. Dostępna jest konfiguracja wykorzystująca składową bias konwolucji (*B*) oraz normalizację poprzez przekształcenie afiniczne (*BN*). Wynik konwolucji oraz normalizacji jest ograniczany (*LIMIT*) do wartości wynikających z wybranej notacji stałoprzecinkowej, czy też zastosowania funkcji *ReLU*.

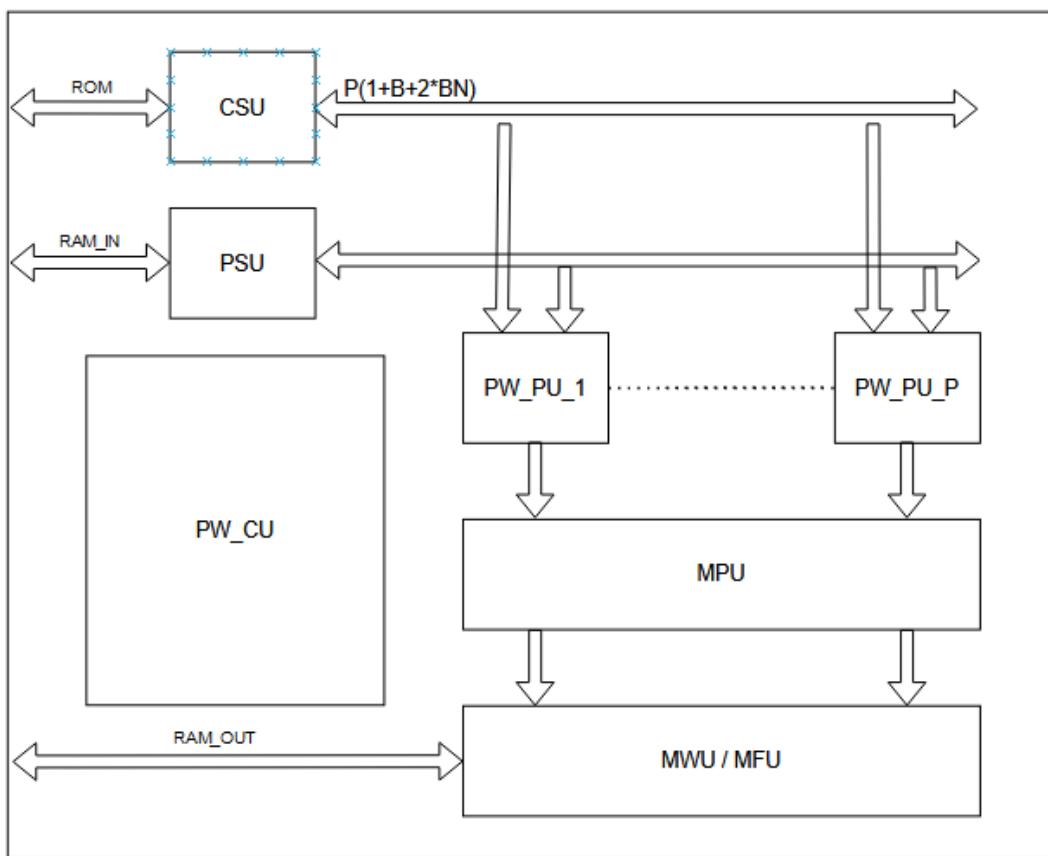
Uzyskany rezultat jest zapisywany pod adresem wyznaczonym przez MWU. Sterowanie procesem akceleracji odbywa się poprzez logikę reprezentowaną przez DW_{CU} (ang. *Depth-Wise Control Unit*). Wykonanie wielokrotnej konwolucji DW odbywa się poprzez wielokrotne (cykliczne) generowanie strumienia danych wejściowych przez SWU.



Rys. 5.5. Schemat *DW_PU* – kaskadowe połączenie *DSP*. Bloki *DSP10+L2* są opcjonalne zależnie od konfiguracji.

5.2.3. Warstwa *pointwise (PW)*

Podobnie jak w poprzednim przypadku akcelerator konwolucji *PW* dostępny jest w wielu konfiguracjach. Schemat akceleratora *PW* prezentuje rysunek 5.6. Operacja konwolucji *PW* nie wymaga gromadzenia otoczenia rozpatrywanego punktu, lecz iterowania po kolejnych kanałach wejściowych w danym punkcie. Jest to realizowane przez moduł *PSU* (ang. *Point Streamer Unit*). Dla każdego kanału rozważanego punktu jest wymagana odpowiednia waga. Ponadto identyczna sekwencja wag jest powtarzana dla następnych punktów. W tym celu moduł *CSU* (ang. *Cyclic Streamer Unit*) generuje strumień w sposób cykliczny. Następuje wielokrotny odczyt z kolejnych adresów zadanej puli adresowej definiowanej przez liczbę wag filtra. Przed rozpoczęciem generowania strumienia danych, odczytywane są wagi bias oraz przekształcenia aficznego, które następnie są przechowywane w odpowiednich rejestrach. Obliczenie konwolucji dla kolejnych punktów jest realizowane przez moduł *PW_PU* (ang. *PointWise Processing Unit*) (rysunek 5.7). Wykorzystuje się do tego celu operacje akumulacji z mnożeniem. Zakumulowana suma iloczynów jest (zależnie od konfiguracji) powiększana o wartość bias. Uzyskany rezultat zostaje ograniczony (identycznie jak w konwolucji *DW*). W następnym kroku wykonywana jest operacja normalizacji wraz z ponownym ograniczeniem wartości.

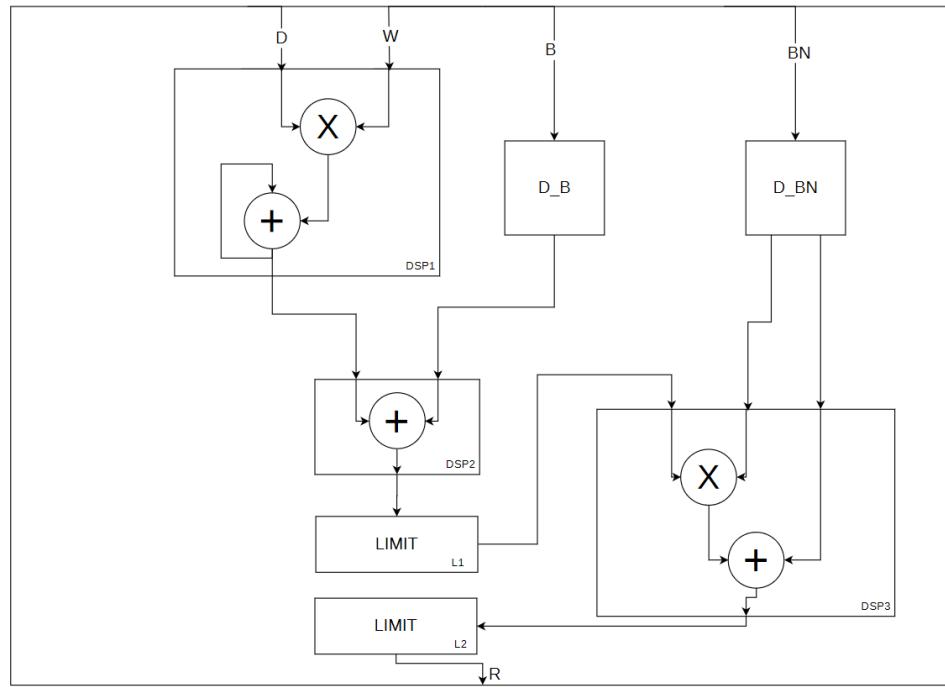


Rys. 5.6. Schemat akceleratora *PW* dostępnego w wielu konfiguracjach.
Opcjonalne są moduły *MPU* oraz *MWU* lub *MFU*.

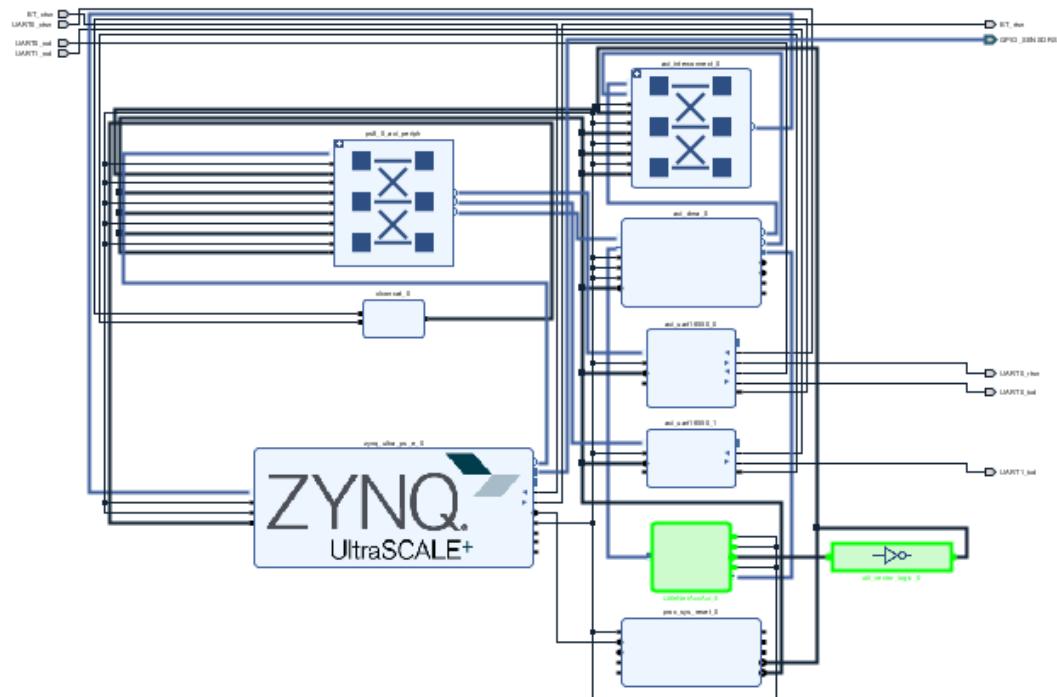
Następnym etapem przetwarzania jest (ewentualna) operacja *Max Pooling*. Do tego celu wykorzystuje się odpowiednie opóźnienia oraz porównywania wartości. Uzyskany rezultat jest zapisywany pod adresem wyznaczanym przez moduł *MWU*. Opcjonalną funkcjonalnością (dla warstwy *YOLOv3*) jest zastąpienie *MWU* przez *MFU* (ang. *Max Finder Unit*) pozwalającego znaleźć punkt o największej wartości (funkcja *argmax*) wykrywalności obiektu, a także zwrócić jego parametry (funkcja *max*). Ponadto istnieje możliwość zrównoleglenia obliczeń poprzez wykorzystanie P modułów *PW_PU*. Wymaga to zastosowania pamięci ROM P -krotnie szerszej. Sterowanie poszczególnymi modułami jest zaimplementowane w bloku *PW CU* (ang. *PointWise Control Unit*).

5.2.4. Sterowanie akceleracją

Zaimplementowany akcelerator sieci *LittleNet* został dołączony do schematu blokowego dostarczonego przez organizatorów konkursu. Zmodyfikowany schemat przedstawiono na rysunku 5.8.



Rys. 5.7. Schemat PW_PU – akumulacja z mnożeniem realizowane przez DSP. Bloki DSP2 oraz DSP3+L2 są opcjonalne i zależne od konfiguracji.



Rys. 5.8. Schemat blokowy w środowisku Vivado 2019.1. Zaznaczono dodane elementy – akcelerator oraz negacja sygnału *reset*.

Uzyskaną modyfikację poddano syntezie, implementacji oraz wygenerowano plik *bit* w środowisku *Vivado 2019.1* dostarczonym przez firmę *Xilinx*. Otrzymane rezultaty (pliki **.hwh* oraz **.bit*) umieszczone w odpowiednio skonfigurowanym środowisku na platformie *Avnet Ultra96 V2*. Aplikacje sterującą zrealizowano w postaci notatnika *Jupyter*. Konfiguracja logiki programowej oraz przesył danych do oraz z akceleratora zostały zrealizowane z użyciem biblioteki *PYNQ*. Odczyt obrazów oraz przetwarzanie rezultatów zaimplementowano w sposób równoległy do procesu akceleracji. Analizując dokładniej schemat akceleracji można zauważyć, iż rezultat przetwarzania danego obrazu jest uzyskiwany po 4 cyklach przetwarzania. Wymaga to odrzucenia początkowych rezultatów oraz wykonania dodatkowych cykli akceleracji, aby uzyskać poprawny wynik dla wszystkich obrazów wejściowych.

5.3. Podsumowanie

Przeprowadzony proces implementacji wymagał na początkowym etapie wyznaczenia modelu programowego. W tego celu wykorzystano bibliotekę *PyTorch* oraz *Brevitas*. Pierwszy etap wymagał wytrenowania modelu zmiennoprzecinkowego. Uzyskany model zastał następnie poddany odpowiedniej kwantyzacji. Zrealizowanie sprzętowej akceleracji wymagało opracowania sposobu implementacji poszczególnych warstw, jak i całej sieci. Wyszczególniono 3 typy akceleratorów: warstwa wejściowa, *DW* oraz *PW*. Pierwsza z wymienionych dokonuje zmiany formatu danych wejściowych. Pozostałe dwie warstwy realizują odpowiednie konwolucje wraz z normalizacją. Warstwa *PW* możliwa jest do konfiguracji realizującej operacje *Max Pooling* czy też funkcji *argmax* oraz *max*. Możliwe jest także dodatkowe zrównoleglenie poprzez wykorzystanie większej liczby modułów *PW_PU*. Akceleracja całej sieci oparta jest o architekturę potokową gruboziarnistą z opóźnieniem 4 cykli przetwarzania. Zastosowanie akceleratora do przetwarzania obrazów wymagało implementacji z wykorzystaniem notatnika *Jupyter* możliwego do modyfikacji w trakcie ewaluacji.

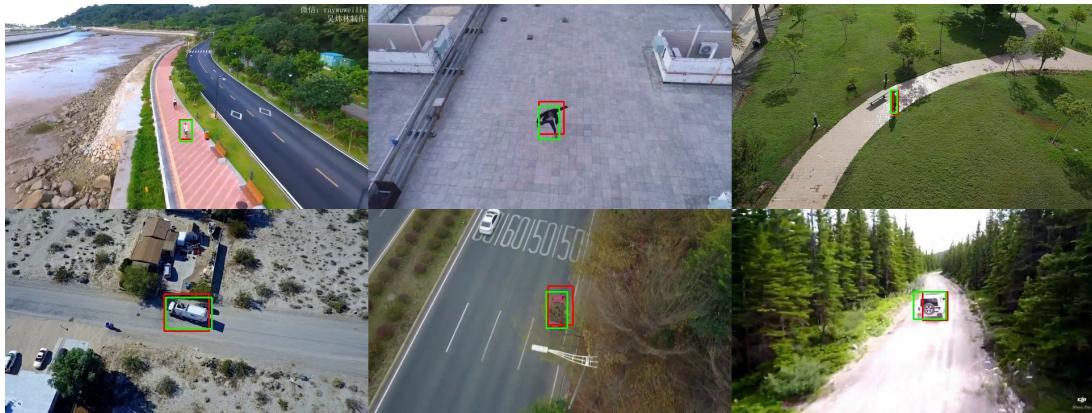
6. Ewaluacja sprzętowa i optymalizacja

W rozdziale 4.4 zaproponowano model sieci *LittleNet*. Architektura ta osiąga wartość $IoU = 0.71$ (dla zbioru testowego). Sieć ta została następnie zaimplementowana sprzętowo co omówiono w rozdziale 5. Rozwiążanie to jednak należy sprawdzić pod kątem dokładności detekcji (poprawności z modelem programowym), przepustowości oraz zużycia energii, a także dokonać próby ich optymalizacji.

6.1. Ewaluacja

W celu sprawdzenia zaprojektowanego rozwiązania wykorzystano zaimplementowaną aplikację sterującą (rozdział 5.2.4). Aplikacja pozwala na pomiar czasu przetwarzania oraz pomiar energii. Aby możliwe było sprawdzenie dokładności otrzymanych rezultatów zaimplementowano funkcję obliczającą wartość metryki IoU względem wartości referencyjnych.

Wstępne wyniki ewaluacji dawały inne rezultaty niż model programowy. W trakcie analizy stwierdzono, iż model programowy wykorzystywał operacje zmienoprzecinkowe do przeskakowania rozmiaru obrazu podawanego na wejścia sieci. Wykorzystanie do tego celu liczb całkowitych nieznacznie pogorszyło wynik. Finalnie oba modele dawały identyczne rezultaty w postaci $IoU = 0.7015$. Wykorzystano tutaj ośmiokrotne zrównoleglenie warstw *PW* pozwalające na osiągnięcie średnich wartości przepustowości $fps = 52.5$ oraz zużycia energii $e = 3652J$ (wartość dla 52500 obrazów). Uzyskana dokładność oraz przepustowość pozwalają na osiągnięcie maksymalnych wartości funkcji (2.2) oraz (2.3). Jednakże uzyskana wartość IoU przekracza wartość progową jedynie w niewielkim stopniu. Na rysunku 6.1 przedstawiono rezultaty detekcji dla wybranych obrazów wchodzących w skład zbioru testowego.



Rys. 6.1. Uzyskane rezultaty detekcji dla wybranych obrazów zbioru uczącego. Źródło: [1].

6.2. Optymalizacja

Zakładając, iż otrzymana dokładność detekcji została by osiągnięta na zbiorze tajnym, optymalizacja może zostać ograniczona jedynie zmianą parametrów akceleracji, takich jak zrównoleglenie p warstw PW czy częstotliwość zegara f dla logiki programowalnej (dotychczas używana była częstotliwość 100 MHz). Łączną moc układu można wyrazić wzorem (6.1), gdzie $P_{PL}(p, f)$ to moc logiki programowalnej oraz $P_{PS}(p, f)$ moc układu procesorowego.

$$P(p, f) = P_{PS} + P_{PL}(p, f) \quad (6.1)$$

Zużycie energii e układu potrzebnej do przeprowadzenia procesu detekcji N obrazów osiągające przepustowość $fps(p, f)$ definiuje równanie (6.2).

$$e(p, f) = \frac{N}{fps(p, f)}(P_{PS} + P_{PL}(p, f)) \quad (6.2)$$

Ponadto przybliżoną moc $P_{PL}(p, f)$ wyrazić można przez (6.3), zakładając liniowy przyrost mocy od p dla warstw PW oraz zależność mocy od częstotliwości przyjmując za $\beta(f)$. Przez P_{DW} oraz P_{PW} oznaczono moc akceleratorów (wszystkich warstw danego typu) odpowiednio DW i PW (dla akceleracji z użyciem tylko jednego PW_PU).

$$P_{PL}(p, f) = \beta(f)(P_{DW} + pP_{PW}) \quad (6.3)$$

Przybliżony stosunek czasu trwania obliczeń warstwy DW do czasu trwania całego cyku przetwarzania wyrażono poprzez $\alpha(p)$. Wartości stałe stanowią liczbę maksymalnych wymaganych odczytów z pamięci RAM dla warstw PW – 10 444 800 (stan 1 i 3) oraz DW – 460 000 (stan 2 i 4).

$$\alpha(p) = \frac{460000}{460000 + \frac{10444800}{p}} = \frac{p}{p + 22.7} \quad (6.4)$$

Z równania (6.4) można stwierdzić, iż znaczna część czasu przetwarzania przypada na akcelerację warstw PW ($\alpha(1) = 0.04$, $\alpha(8) = 0.26$, $\alpha(16) = 0.41$). Tym samym zakładając (6.5), gdzie f_0 oraz fps_0 to stałe.

$$fps(p, f) = p * \frac{f}{f_0} * fps_0 \quad (6.5)$$

Równanie zużycia energii przybiera postać (6.6).

$$e(p, f) = \frac{N}{p * \frac{f}{f_0} * fps_0} (P_{PS} + \beta(f)P_{DW} + p\beta(f)P_{PW}) \quad (6.6)$$

Przyjmując f jako stałą oraz zrównoleglenia p_1 oraz p_2 takie, że $p_1 < p_2$ to, aby dla p_2 osiągnąć mniejsze zużycie energii, niż dla p_1 wymagane jest spełnienie zależności (6.7).

$$e(p_2, f) < e(p_1, f) \quad (6.7)$$

$$\frac{1}{p_2} (P_{PS} + \beta(f)P_{DW} + p_2\beta(f)P_{PW}) < \frac{1}{p_1} (P_{PS} + \beta(f)P_{DW} + p_1\beta(f)P_{PW}) \quad (6.8)$$

$$\frac{P_{PS} + \beta(f)P_{DW}}{p_2} + \beta(f)P_{PW} < \frac{P_{PS} + \beta(f)P_{DW}}{p_1} + \beta(f)P_{PW} \quad (6.9)$$

$$\frac{1}{p_2} < \frac{1}{p_1} \quad (6.10)$$

$$p_1 < p_2 \quad (6.11)$$

co jest zawsze prawdziwe jako założenie. Wnioskując stwierdza się, iż im wyższy stopień zrównoleglenia tym osiągane jest mniejsze zużycie energii przetwarzania obrazu. W tym celu ustalono $p = 16$ (jako maksymalne zrównoleglenie pierwszej warstwy PW) oraz przeprowadzono ewaluację rozwiązań. W rezultacie uzyskano $fps = 72.7$ oraz zużycie energii $e = 2739J$. Co jest zgodne z wyznaczonymi nierównościami, pomimo zastosowania znaczących przybliżeń.

Ponadto dla (6.6) przyjmując p jako stałą oraz P_{PL} jako moc dynamiczną bez straty ogólności (nie znana jest moc P_{PS}). Zależność mocy P_{PL} od częstotliwości f może zostać wyrażona poprzez (6.12) [49] oraz zużycie energii poprzez (6.13).

$$\beta(f) = \frac{f}{f_0} \quad (6.12)$$

$$e(p, f) = \frac{N}{p * \frac{f}{f_0} * fps_0} (P_{PS} + \frac{f}{f_0} P_{PL}) \quad (6.13)$$

Przyjmując częstotliwości f_1 oraz f_2 takie, że $f_1 < f_2$, możliwe jest zmniejszenie zużycia energii, gdy spełniona jest zależność (6.14).

$$e(p, f_2) < e(p, f_1) \quad (6.14)$$

Resource	Utilization	Available	Utilization %
LUT	21141	70560	29.96
LUTRAM	4159	28800	14.44
FF	35593	141120	25.22
BRAM	212.50	216	98.38
DSP	269	360	74.72
IO	14	82	17.07
BUFG	4	196	2.04

Rys. 6.2. Zużycie zasobów – zrzut ekranu z programu Vivado 2019.1.

$$\frac{1}{f_2}(P_{PS} + \frac{f_2}{f_0}P_{PL}) < \frac{1}{f_1}(P_{PS} + \frac{f_1}{f_0}P_{PL}) \quad (6.15)$$

$$\frac{P_{PS}}{f_2} < \frac{P_{PS}}{f_1} \quad (6.16)$$

$$f_1 < f_2 \quad (6.17)$$

co jest zawsze prawdziwe jako stawiane założenie.

Wnioskując można stwierdzić, iż zastosowanie większej częstotliwości pracy akceleratora, tym uzyskiwane jest mniejsze zużycie energii całego systemu przetwarzania. Wartość częstotliwości musi jednak zapewniać prawidłowe wykonanie obliczeń. Ponadto niejawnie zakładano, iż część przetwarzania realizowana przez system procesorowy będzie wykonana w czasie krótszym, niż obliczenia logiki programowej. Eksperymentalnie sprawdzono, iż maksymalna częstotliwość pracy akceleratora nie może być większa niż 215 MHz. Obecna implementacja programowa nie pozwoliła jednak na uzyskanie wyższej przepustowości niż dotychczas. Dla wspomnianej maksymalnej częstotliwości uzyskano $fps = 71.0$ oraz zużycie energii $e = 2798J$.

Aby sprawdzić zużycie energii wynikające z pracy akceleratora, postanowiono pominąć etap odczytu danych oraz późniejszego przetwarzania z użyciem systemu procesorowego. Uzyskano wówczas $fps = 183$ oraz $e = 1070J$, co dowodzi słuszności wcześniejszych rozważań.

Ostateczne rozwiązanie zostało zaimplementowane ze zrównolegleniem $p = 16$. Na rysunku 6.2 przedstawiono zużycie zasobów logiki programowej. Implementacja wykorzystuje niemal wszystkie bloki pamięci *BRAM* oraz znaczączę część dostępnych *DSP*. Uzyskanie ostatecznie lepszych rezultatów dla tej architektury jest możliwe, jedynie poprzez bardziej wydajną implementację części programowej.

7. Podsumowanie

W ramach niniejszej pracy przedstawiono proces projektowania architektury sieci neuronowej do detekcji obiektów, która została następnie zaimplementowana na platformie sprzętowo-programowej Zynq UltraScale+ MPSoC. System ten był opracowywany na potrzeby konkursu *2021 DAC SDC*. Analizując stawiane wymagania, parametry docelowej platformy sprzętowej, a także rozwiązania z poprzednich edycji zaproponowano architekturę sieci *LittleNet*. Zastosowano konwolucje *depthwise* wykorzystującą wielu filtrów dla każdego kanału oraz konwolucje *pointwise*. Rozwiązanie to osiągało dokładność $IoU = 0.78$ dla modelu zmienoprzecinkowego. Przejście przez etap kwantyzacji do zapisu stałoprzecinkowego pozwoliło uzyskać już wartość $IoU = 0.7015$ (przy skalowaniu obrazu z wykorzystaniem liczb całkowitych). Wartość ta niestety tylko w niewielkim stopniu przekracza próg pozwalający na osiągnięcie maksymalnej wartości funkcji oceny dokładności detekcji. Zdecydowano się na zaprojektowanie własnego akceleratora sprzętowego z wykorzystaniem języka *System Verilog*. Moduł, po przeprowadzonej optymalizacji, osiągał przepustowość rzędu 72.7fps zużywając 2739J energii. Możliwa jest praca przy częstotliwości nawet 215 MHz osiągając przepustowość 183fps oraz zużycie energii wynoszące 1070J . Jednakże wynik ten uzyskiwany był z pominięciem odczytu obrazów oraz wszelkiego przetwarzania z użyciem systemu procesorowego. Zatem na obecnym etapie "wąskim gardłem" jest niewydajna implementacją programowa i opisany powyżej zabieg nie ma uzasadnienia.

Zaimplementowane rozwiązanie pozwala na osiągnięcie dobrych rezultatów. Jednakże, aby możliwe było poprawienie wydajności niezbędne jest zaimplementowanie części programowej w sposób wydajny np. wykorzystując język *C* wraz z przetwarzaniem wielowątkowym. Na etapie uczenia zastosowano funkcję *GCIOU* (4.11), której rezultaty należy jeszcze porównać z innymi funkcjami bazującymi na metryce *IoU*. Możliwe jest również zwiększenie dokładności obliczeń poprzez zmniejszenie stopnia kwantyzacji (zapis wykorzystujący więcej bitów) czy też uzależnienie go od konkretnych warstw (różna kwantyzacja dla warstw). Ponadto możliwe jest zredukowanie rozmiaru sieci, poprzez usunięcie wybranych filtrów tzw. *pruning*. Dobór liczby bitów części całkowitej można spróbować zrealizować w sposób automatyczny, poprzez ustanowienie jako parametr podlegający uczeniu. Zaproponowane typy akceleratorów można również poszerzyć m.in. o operację pełnej konwolucji, czy warstwy *FC*. Ponadto proces uczenia

sieci można zrealizować np. z użyciem klastra obliczeniowego *Prometeusz* z Akademickiego Centrum Komputerowego CYFRONET.

Bibliografia

- [1] „*DAC SDC 2021*”. [Dostęp online - 12.08.2021]. <https://dac-sdc-2021.groups.et.byu.net/doku.php>.
- [2] „*Ultra96-V2*”. [Dostęp online - 12.08.2021]. <https://www.avnet.com/wps/portal/us/products/new-product-introductions/npi/aes-ultra96-v2/>.
- [3] „*Pynq*”. [Dostęp online - 12.08.2021]. <http://www.pynq.io/>.
- [4] Xilinx. „*Zynq UltraScale+ MPSoC Product tables and product selection guide*”. Spraw. tech. TR-576. <https://www.xilinx.com/support/documentation/selection-guides/zynq-ultrascale-plus-product-selection-guide.pdf>. Xilinx, [Dostęp online - 12.08.2021].
- [5] Xilinx. „*UltraScale Architecture Configurable Logic Block*”. Spraw. tech. UG574 (v1.5). [https://www.xilinx.com/support/documentation/user_guides/ug574 -- ultrascale -- clb.pdf](https://www.xilinx.com/support/documentation/user_guides/ug574--ultrascale-clb.pdf). Xilinx, 2017.
- [6] Xilinx. „*UltraScale Architecture DSP Slice*”. Spraw. tech. UG579 (v1.10). [https://www.xilinx.com/support/documentation/user_guides/ug579 -- ultrascale -- dsp.pdf](https://www.xilinx.com/support/documentation/user_guides/ug579--ultrascale--dsp.pdf). Xilinx, 2020.
- [7] „*Brevitas*”. [Dostęp online - 13.08.2021]. <https://github.com/Xilinx/brevitas>.
- [8] „*PyTorch*”. [Dostęp online - 13.08.2021]. <https://pytorch.org/>.
- [9] „*FINN*”. [Dostęp online - 13.08.2021]. <https://xilinx.github.io/finn/>.
- [10] „*QKeras*”. [Dostęp online - 13.08.2021]. <https://github.com/google/qkeras>.
- [11] „*TensorFlow*”. [Dostęp online - 13.08.2021]. <https://www.tensorflow.org/>.
- [12] „*hls4ml*”. [Dostęp online - 13.08.2021]. <https://fastmachinelearning.org/hls4ml/>.
- [13] „*Vitis AI*”. [Dostęp online - 13.08.2021]. <https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html>.
- [14] P. Viola i M. Jones. „*Rapid Object Detection using a Boosted Cascade of Simple Features*”. W: *IEEE Conf Comput Vis Pattern Recognit* (2001).

- [15] N. Dalal i B. Triggs. „*Histograms of oriented gradients for human detection*”. T. 1. 2005, 886–893 vol. 1.
- [16] P. Felzenszwalb i in. „*Object Detection with Discriminatively Trained Part-Based Models*”. W: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 32.9 (2010), s. 1627–1645. DOI: 10.1109/TPAMI.2009.167.
- [17] R. Tadeusiewicz. „*Sieci neuronowe*”. Wydanie II. Warszawa: Akademicka Oficyna Wydawnicza, 1993.
- [18] „*A Beginner’s Guide to Neural Networks and Deep Learning*”. [Dostęp online - 30 .08.2021]. <https://wiki.pathmind.com/neural-network>.
- [19] „*What is a Neural Network?*” [Dostęp online - 30 .08.2021]. <https://medium.datadriveninvestor.com/what-is-a-neural-network-9ca88b29f7cb>.
- [20] S. Pokutta, C. Spiegel i Z. Zimmer. „*Deep Neural Network Training with Frank-Wolfe*”. W: *CoRR* abs/2010.07243 (2020). arXiv: 2010.07243.
- [21] „*Text Information Extraction*”. [Dostęp online - 29.08.2021]. <https://mageswaran1989.medium.com/text-information-extraction-2b4a976409ed>.
- [22] R. Girshick i in. „*Rich feature hierarchies for accurate object detection and semantic segmentation*”. W: *CoRR* abs/1311.2524 (2013). arXiv: 1311.2524.
- [23] J. Uijlings i in. „*Selective Search for Object Recognition*”. W: *International Journal of Computer Vision* 104 (2013), s. 154–171.
- [24] „*ImageNet*”. [Dostęp online - 17.08.2021]. <https://www.image-net.org/>.
- [25] A. Krizhevsky, I. Sutskever i G. Hinton. „*ImageNet Classification with Deep Convolutional Neural Networks*”. W: *Neural Information Processing Systems* 25 (sty. 2012). DOI: 10.1145/3065386.
- [26] K. Simonyan i A. Zisserman. „*Very Deep Convolutional Networks for Large-Scale Image Recognition*”. W: *arXiv* 1409.1556 (wrz. 2014).
- [27] R. Girshick. „*Fast R-CNN*”. W: *CoRR* abs/1504.08083 (2015). arXiv: 1504.08083.
- [28] S. Ren i in. „*Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*”. W: *CoRR* abs/1506.01497 (2015). arXiv: 1506.01497.
- [29] „*R-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms*”. [Dostęp online - 29.08.2021]. <https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e>.
- [30] W. Liu i in. „*SSD: Single Shot MultiBox Detector*”. W: *CoRR* abs/1512.02325 (2015). arXiv: 1512.02325.

- [31] J. Redmon i in. „*You Only Look Once: Unified, Real-Time Object Detection*”. W: *CoRR* abs/1506.02640 (2015). arXiv: 1506.02640.
- [32] J. Redmon i A. Farhadi. „*YOLO9000: Better, Faster, Stronger*”. W: *CoRR* abs/1612.08242 (2016). arXiv: 1612.08242.
- [33] J. Redmon i A. Farhadi. „*YOLOv3: An Incremental Improvement*”. W: *CoRR* abs/1804.02767 (2018). arXiv: 1804.02767.
- [34] A. Bochkovskiy, C. Wang i H. Liao. „*YOLOv4: Optimal Speed and Accuracy of Object Detection*”. W: *CoRR* abs/2004.10934 (2020). arXiv: 2004.10934.
- [35] Z. Zhaojun i in. „*Distance-IoU Loss: Faster and Better Learning for Bounding Box Regression*”. W: *CoRR* abs/1911.08287 (2019). arXiv: 1911.08287.
- [36] H. Rezatofighi i in. „*Generalized Intersection over Union*”. W: (czer. 2019).
- [37] C. Szegedy i in. „*Going Deeper with Convolutions*”. W: *CoRR* abs/1409.4842 (2014). arXiv: 1409.4842.
- [38] A. Howard i in. „*MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*”. W: *CoRR* abs/1704.04861 (2017). arXiv: 1704.04861.
- [39] I. Hubara i in. „*Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations*”. W: *CoRR* abs/1609.07061 (2016). arXiv: 1609.07061.
- [40] M. Rastegari i in. „*XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks*”. W: t. abs/1603.05279. 2016. arXiv: 1603.05279.
- [41] A. Bulat i G. Tzimiropoulos. „*XNOR-Net++: Improved Binary Neural Networks*”. W: *CoRR* abs/1909.13863 (2019). arXiv: 1909.13863.
- [42] „*Ultra_net*”. [Dostęp online - 12.08.2021]. https://github.com/heheda365/ultra_net.
- [43] X. Zhang i in. „*SkyNet: a Hardware-Efficient Method for Object Detection and Tracking on Embedded Systems*”. W: *CoRR* abs/1909.09709 (2019). arXiv: 1909.09709.
- [44] H. Kaiming i in. „*Deep Residual Learning for Image Recognition*”. W: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385.
- [45] „*OpenCV*”. [Dostęp online - 29.08.2021]. <https://opencv.org/>.
- [46] „*GoogleColab*”. [Dostęp online - 29.08.2021]. <https://colab.research.google.com>.
- [47] Xilinx. „*AXI DMA v7.1*”. Spraw. tech. PG021. https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf. Xilinx, 2019.

- [48] „AMBA 4 AXI4-Stream Protocol Specification”. [Dostęp online - 30.08.2021]. <https://developer.arm.com/documentation/ihi0051/a/Introduction/About-the-AXI4-Stream-protocol>.
- [49] „Accurate Dynamic Voltage and Frequency Scaling Measurement for Low-Power Microcontrollers in Wireless Sensor Networks”. W: *Microelectronics Journal* 105 (2020), s. 104874. ISSN: 0026-2692. DOI: <https://doi.org/10.1016/j.mejo.2020.104874>.

A. Dodatek A

W ramach realizacji pracy dyplomowej, poza powyższym dokumentem przygotowane zostały:

- Aplikacja do przeprowadzania procesu uczenia wybranej architektury (wliczając m.in. augmentację danych, uczenie modelu zmiennoprzecinkowego i kwantyzowanego, podział zbioru treningowego).
- Sprzętowy opis akceleratora sieci *LittleNet* z wykorzystaniem języka *System Verilog*.
- Aplikacja do konwersji wag modelu kwantyzowanego w zapisie zapisie zmiennoprzecinkowym do zapisu wymaganego przez akcelerator.
- Aplikacja testująca model sprzętowy akceleratora.
- Aplikacja sterująca procesem akceleracji.