Politechnika Warszawska

WYDZIAŁ ELEKTRYCZNY

Instytut Sterowania i Elektroniki Przemysłowej

Praca dyplomowa inżynierska

na kierunku Informatyka Stosowana

w specjalności Informatyka Stosowana

Porównanie wydajności wybranych języków programowania w realizacji sieci neuronowych do przetwarzania obrazów

Piotr Heinzelman
numer albumu 146703

promotor
dr inż. Witold Czajewski

WARSZAWA 2025

Porównanie wydajności wybranych języków programowania w realizacji sieci neuronowych do przetwarzania obrazów

Streszczenie

W niniejszej pracy poruszono zagadnienia tworzenia wysokowydajnych systemów obliczeniowych w celu realizacji modeli głębokich sieci neuronowych. Duże modele wymagają efektywnych sposobów obliczania odpowiedzi sieci i prowadzenia procesu uczenia, ponieważ wraz ze wzrostem wielkości modelu wyraźnie wzrasta czas wykonywania obliczeń.

Przypomniano sposoby zwiększania wydajności sprzętu, zwłaszcza te, które mogą być zastosowane w modelach sieci neuronowych.

We wprowadzeniu przybliżono matematyczne modele sieci CNN i MLP, a następnie rozpisano realizację tych modeli jako sekwencję działań arytmetycznych. Zwrócono uwagę na możliwość zwiększenia wydajności poprzez realizację przetwarzania w sposób równoległy.

Zaprezentowano pokrótce zadania postawione do wykonania porównywanym modelom.

W pracy dokonano porównania takich własności języków jak: popularność, dostępność bibliotek, wygoda instalacji, wsparcie obliczeń na kartach graficznych, stopień trudności języka, koszt licencji. Zaprezentowano również fragmenty kodu, a także opisano doświadczenia w trakcie pisania kodu.

Omówiono wybrane języki: Python, Maltab, Java, C++. Do budowania sieci Yolo8 wykorzystano biblioteki: Torch, Ultralytics. W projektach sieci CNN użyto: TensorFlow, Scikit-Learn, PyTorch. W projektach Matlab użyto pakietu Deep Learning Toolbox. W języju Java wykorzystano własne implementacje.

W podsumowaniu zebrano wyniki, oraz zaprezentowano przesłanki, które mogą być pomocne przy doborze języka w celu realizacji głębokich sieci neuronowych.

Słowa kluczowe: uczenie głębokich sieci neuronowych, sieci splotowe CNN, YOLO, wydajność, klasyfikacja obrazów, obliczenia równoległe, dodawanie sekwencyjne

Comparison of the performance of selected programming languages in the implementation of neural networks for image processing

Abstract

This paper addresses the issues of creating high-performance computing systems to implement deep neural network models. Large models require effective methods of calculating network responses and conducting the learning process, because the larger the model size, the more computational time is needed.

Methods of increasing hardware performance are recalled, especially those that can be used in neural network models.

The introduction presents mathematical models of CNN and MLP networks, and then describes the implementation of these models as a sequence of arithmetic operations. Attention is drawn to the possibility of increasing performance by implementing parallel processing.

The tasks set for the compared models are briefly presented.

The paper compares such language properties as: popularity, availability of libraries, ease of installation, support for computing on graphics cards, language difficulty level, and license cost. Code fragments are also presented, and experiences during writing the code are described.

Selected languages are discussed: Python, Maltab, Java, C++. The following libraries were used to build the Yolo8 network: Torch, Ultralytics. In the CNN projects, the following were used: TensorFlow, Scikit-Learn, PyTorch. In the Matlab projects, the Deep Learning Toolbox was used. In Java, own implementations were used.

The summary summarizes the results and presents premises that may be helpful in selecting a language for implementing deep neural networks.

Keywords: deep learning, convolution network, image clasification, efficiency, parallel computing

Spis treści

Rozdział 1

Wstęp

Inspiracją do podjęcia tematu pracy związanego z sieciami neuronowymi był kontakt autora z projektem budowy ”Sieci do rozpoznawania zanieczyszczeń w powietrzu” realizowanej na wydziale chemii PW około roku 1995 (był to model sieci MLP). A w konsekwencji lektura [1]. Na wybór konkretnego tematu i dobór języków miała wpływ także lektura [2] oraz [3], [4], [5], [6].

Dzisiejsze prawie powszechne zainteresowanie sztucznymi sieciami neuronowymi w środowiskach zarówno neurobiologów, fizyków, matematyków jak i inżynierów wynika z potrzeby budowania bardziej efektywnych i niezawodnych systemów przetwarzania informacji, w oparciu o metody jej przetwarzania w komórkach nerwowych. Fascynacje mózgiem człowieka i jego właściwościami (np. odpornością na uszkodzenia, przetwarzaniem równoległym informacji rozmytej i zaszumionej i innymi) w latach 40-stych dały początek pracom w zakresie syntezy matematycznej modelu pojedynczych komórek nerwowych, a później na tej podstawie struktur bardziej złożonych w formie regularnych sieci. Należy pamiętać, że z obliczeniowego punktu widzenia, rozwijane i budowane sztuczne sieci neuronowe są oparte na nowych regułach wynikających z zasad neurofizjologii[1]. Sztuczne sieci neuronowe zdobyły szerokie uznanie w świecie nauki poprzez swoją zdolność łatwego zaadaptowania do rozwiązywania różnorodnych problemów obliczeniowych w nauce i technice. Mają właściwości pożądane w wielu zastosowaniach praktycznych: stanowią uniwersalny układ aproksymacyjny odwzorowujący wielowymiarowe zbiory danych, mają zdolność uczenia się i adaptacji do zmieniających się warunków środowiskowych, zdolność generalizacji nabytej wiedzy, stanowiąc i pod tym względem szczytowe osiągnięcie sztucznej inteligencji[3]. Za protoplastę tych (głębokich) sieci można uznać zdefiniowany na początku lat ‘90 wielowarstwowy neocognitron prof. Kunihiko Fukushimy. Prawdziwy rozwój tych sieci zawdzięczamy jednak profesorowi Yann A. LeCun, który zdefiniował podstawową strukturę i algorytm uczący specjalizowanej sieci konwolucyjnej CNN. Aktualnie sieci CNN stanowią podstawową strukturę stosowaną na szeroką skalę w przetwarzaniu sygnałów i obrazów. W międzyczasie powstało wiele odmian sieci będącej modyfikacją struktury podstawowej (R-CNN, AlexNET, GoogLeNet, ResNet, U-Net, YOLO)[2]. Ważnym rozwiązaniem jest sieć YOLO (ang. You Only Look Once), wykonująca jednocześnie funkcje klasyfikatora i systemu regresyjnego, który służy

do wykrywania określonych obiektów w obrazie i określaniu ich współrzędnych. Obecnie dostępna jest już 12 wersja[2].

Biblioteki dostarczające implementacje modeli sieci neuronowych powstały dla większości języków programowania ogólnego przeznaczenia. Niektóre z nich wykorzystują procesory graficzne do wysokowydajnych równoległych obliczeń, co powoduje gwałtowny wzrost wydajności i znaczące obniżenie czasu uczenia sieci. Budowanie własnych modeli sieci jest dziś w zasięgu osób prywatnych, hobbystów, studentów czy małych zespołów badawczych i nie wymaga ogromnych nakładów finansowych.

1.1 Cel pracy

Podstawowym celem pracy jest ułatwienie podjęcia decyzji o wyborze języka i środowiska w fazie projektowej dla realizacji aplikacji wykorzystujących głębokie sieci neuronowe CNN.

Celem dydaktycznym jest dogłębne zapoznanie się z tematyką sieci MLP [1] oraz CNN [7] [8] poprzez realizacje i testy własnego rozwiązania zwłaszcza z wykorzystaniem możliwości obliczeniowych karty graficznej.

1.2 Układ pracy

Warunkiem niezbędnym do prowadzenia efektywnych badań nad głębokimi sieciami i dużymi modelami jest zdolność efektywnego wykorzystania systemów o dużych mocach obliczeniowych. W pierwszej części opisano metody zwiększania wydajności systemów cyfrowych i zagadnienia przetwarzania równoległego.

W drugiej części przedstawiono stan wiedzy z zakresu działania głębokich sieci neuronowych tj. Perceptronu wielowarstwowego (ang. Multilayer Perceptron, MLP) oraz Konwolucyjnej sieci neuronowej (ang. Convolutional Neural Network, CNN).

Zaprezentowano propagację sygnałów przez sieć, propagację wsteczną i oparty na niej proces uczenia sieci.

W trzeciej części zaprezentowano zadania, które będą rozwiązywane przez badane modele sieci.

W czwartej części dokonano porównania języków, opisano wybrane cechy, informacje o wykorzystanych bibliotekach, pokazano fragmenty kodu. Zaprezentowano też wyniki pomiarów z podziałem na języki.

Ostatnia część zawiera wyniki oraz przesłanki, które mogą być pomocne przy doborze języka w celu realizacji głębokich sieci neuronowych w zależności od konkretnych wymagań i możliwości stawianych projektowanym rozwiązaniom.

1.3 Kod źródłowy i dane uczące

Przykłady rozwiązań w Python i Matlab zaczerpnięto

- z książek: [2] [3] [9] [10]
- instrukcji i przykładów załączonych do bibliotek
- otwartych repozytoriów git [11] [12] [13]

Obrazy treningowe

- pisma odręcznego pochodzą z bazy MNIST (yann.lecun.com)
- zdjęcia twarzy z serwisów: google.com oraz filmweb.pl

Pełen kod dostępny na github:

- github.com/piotrHeinzelman/inz/tree/main/MixedProj

W analizie nie brano pod uwagę czasów czytania plików oraz przygotowania danych.

Rozdział 2

Model perceprtonu wielowarstwowego MLP

Neuron biologiczny przetwarza impulsy wejściowe w trzech krokach:

- ważenie wejściowych sygnałów - jest to skomplikowany proces, zmienny w czasie i zależny od historii wcześniejszych impulsów; w modelu matematycznym odpowiada mu mnożenie sygnału wejściowego x_i przez wielkość wagi w_i przypisanej do tego sygnału. $x_i w = x_i * w_i$
- jednoczesne sumowanie potencjałów sygnałów wejściowych - w modelu matematycznym odpowiada mu algebraiczna suma ważonych sygnałów: $y = \sum x_i w$
- nieliniowa aktywacja - w modelu matematycznym obliczenie wartości funkcji aktywacji $z = F(y)$
- w modelu sygnałowym dla zwiększenia wydajności zapamiętano wartość pochodnej $\frac{\partial F}{\partial y}(z)$ - zostanie ona wykorzystana w późniejszych obliczeniach w procesie uczenia.

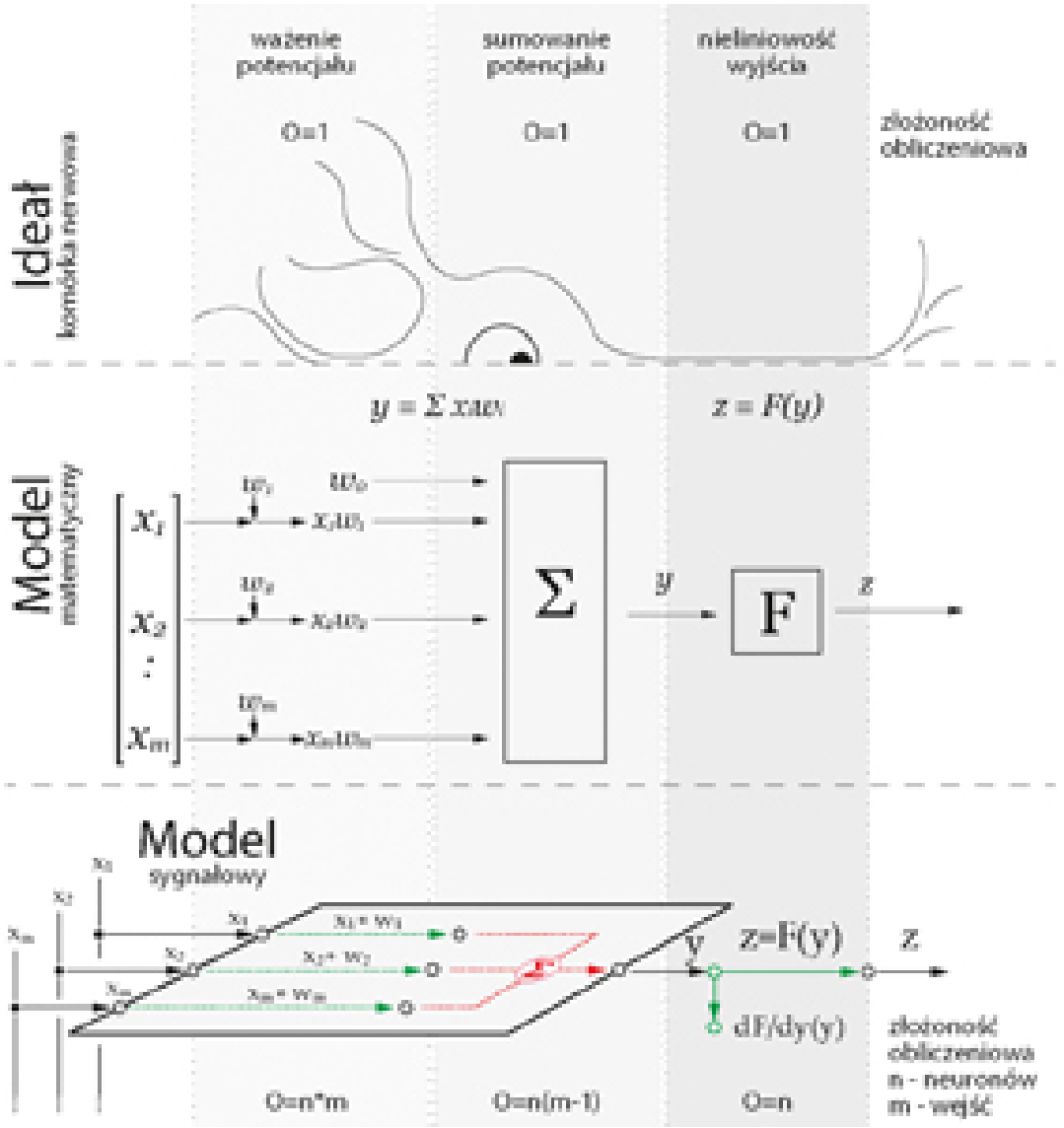
rodzaj warstwy	funkcja aktywacji $z = F(y)$	wartość pochodnej $\frac{\partial F}{\partial y}$
logistyczna (sigmoid)	$\frac{1}{1+e^{-}y}$	$(z)(1-z)$
ReLU	jeśli y<0 to z=y, jeśli y>=0 to z=0	jeśli y<0 to 1, jeśli y>=0 to 0
softmax	$z_i = \frac{e^{x_i}}{\sum e^{x_i}}$	dla prawidłowej klasy k: $y_k(1-y_k)$,
softmax		dla pozostałych klas: $-y_i * y_k$

2.1 Perceptron warstwowy

Wejście neuronu przyjmuje wartości $[x_1, x_2, ..., x_m]$ (wektor), natomiast na wyjściu jednego neuronu pojawia się tylko jeden sygnał wyjściowy z.

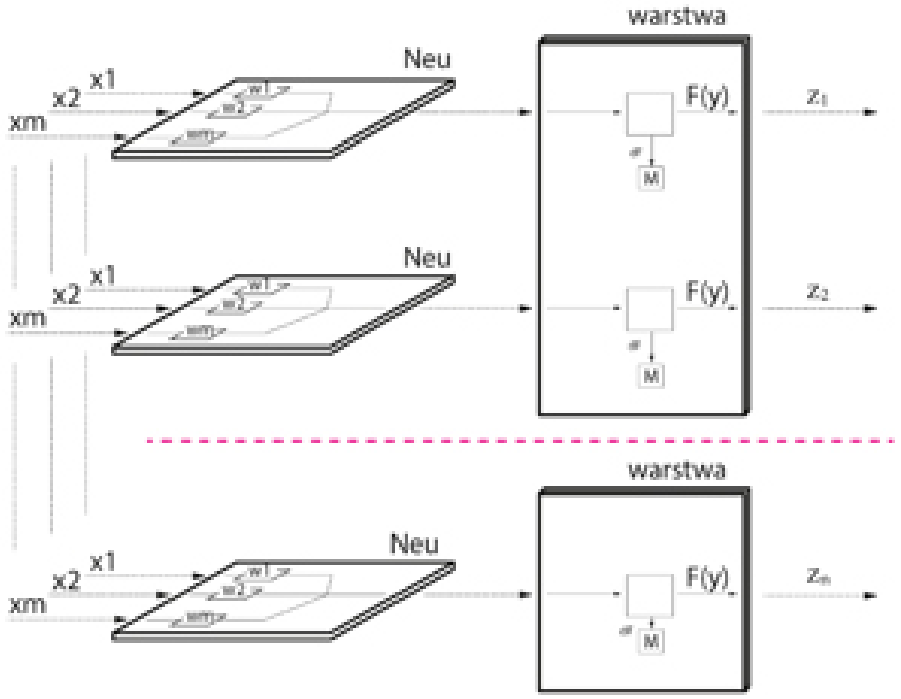
2.2 Proces uczenia MLP

Odpowiedzią sieci jest sygnał wyjściowy ostatniej warstwy. W procesie uczenia z nauczycielem podlega on ocenie, a wielkość błędu sieci jest przekazywana do warstwy wyjściowej. Stosowane są

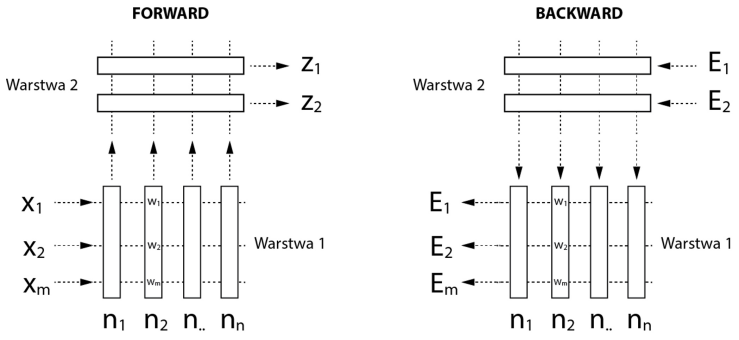


Rysunek 1. Neuron, model matematyczny i sygnałowy

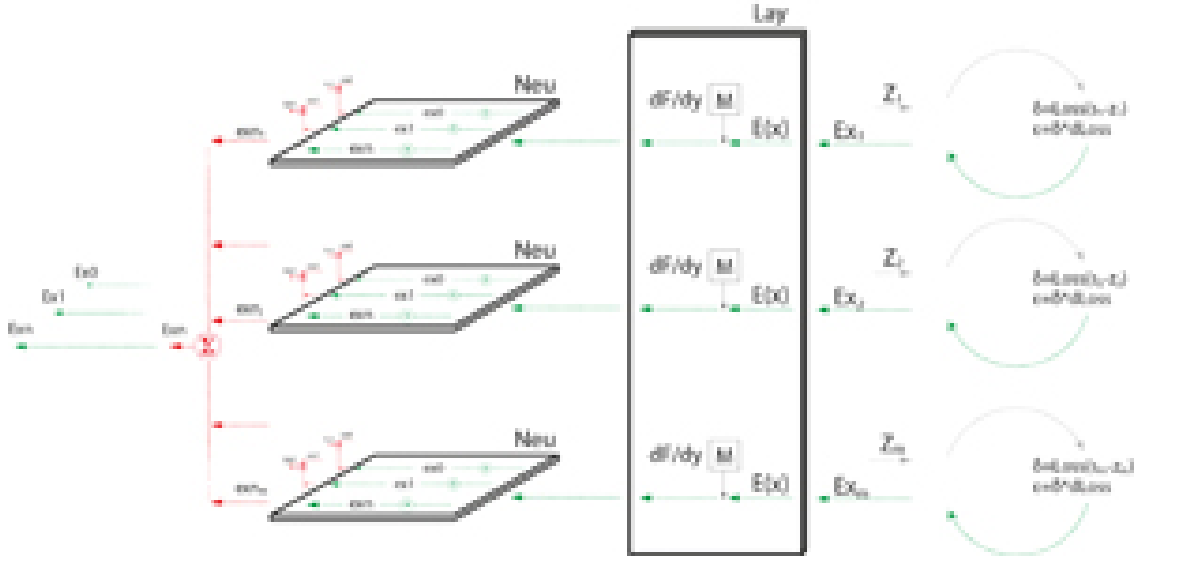
różne strategie wyznaczania wielkości tego wektora. Dla warstwy wyjściowej logistycznej może to być różnica między oczekiwaną odpowiedzią, a odpowiedzią uzyskaną: $\begin{bmatrix} e_1 \\ e_2 \\ e_m \end{bmatrix} = \begin{bmatrix} s_1 \\ s_2 \\ s_m \end{bmatrix} - \begin{bmatrix} z_1 \\ z_2 \\ z_m \end{bmatrix}$ gdzie S jest wektorem oczekiwanym. Dla każdej wartości wyjściowej z_i zostaje zwrócona wielkość błędu e_i, ponadto dla każdej wartości z_i zostaje zapamiętana wartość pochodnej $\frac{\partial F}{\partial y}$. W procesie uczenia do każdego z neurosumatorów zostaje przekazana odpowiadająca mu wielkość błędu: $dFe = e_i * \frac{\partial F}{\partial y}$.



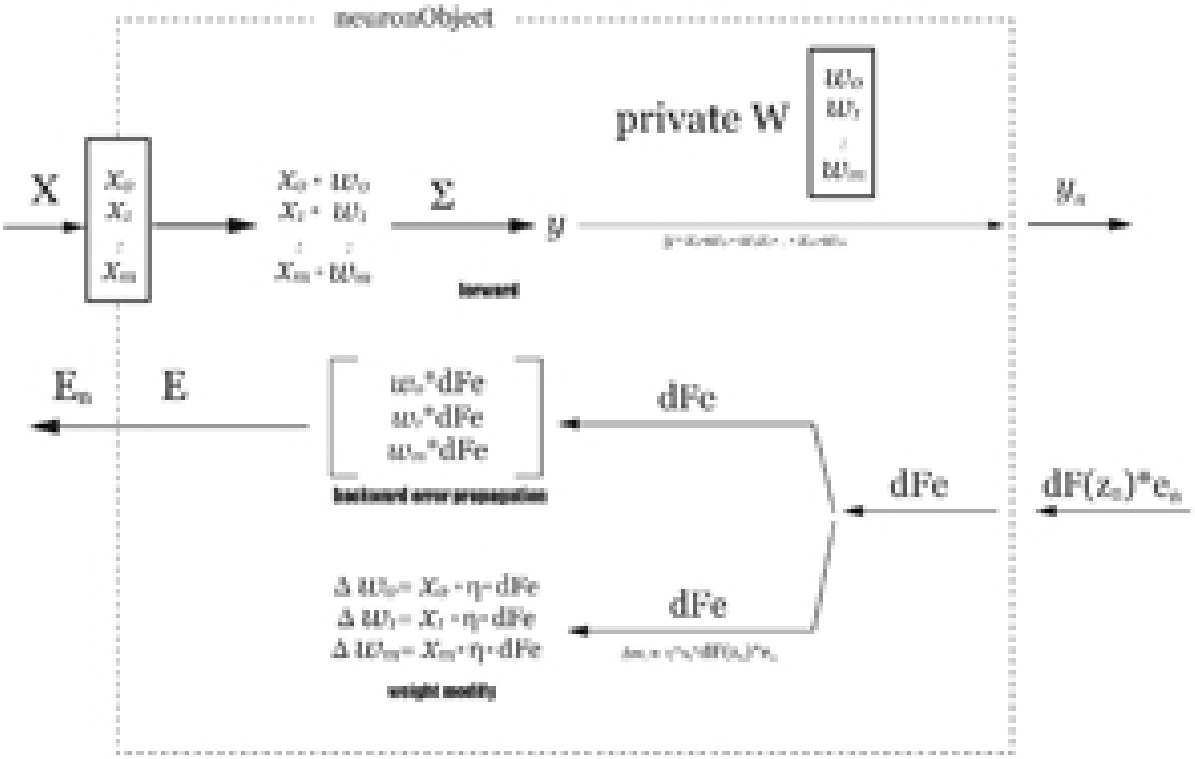
Rysunek 2. Przepływ sygnałów przez warstwę w czasie propagacji sygnału ”w przód”



Rysunek 3. Propagacja sygnału przez zespół warstw w przód i wstecz



Rysunek 4. Propagacja wstecz - uczenie sieci. (Operacje niezależne (mnożenie) oznaczone kolorem zielonym i operacje wymagające synchronizacji (dodawanie) oznaczone kolorem czerwonym)



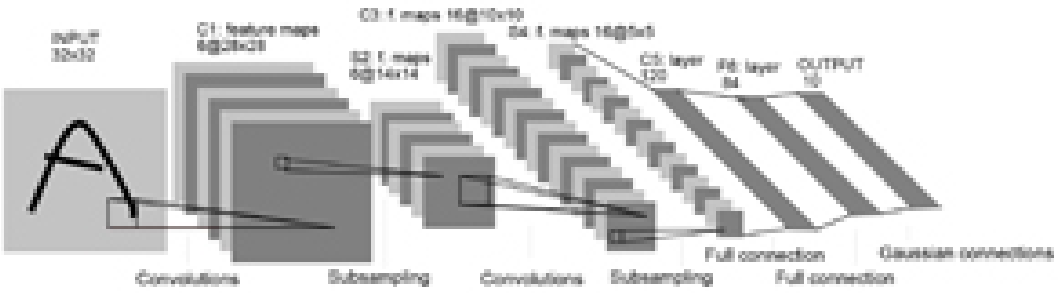
Rysunek 5. Przepływu sygnałów w obiekcie Neurosumator

Rozdział 3

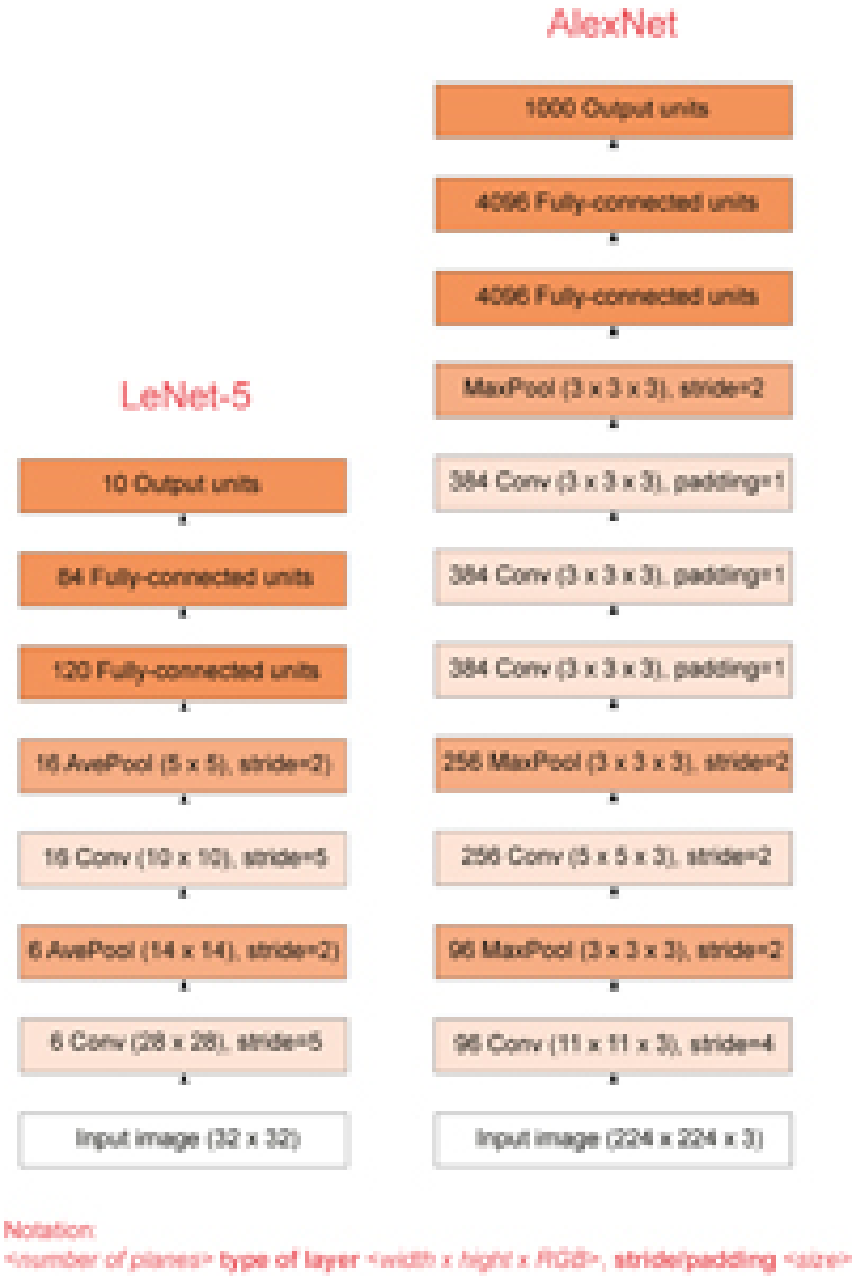
Model sieci głębokiej CNN

3.1 Wstęp

Sieci CNN powstały jako narzędzie analizy i rozpoznawania obrazów wzorowanym na sposobie działania naszych zmysłów. Wyeliminowały kłopotliwy proces manualnego opisu cech charakterystycznych obrazów. W tym rozwiązaniu sieć sama odpowiada za generację cech charakterystycznych dla klas [2]. Przetwarzania obrazu przez zestaw filtrów i warstw generuje obrazy o coraz mniejszych wymiarach, lecz o zwiększającej się liczebności kanałów reprezentujących cechy charakterystyczne dla przetwarzanego zbioru. Próbka danych w kolejnych etapach jest reprezentowana przez tensory (struktury trójwymiarowe, których szerokość i wysokość odpowiada wymiarom obrazu, a głębokość jest równa liczbie kanałów w obrazie). Wyjście z ostatniej warstwy sieci w procesie wypłaszczania jest przetwarzane na postać wektorową (jednowymiarowa tablica liczb), a następnie przekazywane na wejście układu klasyfikującego - sieci MLP z wyjściem Softmax [14] [15].



Rysunek 6. Architektura LaNet-5 [14]

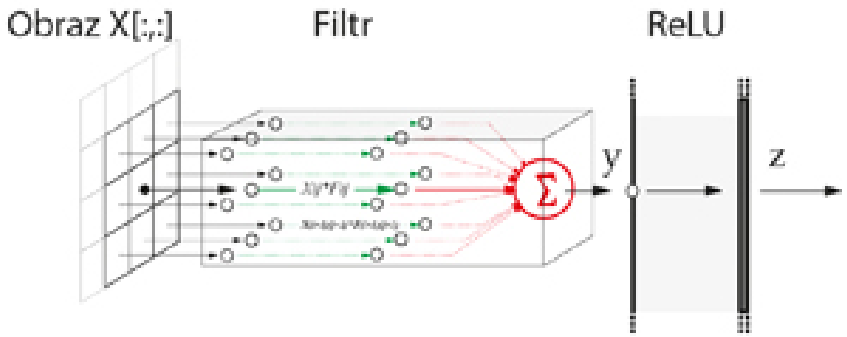


Rysunek 7. Porównanie struktury LaNet i AlexNet [16]

3.2 Operacja splotu

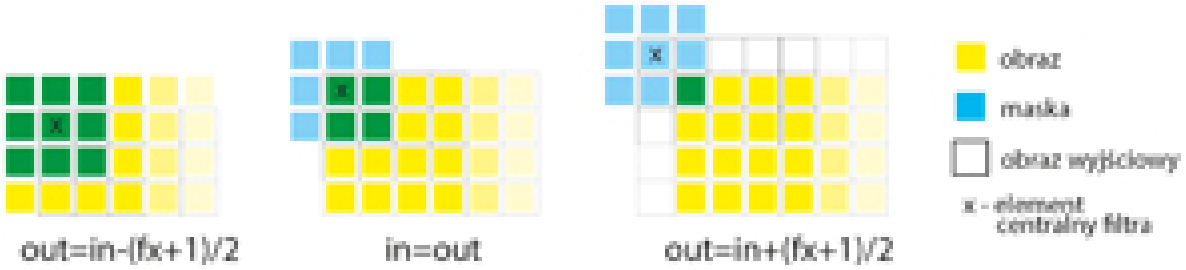
Jeśli wartości pikseli obrazu wejściowego X oznaczymy $X_{(i,j)}$, a wartości filtra F oznaczymy jako $F_{(m,n)}$ obraz wejściowy Y, $Y_{(o,p)}$, wówczas każdy piksel obrazu wyjściowego obliczymy:

$$Y_{(O,P)} = \sum_{(n=1)}^{(N)} \sum_{(m=1)}^{(M)} F_{(m,n)} * X_{(m+O,n+P)} \tag{1}$$



Rysunek 8. Obliczanie wartości pojedynczego piksela w procesie splotu analogicznie jak w MLP, sprowadza się do obliczenia sumy ważonej piksela centralnego filtra z pikselem obrazu oraz najbliższych sąsiadów centralnego piksela - tych które znajdują się naprzeciw pikseli filtra.

3.2.1 Padding



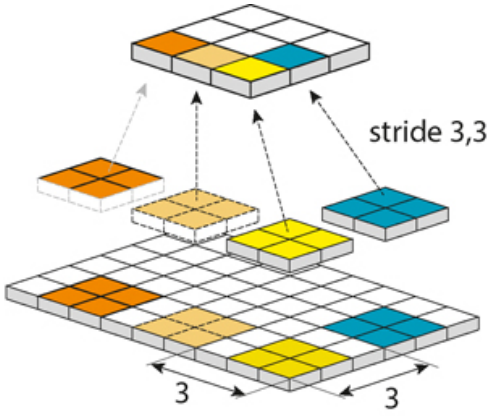
Rysunek 9. Wpływ punktu startowego maski na wymiar obrazu wynikowego operacji splotu

Padding oznacza wysunięcie maski poza obszar obrazu.
Jeśli w procesie konwolucji centralny piksel maski:

- znajduje się przed granicą obrazu, wówczas obraz wynikowy będzie mniejszy (p>0).
- znajduje się nad pierwszym pikselem obrazu, wówczas wielkość obrazu wyjściowego jest taka jak obrazu wejściowego (p=0).
- znajduje się poza pikselem obrazu, wówczas wielkość obrazu wyjściowego będzie większa (p<0).

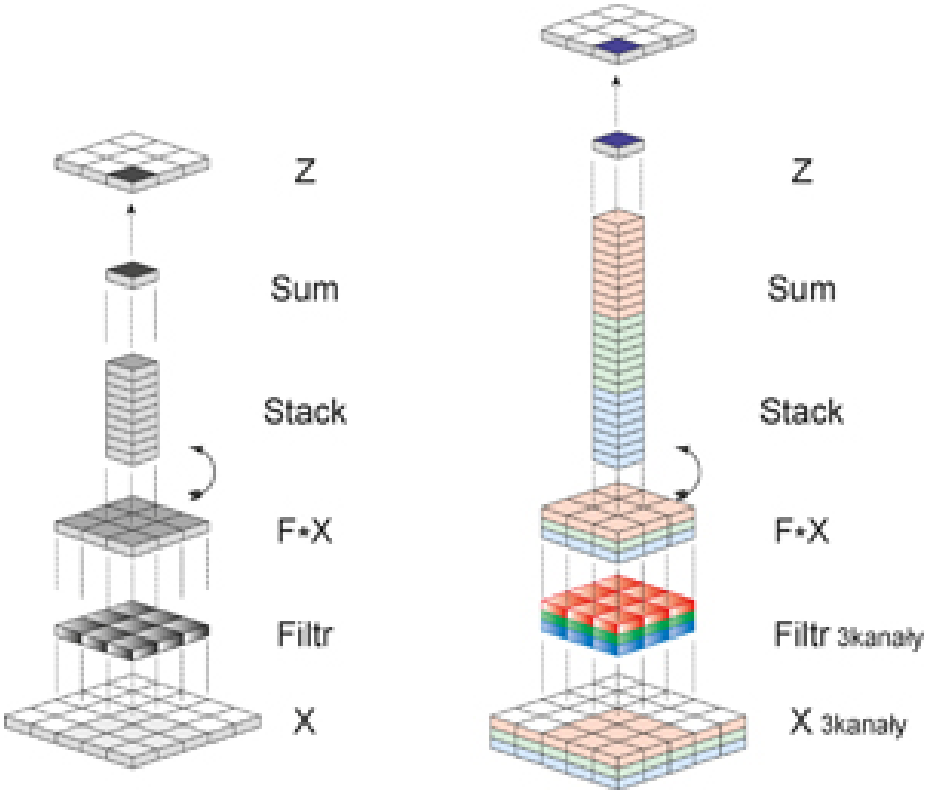
W założeniu procesu uczenia, każdy przesyłany sygnał wejściowy i wyjściowy jest sprzężony z powrotnym sygnałem określającym wielkość błędu tego sygnału. Zatem jeśli w operacji splotu padding jest dodatni, wówczas przy obliczeniu błędu wyjściowego zastosowany będzie padding ujemny.

3.2.2 Stride



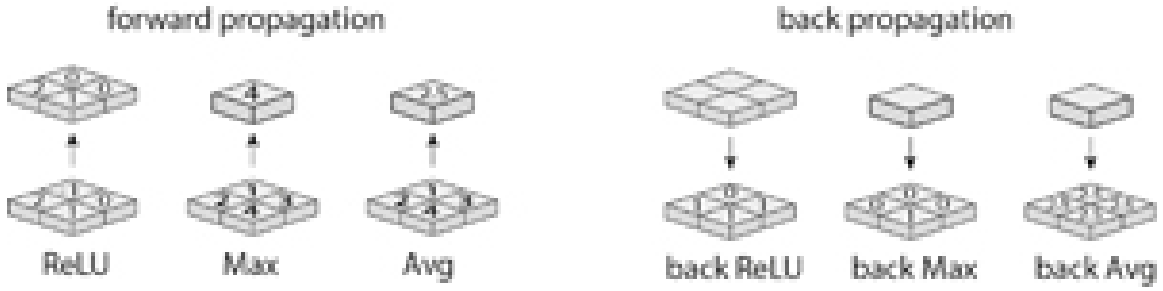
Rysunek 10. Stride, krok przesunięcia maski

Wymiar wyjściowy obrazu $dimOut = (dimIn + 1 - dimF + 2 * padding)/stride$



Rysunek 11. Konwolucja jedno i wielokanałowa

3.3 ReLU i warstwy redukujące wymiar



Rysunek 12. Wynik działania warstw ReLU, Avg, Max przy przepływie wprost i wstecz.

Po operacji splotu można zastosować warstwę ReLU. Funkcja aktywacji ReLU = max{x, 0}. Zerowane są wartości ujemne, funkcja nie ma ciągłej pochodnej, przy obliczeniach przyjmowano że pochodna wynosi 1 gdy wartość wyjścia z była większa od 0, lub 0 jeśli wartość z była mniejsza od zera.
Można także zastosować warstwy redukującą rozmiar. Warstwa AVG - na wyjściu zwraca średnią wartości sąsiednich pól; pochodna przy przepływie wstecz dla każdego pola wynosi $1/n^2$ gdzie n jest wielkością filtra. Warstwa MAX - na wyjściu zwraca wartość maksymalną z pól objętych zasięgiem filtra. Pochodna przy propagacji wstecz wynosi 1 dla wartości maksymalnej i 0 dla pozostałych.

3.4 Przetwarzanie w przód

Obraz wejściowy w skali szarości, lub obraz kolorowy rozbity na 3 kanały RGB zostaje przetworzony w operacji splotu przez grupy warstw filtrów i warstwy redukujące. Warstwa konwolucyjna może zmniejszyć, a warstwa redukująca zawsze zmniejsza rozmiar obrazu, w efekcie otrzymujemy coraz mniejsze obrazy wyjściowe.
Wyjściem każdego filtra jest jeden kanał obrazu. W miarę zmniejszania obrazów stosowana jest większa liczbę filtrów, co zwiększa liczbę kanałów. Przedostatnia warstwa spłaszcza wszystkie kanały obrazu do postaci pojedynczego jednowymiarowego wektora, warstwa ostatnia Full Connected z wyjściem Softmax dokonuje klasyfikacji obrazu.

3.5 Przetwarzanie wstecz - modyfikacja filtra w warstwie splotowej

Wektor wielkości błędu zostaje przekazany z warstw MLP przez warstwy redukujące do warstw splotowych, w których następuje modyfikacja Filtra zgodnie z formułą.

$$\frac{\partial L}{\partial F} = Conv(Input.X, Loss.Gradient\frac{\partial L}{\partial O}); F = F - \mu * \frac{\partial L}{\partial F} \tag{2}$$

$$\frac{\partial L}{\partial Bias} = Sum(Loss.Gradient\frac{\partial L}{\partial O}); Bias = Bias - \mu * \frac{\partial L}{\partial Bias} \tag{3}$$

3.6 Propagacja błędu przez warstwę splotową

Do warstwy poprzedniej zostanie przekazany wektor błędu o wartości wyliczonej wg. wzoru:

$$\frac{\partial L}{\partial X} = FullConv(180Rotated.filter.F, Loss.Gradient\frac{\partial L}{\partial O}) \tag{4}$$

Rozdział 4

Problemy i skuteczne sposoby ich rozwiązywania

4.1 Maszyna Turinga

Zasada działania maszyny Turinga sprowadza się do cyklicznego wykonywania sekwencji operacji: odczytu wartości z pamięci, wykonaniu działania, zapisaniu wyniku w pamięci. Modelowanie tworu przetwarzającego informacje całą objętością przy użyciu maszyny sekwencyjnej w miarę powiększania modelu spowoduje lawinowy wzrost czasu przetwarzania.

4.2 Szybsze obliczenia grafiki 3D

W zastosowaniach inżynierskich (oraz rozrywkowych) maszyn liczących bardzo szybko pojawiała się potrzeba realizacji szybkich obliczeń translacji punktów w przestrzeni 3D. Operacje te sprowadzają się do mnożenia wektora współrzędnych znormalizowanych punktu przez macierz translacji:

$$P = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}, T = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}, O_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, R = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

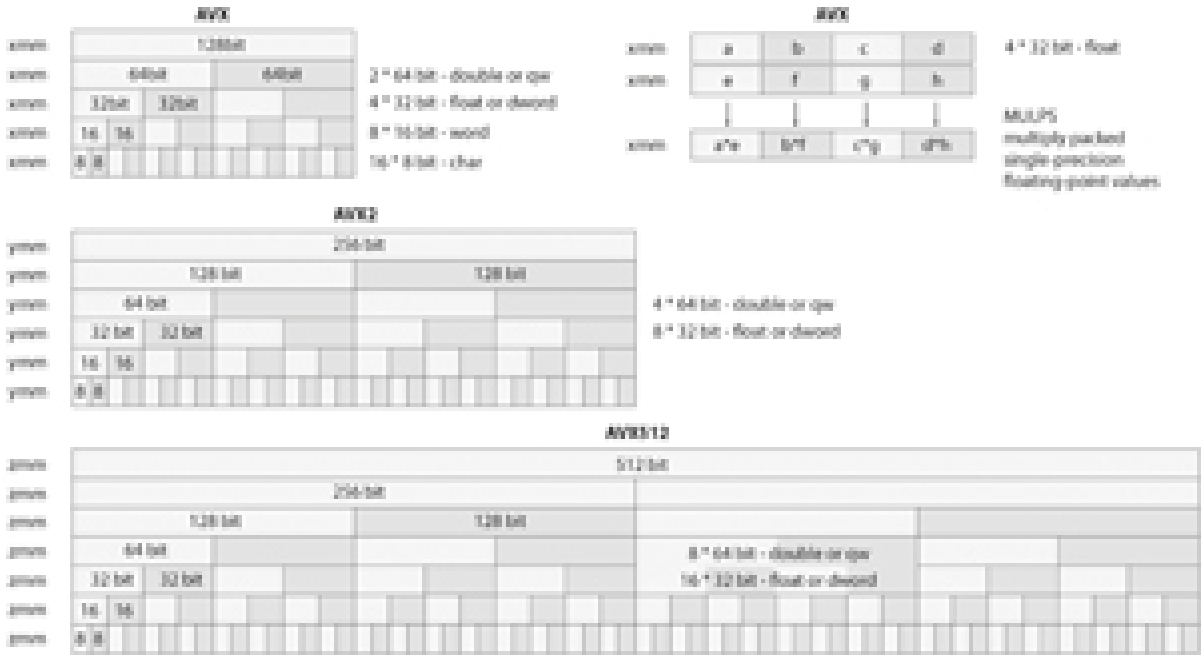
a po rozpisaniu mamy:

$$\begin{aligned} X &= x * m[0][0] + y * m[1][0] + z * m[2][0] + w * m[3][0] \\ Y &= x * m[0][1] + y * m[1][1] + z * m[2][1] + w * m[3][1] \\ Z &= x * m[0][2] + y * m[1][2] + z * m[2][2] + w * m[3][2] \\ W &= x * m[0][3] + y * m[1][3] + z * m[2][3] + w * m[3][3] \end{aligned}$$

Pierwszą odpowiedzią na to zapotrzebowanie był wprowadzony na rynek w 1980 roku koprocesor arytmetyczny Intel 8087, jego zadaniem było przyspieszenie operacji arytmetycznych zwłaszcza na liczbach zmiennoprzecinkowych w systemach opartych o procesor Intel 8088 i Intel 8086.

4.2.1 Funkcje procesora MMX, SIMD, SSE, AVX

SIMD (Single Instruction, Multiple Data) to funkcja procesora, która pozwala wykonywać jedną instrukcję na wielu 'strumieniach' danych. Może ona ewentualnie zwiększyć wydajność Twoich programów. SIMD to postać przetwarzania równoległego; jednak w niektórych przypadkach przetwarzanie różnych strumieni może odbywać się sekwencyjnie. Pierwszą implementacją był zbiór instrukcji MMX, zastąpiony przez strumieniowe rozszerzenia SIMD (ang. Streaming SIMD Extension, SSE). Później do SSE dodano zaawansowane rozszerzenia wektorowe (ang. Advanced Vector Extension, AVX).[17] Procesor obsługujący SSE ma 16 dodatkowych rejestrów 128-bitowych (xmm0-xmm15). Procesor obsługujący AVX2 ma 256-bitowe rejestry ymm, AVX-256 i AVX-10 512-bitowe rejestry zmm.[17]



Rysunek 13. Rejestry AVX, AVX2 i AVX512,operacja równoległego mnożenia AVX

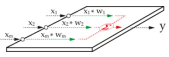
Rozkaz MULPS wykonuje jednocześnie jednocześnie 4 mnożenia zestawu 8 liczb 32 bitowych. Instrukcje w AVX512 VMPADDWD, VMPADDUBSW - wykonują mnożenie wektorowe, następnie dodawanie sąsiednich elementów. $(a_i \cdot b_i + a_{i+1} \cdot b_{i+1})$

Funkcje te przyspieszają operacje iloczynu Hadamarda (.*)[18] czyli wielokanałowego mnożenia. W zależności od wersji dla zmiennych typu float uzyskano od 4 do 16 kanałów, czyli 16 operacji mnożenia w czasie jednego rozkazu. [19]

4.3 Obliczenia na GPU

4.3.1 Równoległe mnożenie

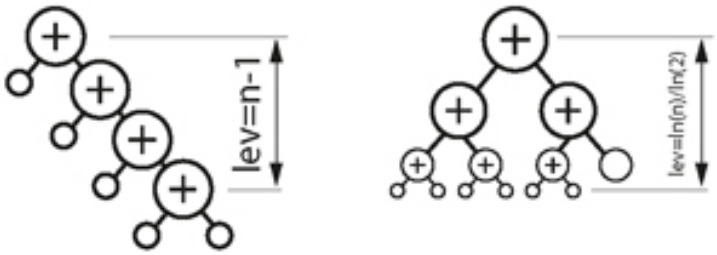
Wydajne obliczanie iloczynu Hadamarda (.*) wektorów o dużych rozmiarach mogą zapewnić procesory graficzne, w których znajdują się tysiące jednostek obliczeniowych. Czas wykonania kilku tysięcy mnożeń zajmuje tyle samo czasu co wykonanie jednego mnożenia. Użycie kart graficznych przy modelowaniu głębokich sieci jest koniecznością, ponieważ skraca czas trenowania sieci o rzędy wielkości w stosunku do obliczeń na CPU. W niniejszej pracy wykorzystano kartę NVIDIA GeForce RTX 4070 wyposażonej w 5888 rdzeni CUDA oraz 12282 MB pamięci na karcie graficznej. [20] [21]



Rysunek 14. Mnożenie równoległe i wielokanałowe dodawanie

4.3.2 Wielokanałowe dodawanie

Sumowanie a+b+c+d realizowane jest analogiczne do zdegenerowanego drzewa binarnego czyli (((a+b)+c)+d). Przy dużej ilości składników możemy spróbować zastosować zwykłe drzewo binarne niezdegenerowane, a operacja dodawania może stanie się częściowo równoległa ((a+b) + (c+d). Maszyny cyfrowe obecnie nie są wyposażone w sumatory wielokanałowe pozwalające dodawać więcej niż 2 liczby jednocześnie. Można zasymulować quasi-równoległe dodawanie stosując nawiasy.



Rysunek 15. drzewo zdegenerowane i niezdegenerowane

- $b = ((..((a1 + a2) + (a3 + a4)) + ((a5 + a6)...$ czas wykonania: **0.09568 [sek.]**
- $b = a1 + a2 + a3 + a4 + ... + a1024$ czas wykonania: **0.247875 [sek.]**

Uzyskamy dwukrotne zwiększenie szybkości przez dodanie nawiasów. Przy 32 składnikach mamy 16 + 8 + 4 + 2 + 1 dodawań, jednak pierwsze 16 może być wykonane równolegle, kolejna 8 także jest od siebie niezależna, podobnie 4 i 2. Czyli dla 32 elementów mamy 31 dodawań w 5 poziomach. Dla 1024 składników mamy 1023 dodawania, 512+256+128...+1 w 10 poziomach. [20]

4.4 Matlab

Dodatek Parallel computing w Matlab umożliwia wykonywanie obliczeń równolegle, w osobnych wątkach, procesach, a także z wykorzystaniem kart graficznych (obecnie tylko NVidia). Dodatki Optimalization Toolbox i Optimalization Computing Toolbox optymalizują tworzony kod, zwiększając jego wydajność. Dodatek Deep Learning Toolbox dostarcza gotowych metod do obliczeń sieci neuronowych.
w Matlab karty graficzne (NVidia) są widoczne nawet bez instalowania w systemie dedykowanych sterowników. Karty AMD nie są widoczne.

Rozdział 5

Zadania

5.1 Zadanie 1 - regresia liniowa

Zadanie zaczerpnięto z [2] i polegało na wyznaczeniu czasu obliczeniu funkcji polyfit:

$$p(x) = p_1 x^n + p_2 x^{n1} + ... + p_n x + p_{n+1}$$

Dane wejściowe wygenerowano losowo i zapisano w pliku. Użyto tych samych danych we wszystkich realizacjach. W językach C++ i Java napisano własny kod:

C++

```
for (int c = 0; c < CYCLES; c++) {
   xsr = 0.0; ysr = 0.0;
   w1  = 0.0; w0  = 0.0;
   for ( int i=0; i<len; i++ ){
      xsr +=  X[i];
      ysr +=  Y[i];
   }
   xsr=xsr / len;
   ysr=ysr / len;

   double sumTop=0.0;
   double sumBottom=0.0;
   for ( int i=0;i<len;i++ ){
      sumTop    += ((X[i]-xsr)*(Y[i]-ysr));
      sumBottom += ((X[i]-xsr)*(X[i]-xsr));
   }
   w1 = sumTop / sumBottom;
   w0 = ysr -(w1 * xsr) ;
}
```

Python

```
for ( int C=0; C<cycles; C++ ) {
   double xsr = 0.0;
   double ysr = 0.0;
   for (int i = 0; i < x.length; i++) {
      xsr += x[i];
      ysr += y[i];
   }
   xsr = xsr / x.length;
   ysr = ysr / y.length;

   w1 = 0.0;
   w0 = 0.0;
   double sumTop = 0.0;
   double sumBottom = 0.0;
   for (int i = 0; i < x.length; i++) {
      sumTop += ((x[i] - xsr) * (y[i] - ysr));
      sumBottom += ((x[i] - xsr) * (x[i] - xsr));
   }
   w1 = sumTop / sumBottom;
   w0 = ysr - w1 * xsr;
}
```

W językach Python i Matlab wykorzystano dostępne funkcje języka:

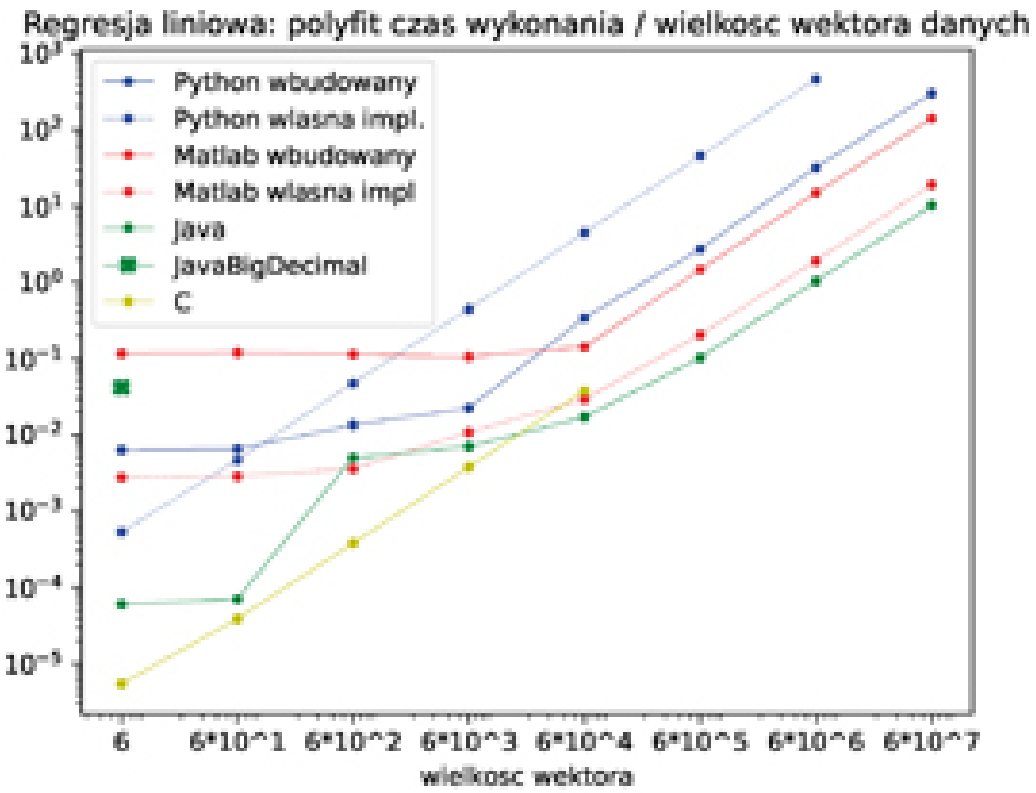
Matlab

```
for i = 1:cycles
    a = polyfit(x,y,1);
end
```

Python

```
for c in range( cycles ):
    a = np.polyfit(x,y,1)
```

Zadanie realizowano bez użycia GPU, czas wczytania danych jest pomijany.



Rysunek 16. Zadanie 1 - zestawienie czasu obliczeń w zależności od liczby próbek w serii, skala logarytmiczna

W kolejnych seriach wielkość zbioru zwiększano o 10. Odnotowano, że przy dużych zbiorach danych program w C++ naruszał ochronę pamięci, a system kończył jego pracę, (i zgłaszając błąd). Zestawienie ogólnej wydajności języków w kolejności od najszybszego wygląda następująco:

1. C++
2. Java
3. Matlab
4. Python

5.2 Zadanie 2 - perceptron wielowarstwowy

*** CHECK ***

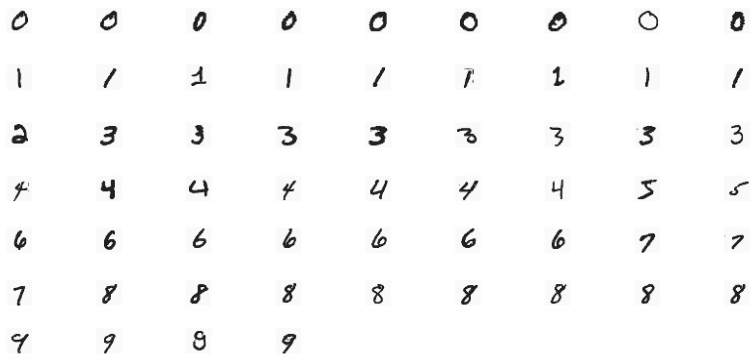
Struktury głębokich sieci neuronowych, takich jak LaNet, AlexNet i inne, mogą różnić się ilością i rodzajem warstw. Istnieje jednak struktura pojawiająca się na wyjściu z sieci prawdopodobnie we wszystkich rozwiązaniach. Strukturą tą jest perceptron wielowarstwowy MLP, na schematach sieci jej warstwy opisane są ”Fully-Connected” lub skrótem ”FC”. Warstwy ukryte perceptronu mogą mieć dowolne funkcje aktywacji: funkcje logistyczną, ReLU lub podobną do ReLU. Jeśli sieć realizuje zadanie klasyfikacji wieloklasowej, wtedy ostatnia wyjściowa warstwa perceptronu jest warstwą Softmax. Głęboka sieć splotowa pracuje na trójwymiarowych strukturach danych zwanych tensorami, perceptron wielowarstwowy przetwarza jednowymiarowe struktury danych. Aby była możliwa współpraca sieci głębokiej i perceptronu wielowarstwowego, potrzebna jest warstwa spłaszczająca, której zadaniem jest konwersja tensora do wektora podczas wnioskowania, i konwersja wektora do tensora w procesie uczenia.

*** /CHECK ***

To zadanie również zaczerpnięto z [2]. Polegało ono na zbudowaniu modelu sieci MLP, a następnie przeprowadzeniu procesu uczenia sieci klasyfikującej cyfry pisanych ręcznie. Założono wykonanie 100 epok uczenia przy wykorzystaniu 50% objętości zbioru danych uczących.

Zaproponowano sieć MLP o strukturze :

- Warstwa 1: 64 perceptronów, wejścia 784, aktywacja - funkcja sigmoidalna
- Warstwa 2: 64 perceptronów, wejścia 64, aktywacja - funkcja sigmoidalna
- Warstwa 3: 10 perceptronów, wejścia 64, aktywacja - funkcja softmax

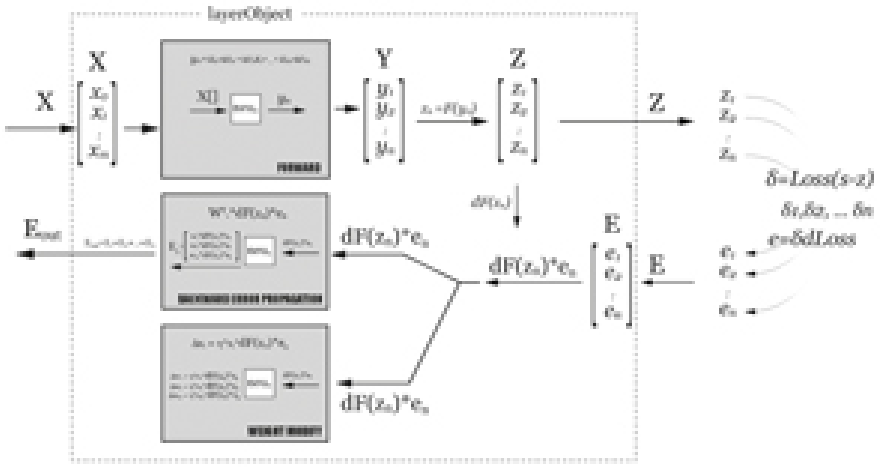


W procesie wykorzystano zestaw obrazów treningowych i testowych ze zbiorów MNIST [22]

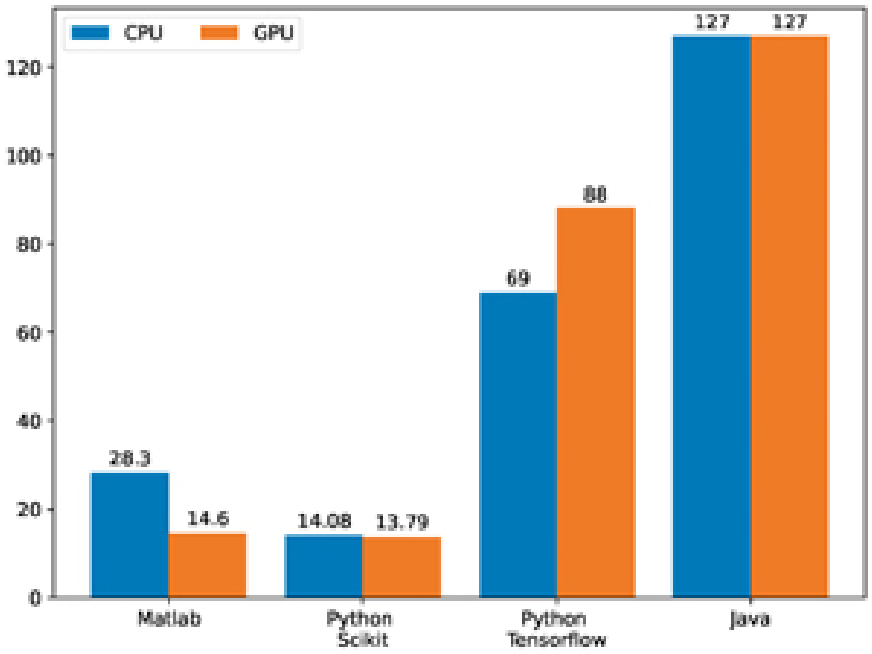
W Matlabie wykorzystano dostępne dodatki, a w Pythonie zainstalowano biblioteki Scikit-learn, zaś w kolejnym rozwiązaniu Tensorflow.

Matlab	Python Scikit-learn

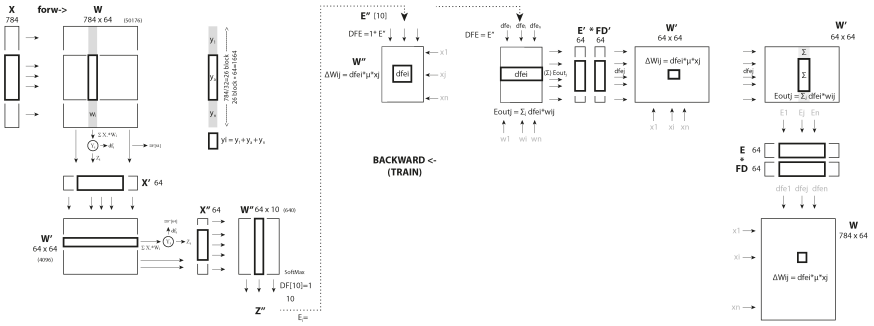
`neurons=64;`
`net = feedforwardnet([64,64],'traingd');`
`net.trainParam.epochs = 100;`
`net.trainParam.goal = 0.0003;`
`net.input.processFcns = {'mapminmax'};`
`net.output.processFcns = {'mapminmax'};`
 	
`gxtrain = gpuArray( xtrain );`
`gytrain = gpuArray( ytrain );`
 	
`net = configure(net,xtrain,ytrain);`
`net = train(net, gxtrain, gytrain, \`
`'useParallel','yes','useGPU','yes');`
	`net = snn.MLPClassifier( \`
	`hidden_layer_sizes=(64,64), \`
	`max_iter=100, \`
	`random_state=1, \`
	`alpha=0.0001, \`
	`early_stopping=False, \`
	`activation='logistic', \`
	`solver='sgd', \`
	`learning_rate='constant', \`
	`learning_rate_init=0.001 )`
	
	`start=time.time()`
	`net.fit( trainX, trainY )`



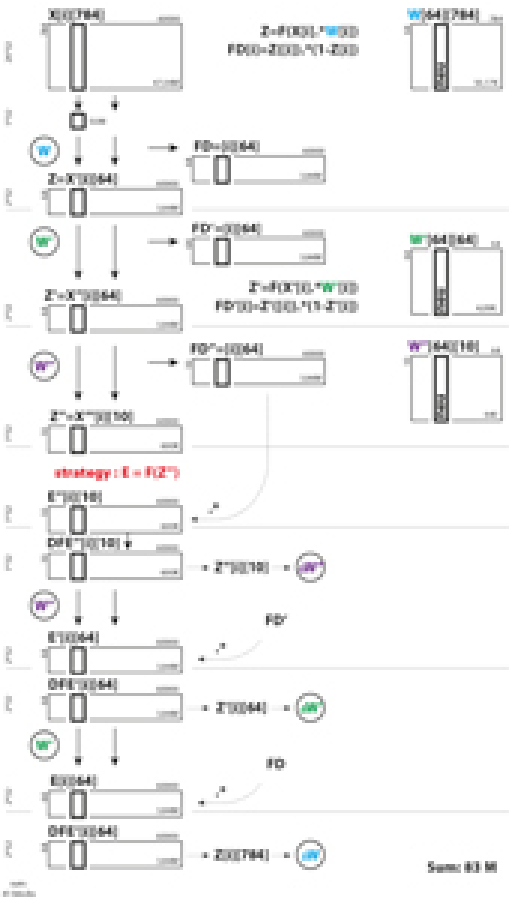
Rozwiązanie w Javie oparto na własnej implementacji sieci MLP. Szczególny nacisk położono na czytelność kodu i idei kosztem wydajności. Wykorzystano wiedzę z poprzednich rozdziałów i napisano kod zgodnie z zasadami programowania obiektowego. Zasady hermetyzacji wymuszają utworzenie dwu współpracujących ze sobą obiektów: neurosumatora oraz warstwy. Własnością warstwy jest rodzaj funkcji aktywacji, a jej metodą jest obliczenie pochodnej funkcji aktywacji w punkcie pracy.



Rysunek 17. Zadanie 2 - zestawienie czasów uczenia perceptronu MLP



Rysunek 18. Zadanie 2 - obliczenia na GPU



Rysunek 19. Zadanie 2 - obliczenia na GPU mapa pamięci

5.3 Zadanie 3 - sieć splotowa

Zadanie polegało na zbudowaniu modelu sieci splotowej CNN połączonej z siecią MLP, wg. struktury LaNet-5 zaproponowanej przez prof. Yann LeCun [14] przedstawionej m.in. w [2].

Przeprowadzeniu procesu uczenia sieci, polegającego na rozpoznawaniu cyfr pisanych ręcznie. Założono wykonanie 100 epok uczenia, przy wykorzystaniu 100% objętości zbioru danych uczących.

Struktura sieci:

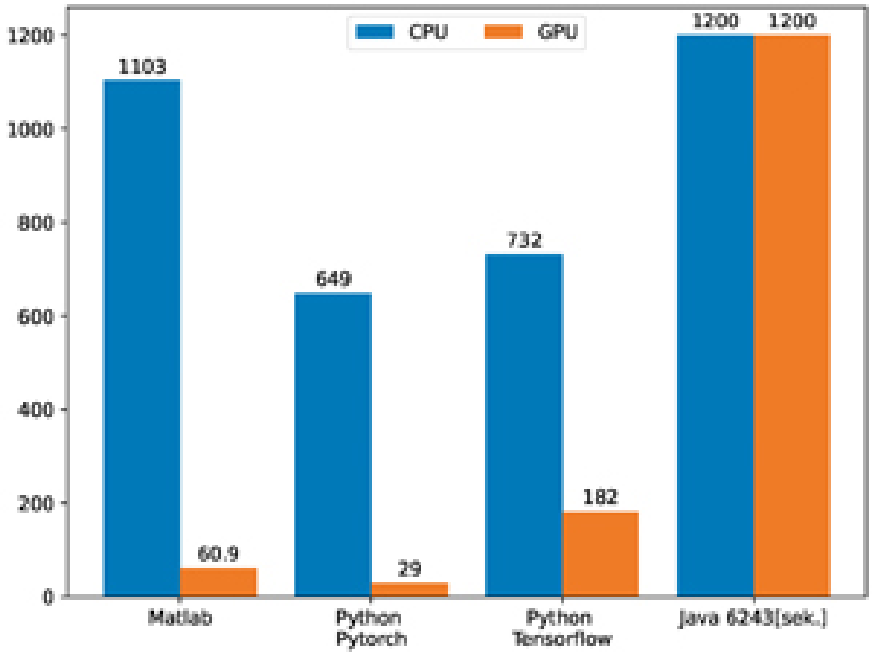
- Warstwa 1: Wejście obraz 28x28 pikseli 1 kanał.
- Warstwa 2: Konwolucyjna, 20 @ filtr 5x5, aktywacja ReLU.
- Warstwa 3: Normalizacja
- Warstwa 4: PoolMax, rozmiar 2x2, krok 2x2
- Warstwa 5: Spłaszczenie obrazu do jednowymiarowej tablicy
- Warstwa 6: MLP: 64 neurony, aktywacja ReLU
- Warstwa 7: MLP: 10 neuronów, wyjście Softmax

Python PyTorch	Python Tensorflow.Keras

```
\\ CITE -> https://www.datacamp.com/tutorial/pytorch-cnn-tutorial
\\ Javier Canales Luna

class CNN(nn.Module):
def __init__(self, in_channels, num_classes):
    super(CNN, self).__init__()
    self.conv1 = nn.Conv2d(in_channels=in_channels, \
    out_channels=20, kernel_size=5, padding=1)
    self.norm  = nn.BatchNorm2d(20)
    self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
    self.fc1 = nn.Linear( 3380, 64 )
    self.fc2 = nn.Linear( 64, num_classes )

def forward(self, x):
    x = F.relu(self.conv1(x))
    x = self.norm(x)
    x = self.pool(x)
    x = x.reshape(x.shape[0], -1)
    x = self.fc1(x)
    x = self.fc2(x)
    return x

start=time.time()
for epoch in range(epochs):
   scores = model(data)
   loss = criterion(scores, targets)
   optimizer.zero_grad()
   loss.backward()
   optimizer.step()
```

```
\\ CITE -> https://keras.io/examples/vision/mnist_convnet/

 def AlexNet():
   NUMBER_OF_CLASSES = 10
   return keras.models.Sequential([
      keras.layers.Input(shape=( 28, 28, 1 )),
      keras.layers.Conv2D(name='conv1', \
      filters=20, kernel_size=(5,5), \
      activation='relu' ),
      keras.layers.BatchNormalization(),
      keras.layers.MaxPool2D(pool_size=(2,2), \
      strides=(2,2)),
      keras.layers.Flatten(),
      keras.layers.Dense(64, activation='relu'),
      keras.layers.Dense(10, activation='softmax')
])
model = AlexNet()
model.compile(optimizer='adam',
  loss='sparse_categorical_crossentropy',
  metrics=['accuracy'])
start=time.time()

with tf.device('/device:GPU:0'):
   model.fit(trainX, trainY, epochs=epochs, verbose=0)
```

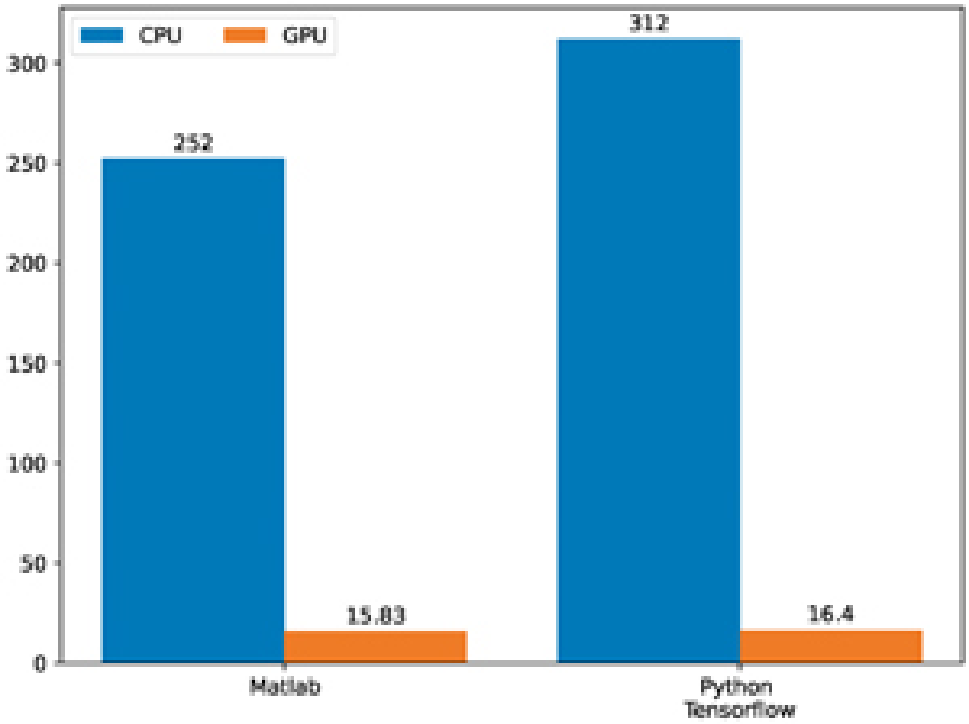
W zadaniu wykorzystano dane z zadania 2, Dwa rozwiązania zostały napisane w Pythonie z wykorzystaniem bibliotek PyTorch oraz Tensorflow. Jedno rozwiązanie powstało w Matlabie z wykorzystaniem bibliotek systemowych. W języku Java napisano własne rozwiązanie, w którym przyjęto te same założenia i strukturę jak w zadaniu 2, rozszerzono działanie klas w projekcie tak by realizowały sieć splotową CNN oraz działanie warstw pomocniczych, ReLU, Flatten, Max i Avg.



Rysunek 20. Zadanie 3 - zestawienie czasów uczenia sieci CNN

5.4 Zadanie 4 - głęboka sieć splotowa

Zadanie polegało na zbudowaniu modelu głębokiej sieci splotowej CNN i nauczeniu sieci rozpoznawania twarzy. Przygotowany zbiór zdjęć uczących i testowych pochodził z zasobów publicznych internetu.

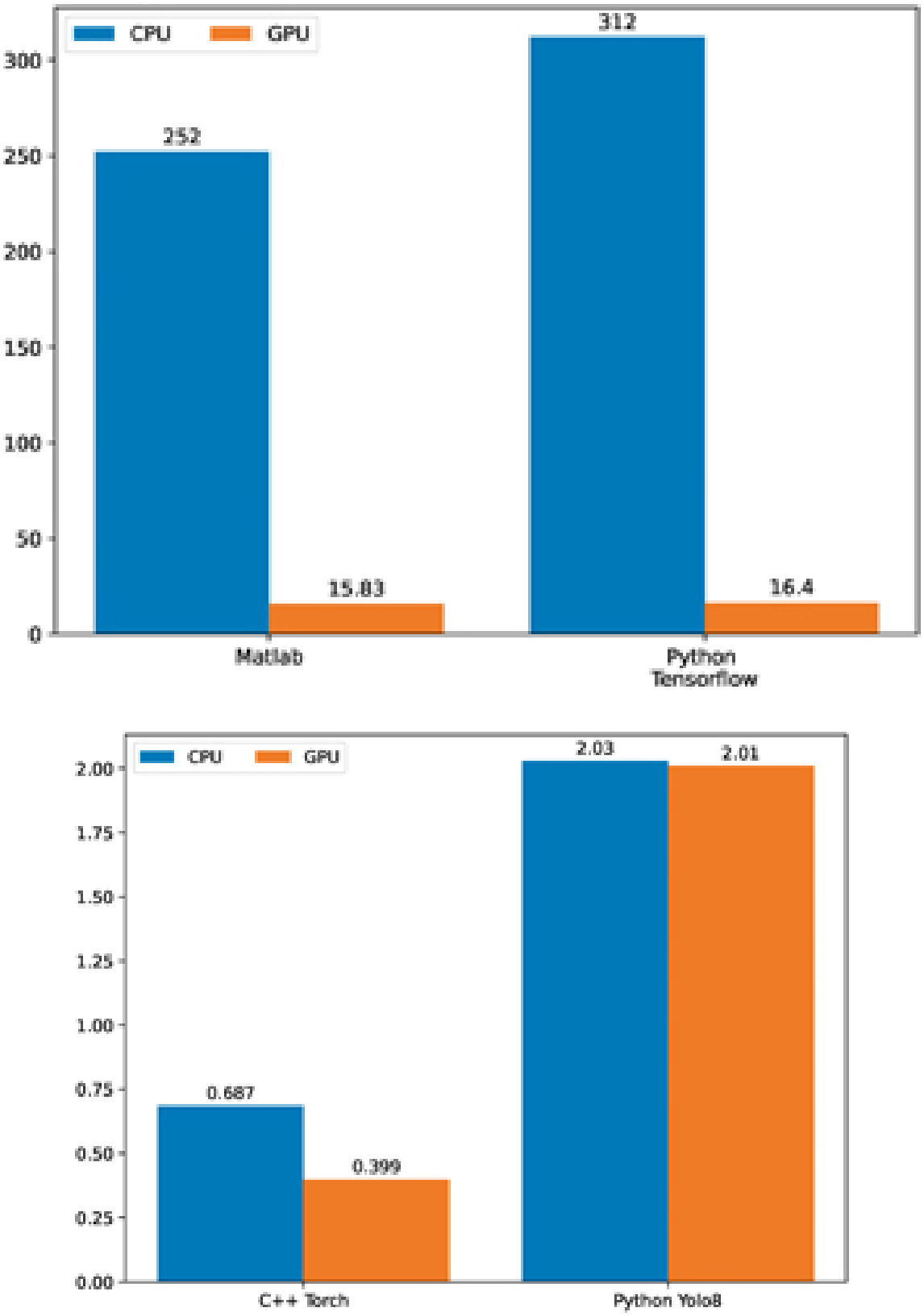


Rysunek 21. Zadanie 4 - zestawienie czasów uczenia sieci głębokiej CNN

Rozwiązanie zrealizowano w dwu językach: Matlab i Python Tensorflow.

5.5 Zadanie 5 - widzenie komputerowe

Zadanie polegało na zbudowaniu aplikacji Yolo8 w językach Python i C++, z wykorzystaniem nauczonego modelu dostarczonych przez ultralytics. Oceniono czas wykonania rozpoznawania obiektów na obrazie. Porównania dokonano tak jak w poprzednich przypadkach w środowisku wyposażonym w kartę graficzną, a następnie powtórzono w tym samym środowisku już bez karty.



Rysunek 22. Zadanie 4 - zestawienie czasów rozpoznawania obiektów Yolo8

Rozdział 6

Porównanie języków

6.1 Aspekty języków

6.1.1 Popularność języka

Duża popularność języka zapewnia łatwy dostęp do dużej społeczności osób pracujących nad jego rozwojem. Dla dynamicznie rozwijających się języków powstaje wiele bibliotek, środowisk IDE, narzędzi, a także literatury i przykładów ich zastosowania w rozwiązywaniu konkretnych zadań. Do zastosowania w projektowanych rozwiązaniach warto wybierać języki popularne i nowoczesne, aby uniknąć trudności ze znalezieniem specjalistów do pracy nad projektem.

6.1.2 Dostępność bibliotek

Dla otwartych języków programowania powstaje wiele narzędzi i bibliotek, z których duża część jest udostępniana społeczności na różne sposoby. Dla rozwiązań komercyjnych bibliotek jest mniej, natomiast ich jakość najczęściej jest bardzo wysoka, dokumentacja dopracowana a pomoc techniczna łatwo dostępna.

6.1.3 Wygoda instalacji

Instalacja języka wraz ze środowiskiem może różnić się znacznie w zależności od systemu operacyjnego. Języki niższych poziomów integrujące się bardziej z systemem operacyjnym mogą być trudniejsze w instalacji i konfiguracji do pracy z np. kartami graficznymi.

6.1.4 Stopień wykorzystania GPU

W modelowaniu sieci głębokich wykorzystanie potencjału kart graficznych ma znaczenie kluczowe dla realizacji wydajnych modeli.

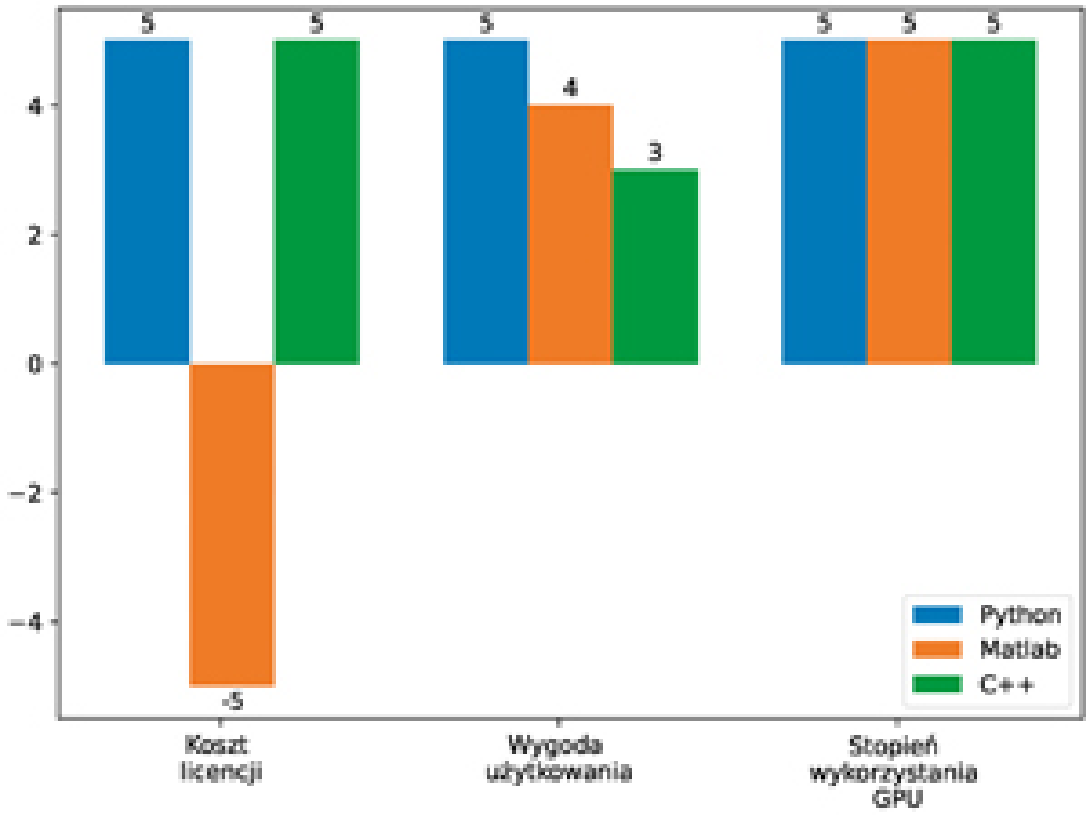
6.1.5 Koszt licencji

Większość języków i środowisk jest dostępna na licencji otwartej. Wykorzystanie niektórych języków, bibliotek czy środowisk IDE wiąże się z koniecznością zakupu licencji. Wykorzystanie w celach komercyjnych może wymagać zakupu innej, droższej licencji niż wykorzystanie w celach naukowych.

6.1.6 Wygoda użytkowania

W większości języków programowania ogólnego przeznaczenia stosuje się bardzo podobne konstrukcje i pracuje się na podobnych rodzajach danych. Języki programowania wysokiego poziomu bazują na bytach abstrakcyjnych, tworzenie kodu jest łatwe i nie wymaga od programisty znajomości dokładnej budowy komputera, znajomości sposobu działania pamięci czy rodzaju procesora na którym będzie uruchamiany program. Nauka programowania w językach wysokiego poziomu jest dużo szybsza niż w językach niższego poziomu jak np. C++, w którym mamy do czynienia ze ścisłą kontrolą typów przy jednoczesnym występowaniu wskaźników, referencji i ramek stosu, czy też w Asemblerze, do programowania w którym niezbędna jest wiedza nie tylko z zakresu np. sposobów reprezentacji liczb w pamięci, przeliczaniu z systemów szesnastkowych na dziesiętne, ale także z zakresu pracy na rejestrach, stosie, wywołaniach przerwań systemowych czy odwołań do funkcji systemu operacyjnego.

6.2 Python



Rysunek 23. Zestawienie aspektów języków

Według bulldogjob.pl najpopularniejszym językiem programowania jest Python [23]. Jest to język młody, jednak prężnie rozwijany i bardzo modny. Składa się na to kilka przyczyn. Po pierwsze - jest to język darmowy (z wyjątkiem specjalistycznych bibliotek). Po drugie - ma bardzo niski próg wejścia. Aby pisać kod w Pythonie, nie trzeba posiadać specjalistycznej wiedzy technicznej ani szczegółowo znać budowy komputera. Naukę programowania w tym języku rozpoczynają już dzieci w wieku szkolnym. Pisanie kodu jest bardzo przyjemne, w kontrolowaniu typów zmiennych pomagają biblioteki narzędziowe. Społeczność użytkowników jest bardzo pomocna, a w sieci znajdziemy dużo przykładów, samouczków i dokumentacji. Istnieje dużo bibliotek, także bibliotek do tworzenia modeli sieci neuronowych. Dla poznawania, trenowania modeli AI nadaje się idealnie. Niestety instalacja systemu z obsługą najnowocześniejszych kart graficznych może być kłopotliwa.

6.2.1 Conda, Anaconda, wirtualne środowiska

Istnieje wiele rozwiązań pomagających zainstalować środowisko, a także budować wiele wirtualnych środowisk na jednej maszynie.

6.2.2 Tensorflow

Biblioteka Tensorflow wymaga rekompilacji do współpracy z GPU. Wersja instalowana z pakietów współpracuje z CPU i wykorzystuje rozkazy procesora AVX. Sprawia trudności przy instalacji,

6.2.3 Scikit-learn

6.2.4 PyTorch

Wygodna do zainstalowania, bardzo współpracuje bardzo dobrze z procesorami graficznymi zarówno od NVidia, ale także może wykorzystać karty AMD.

6.3 Matlab

Środowisko Matlab nie jest ujęte w zestawieniu bulldogjob.pl [23]. Jest to środowisko płatne, tworzone i wykorzystywane przez środowiska akademickie. Jest przykładem wzorowej obsługi procesu instalacji oprogramowania i wsparcia obsługi kart graficznych. Pisanie kodu jest wygodne, a dostarczona dokumentacja i przykłady są obszerne i wygodne w użyciu. Jedynym, co może być mylące jest rozpoczynanie indeksów od 1.
Dodatek Parallel computing w Matlab umożliwia wykonywanie obliczeń równolegle, w osobnych wątkach, procesach, a także z wykorzystaniem kart graficznych (obecnie tylko NVidia). Dodatki Optimalization Toolbox i Optimalization Computing Toolbox optymalizują tworzony kod, dzięki czemu zwiększają jego wydajność. Dodatek Deep Learning Toolbox dostarcza gotowych metod do obliczeń sieci neuronowych.

6.4 C++

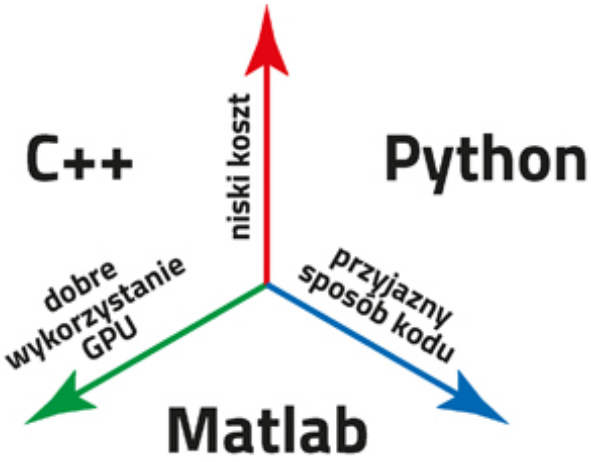
Język C/C++ wg. bulldogjob.pl osiąga popularność na poziomie 6.6% [23]. C++ jest językiem, który nieco traci popularność na korzyść np. Pythona czy Rust. Jest to język darmowy, a integracja ze sterownikami i bibliotekami kart graficznych jest natywna. Idealnie pracuje w systemach Linux, ale dobrze działa także w Windows z Microsoft Visual Studio. Jest to język kompilowany, w którym budowanie programu trwa dłużej niż w Pythonie czy Matlabie, jednak skompilowana aplikacja działa bardzo szybko. Język ten dobrze obsługuje karty graficzne. Jednak stawia duże wymagania użytkownikowi piszącemu kod. Do dobrego wykorzystania potencjału wymaga rozumienia procesów zachodzących w sprzęcie, znajomości wielu pojęć języka oraz dobrego poruszania się w typach danych.

6.5 Java

Według bulldogjob.pl wskaźnik popularności dla języka Java wynosi 26.2% [23]. Java jest językiem podobnym do C++, jest to język kompilowany do kodu bajtowego uruchamianego w maszynie wirtu-

alnej. Ma ścisłą kontrolę typów. Niektóre wersje wymagają opłacenia licencji, większość jednak jest darmowa. Wykonywalny kod działa dość szybko, nieco wolniej niż C++. Instalacja wymaga nieco uwagi. Istnieją biblioteki umożliwiające wsparcie obliczeń na kartach graficznych, jednak w tej pracy nie wykorzystywano tych bibliotek. Jest rzadko wykorzystywany do modelowania sieci neuronowych.

6.6 Podsumowanie i wnioski



Rysunek 24. Priorytety przy wyborze języka

Przy wyborze języka trzeba zdecydować się na dwa z trzech priorytetów, jeden niestety trzeba odrzucić. Najwydajniejsze języki to C++ i Matlab. Najwygodniej pracuje się w Pythonie więc jest najczęściej wybierany do tworzenia modeli w celach badawczych. Duże produkcyjne modele powstają zwykle w języku C/C++ i są uruchamiane w środowiskach serwerowych.

6.7 Dodatkowe wnioski

6.7.1 Zadanie 3

Zbliżone czasy wykonania zadania 3 dla PyTorch z użyciem GPU i bez - sugerują, że możliwości GPU nie zostały użyte prawidłowo.

Poprawa kodu powoduje zdecydowane skrócenie czasu wykonania, jednocześnie zmienia wnioski wynikające z zestawienia:

```
-------------
CUDA available. Using GPU acceleration.
cuda
(59400, 1, 28, 28)
tensor([13.2702, -8.9589,  2.4638,  0.5280, -4.6336,
  2.0661,  0.6992,  0.2631, 0.6639,  0.2546],
  device='cuda:0', grad_fn=<SelectBackward0>)

# Python PyTorch 2.0 60000 Images, 100 Epoch
  Time:  29.587559938430786
  Test accuracy: 0.6212121248245239
------------------
```

Rozdział 7

uwagi

Komentarze do pracy 2 Wydajność - wydajność czego? 2.1 Nie rozumiem po co wspominać o tym i to zaledwie w 2 zdaniach? (Maszyna Turinga) 2.2 Czy chodzi wyłącznie o translację punktów czy raczej o coś więcej (macierze na to wskazują). Nie wiem jak rozpisany wzór ma się do tych macierzy. Ogólnie rozdział 2 jest niezrozumiały. Wydaje mi się, że dostrzegam ideę, czyli pokazanie sposobów realizacji podstawowych obliczeń dodawania i mnożenia (których to operacji w wielu dziedzinach trzeba wykonywać miliardy czy biliony na sekundę), ale to co jest napisane jest niezrozumiałe, powyrywane z kontekstu, fragmentaryczne. Raczej trzeba wyjść od zdefiniowania tej potrzeby i potem pokazać, jak to w kolejnych etapach rozwoju komputerów robiono. ()

3 (Model perceptronu warstwowego MLP) Nie można zaczynać rozdziału od rysunku. Perceptron jest WIELOwarstwowy W ogóle nie wiadomo po co o tym jest napisane. To jakoś trzeba uzasadnić.

4 (Modele sieci splotowych CNN) Też nie wiadomo dlaczego nagle o tego rodzaju sieci Pan wspomina. We wszystkich tych rozdziałach brakuje jakiegoś wprowadzenia, które uzasadniłoby pisanie o tym. Plus te 3 rozdziały należy wrzucić do jednego, bo to raptem 11 stron jest, w tym sporo rysunków.

5.1 Kod w Javie opisany jako Python. Brakuje kodu własnej implementacji w Pythonie i Matlabie, nie wiadomo co to JavaBigDecimal. na wykresie jest C, a nie C++. Choć ten kod to w sumie C, a nie C++. DLa C++ powszechnie używana jest biblioteka Eigen, w której jest m.in. regresja liniowa. Skoro sprawdza Pan funkcję ”wbudowaną” dla Pythona, to wypadałoby sprawdzić analogiczną dla C++.

5.2 Znowu niejasności i nie wiadomo dlaczego tak, a nie inaczej, oraz jak. Są 2 fragmenty kodu, ale sprawdzanych wersji kodu było znacznie więcej. Nie wiadomo więc co i jak tam było zrobione. W szczególności nie wiadomo jak uruchomił Pan sci-kit na GPU. Normalnie nie bardzo się da chyba. Ogólnie eksperyment pokazuje, że nie warto angażować GPU do relatywnie prostych sieci i małych zbiorów danych.

5.3, 5.4 Ogólnie pokazują, że dla większych sieci GPU się opłaca 5.5 nie rozumiem podpisu C++ Torch? Yolo8 w pythonie działa tak samo szybko na CPU i GPU? Coś tu nie gra. Sprawdziłem i jest kilkukrotna (zależy od sieci)

Na razie praca, w bardzo chaotyczny i niejasny sposób, pokazuje (i to nie zawsze), że GPU opłaca się dla większych sieci, a dla mniejszych nie. Zasadnicze pytanie: w jakim języku najlepiej napisać

kod dla głębokich sieci neuronowych pozostaje bez odpowiedzi. Za mało jest tu eksperymentów i są one praktycznie nieopisane. O rozpoznawaniu twarzy są 2 zdania! A powinno być 10 stron. W założeniach miało to być zrobione w Matlabie, w Pythonie i jeszcze w czymś. W zakresie ZABRAKŁO wyszczególnienia pytorcha, jako że torch i tensorflow to dwie podstawowe biblioteki do głębokiego uczenia. Tu brakuje też rozróżnienia czasu uczenia i czasu wnioskowania. Podejrzany jest też fakt, że Matlab jest szybszy. No ale kompletnie nie wiadomo co tam było robione i jak. I tak wygląda cała praca... To MUSI zostać skonkretyzowane. Trzeba pokazać cały proces w Matlabie, w Pythonie (najlepiej z tensorflow i z pytorchem) oraz ewentualnie jeszcze w torchc++ (o ile da się to łatwo uruchomić). Czyli potrzebny jest kod, przebieg uczenia, czas uczenia, czas wnioskowania, porównanie dokładności sieci, potem ocena łatwości przeprowadzenia całego procesu - w jakim środowisku jest to najprostsze, najprzyjaźniejsze, w jakim mamy większą kontrolę nad tym co robimy. I tak dla kilku zadań wymienionych w zakresie pracy. Wówczas będzie można porównać te języki

Bibliografia

[1] Józef Korbicz Andrzej Obuchowicz, D. U., *Sztuczne sieci neuronowe: podstawy i zastosowania*. Akademicka Oficyna wydawnicza PLJ, 1994, ISBN: 83-7101-197-0.

[2] Stanisław Osowski, R. S., *Matematyczne modele uczenia maszynowego w językach MATLAB i PYTHON*. Oficyna Wydawnicza Politechniki Warszawskiej, 2023, ISBN: 978-83-8156-597-4.

[3] Osowski, S., *Sieci neuronowe do przetwarzania informacji*. Oficyna Wydawnicza Politechniki Warszawskiej, 2020, ISBN: 978-83-7814-923-1.

[4] Rasheed, A. F. i Zarkoosh, M., *Unveiling Derivatives in Deep and Convolutional Neural Networks: A Guide to Understanding and Optimization*, working paper or preprint, 2024. DOI: 10.36227/techrxiv.170491744.44652991/v1. adr.:]href%20%7B%20https://hal.science/hal-04409232%7D.

[5] Vincent Dumoulin, F. i †, F. V., *A guide to convolution arithmetic for deep learning*, 2018. adr.: https://arxiv.org/pdf/1603.07285.

[6] Jefkine, *Backpropagation In Convolutional Neural Networks*, 2016. adr.: https://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/.

[7] Mieczysława Muraszkiewicza i Roberta Nowaka, praca zbiorowa pod redakcją, *Sztuczna inteligencja dla inżynierów*. Oficyna Wydawnicza Politechniki Warszawskiej, 2023, ISBN: 978-83-8156-584-4.

[8] Stuart Russell, P. N., *Artificial Intelligence: A Modern Aproach, 4th Edition*, Grażyński, T. A., red. Pearson Education, Inc., Polish language by Helion S.A. 2023, 2023, ISBN: 978-83-283-7773-8.

[9] Sarah Guido, A. M., *Machine learning, Python i data science*. Helion S.A., 2021, ISBN: 978-83-8322-751-1.

[10] Sebastian Raschka, V. M., *Machine learning, Python i data science*. Helion S.A., 2021, ISBN: 978-83-283-7001-2.

[11] Jocher, G., Chaurasia, A. i Qiu, J., *Ultralytics YOLOv8*, wer. 8.0.0, 2023. adr.: https://github.com/ultralytics/ultralytics.

[12] Adr.: https://docs.ultralytics.com/models/yolov8/#how-do-i-train-a-yolov8-model.

[13] Adr.: https://github.com/ultralytics/ultralytics/tree/main/examples/YOLOv8-CPP-Inference.

[14] Y. Bengio - Université de Montréal, Y. L. N. Y. U., *Convolutional Networks for Images, Speech, and Time-Series*, https://www.researchgate.net, 1997.

[15] Qi, H., „Derivation of Backpropagation in Convolutional Neural Network (CNN)", 2016. adr.: https://api.semanticscholar.org/CorpusID:37819922.

[16] Caceres, P. adr.: https://com-cog-book.github.io/com-cog-book/features/cov-net.html.

[17] Hoey, J. V., *Programowanie w asemblerze x64*, Werner, T. G., red. Helion S.A., 2024, ISBN: 978-83-289-0109-4.

[18] Kasprzak, W., *Metody sztucznej inteligencji C6.pdf*, materiał dydaktyczny, 2024.

[19] Adr.: https://www.youtube.com/watch?v=x9Scb5Mku1g.

[20] Adr.: http://www.cs.put.poznan.pl/rwalkowiak/pliki/2019/pr/programowanie_PKG_CUDA_2019.pdf.

[21] Adr.: https://zapisy.ii.uni.wroc.pl/courses/kurs-obliczenia-rownolege-na-kartach-graficznych-cuda-q2-201920-zimowy.

[22] Cun, Y. A. L. adr.: http://yann.lecun.com/exdb/mnist/.

[23] bulldogjob.pl. adr.: https://bulldogjob.pl/it-report/2025/technologie.