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 przetwarzaniu obrazów

Piotr Heinzelman

numer albumu 146703

promotor

dr inż. Witold Czajewski

WARSZAWA 2025

Porównanie wydajności wybranych języków programowania w realizacji sieci neuronowych do przetwarzaniu 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

In this work, the issues of creating high -performance computing systems have been raised for the implementation of deep neural networks models. Large models require effective ways calculating the network response and conducting the learning process, because as the size increases.

The model is clearly increasing. Ways to increase the efficiency of the equipment, especially those that can be used in neural networks models. In the introduction, mathematical models of the CNN and MLP networks were introduced, and then launched Implementation of these models as a sequence of arithmetic activities. Attention was paid to the possibility Increasing performance by performing processing operations in a parallel manner. The tasks set for compared models were briefly presented. The work made comparison of such language properties as: popularity, availability of libraries, Installation convenience, support for calculations on graphic cards, language difficulty level, license cost. Code fragments were also presented, and experiences while writing the code were described. Selected languages were discussed: Python, Maltab, Java, C ++. To build the YOLO8 network was used Libraries: Torch, Ultralytics. The CNN network designs were used: tensorflow, scikit-learn, pythorch. The DEEP Learning Toolbox package was used in Matlab projects. Java was used in Java implementation. The summary collected the results, and the premises that can be helpful at selection of language to implement deep neural networks.

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

Spis treści

1 Wstęp **9**
- 1.1 Cel pracy . . . 10
- 1.2 Układ pracy . . . 10
- 1.3 Kod źródłowy i dane uczące . . . 11

2 Wydajność **13**
- 2.1 Maszyna Turinga . . . 13
- 2.2 Szybsze obliczenia grafiki 3D . . . 13
  - 2.2.1 Funkcje procesora MMX, SIMD, SSE, AVX . . . 14
- 2.3 Obliczenia na GPU . . . 15
  - 2.3.1 Równoległe mnożenie . . . 15
  - 2.3.2 Wielokanałowe dodawanie . . . 15
- 2.4 Matlab . . . 16

3 Model perceprtonu warstwowego MLP **17**
- 3.1 Perceptron warstwowy . . . 18
- 3.2 Proces uczenia MLP . . . 19

4 Modele sieci splotowych CNN **21**
- 4.1 Wstęp . . . 21
- 4.2 Operacja splotu . . . 23
  - 4.2.1 Padding . . . 23
  - 4.2.2 Stride . . . 24
- 4.3 ReLU i warstwy redukujące wymiar . . . 25
- 4.4 Przetwarzanie w przód . . . 26
- 4.5 Przetwarzanie wstecz - modyfikacja filtra w warstwie splotowej . . . 26
- 4.6 Propagacja błędu przez warstwę splotową . . . 26

5 Zadania **27**
- 5.1 Zadanie 1 - regresia liniowa . . . 27
- 5.2 Zadanie 2 - perceptron warstwowy . . . 29
- 5.3 Zadanie 3 - sieć splotowa . . . 31

5.4 Zadanie 4 - głęboka sieć splotowa . . . 32
5.5 Zadanie 5 - widzenie komputerowe . . . 33

6 Porównanie języków **35**
6.1 Aspekty języków . . . 35
6.1.1 Popularność języka . . . 35
6.1.2 Dostępność bibliotek . . . 35
6.1.3 Wygoda instalacji . . . 35
6.1.4 Stopień wykorzystania GPU . . . 35
6.1.5 Koszt licencji . . . 36
6.1.6 Wygoda użytkowania . . . 36
6.2 Python . . . 37
6.3 Matlab . . . 38
6.4 C++ . . . 38
6.5 Java . . . 38
6.6 Podsumowanie i wnioski . . . 39

Bibliografia **41**

Spis załączników **43**
6.7 Model klasyczny - podejście matematyczne . . . 43
6.7.1 Funkcje aktywacji . . . 45
6.7.2 Proces uczenia . . . 46
6.7.3 Algorytm propagacji wstecznej . . . 46

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 [12]. Na wybór konkretnego tematu i dobór języków miała wpływ także lektura [21] oraz [15], [18], [23], [10].

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. [12]

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. [15]

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)[21]

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. [21]

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 [12] oraz CNN [14] [22] 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: [21] [15] [19] [20]
- instrukcji i przykładów załączonych do bibliotek
- otwartych repozytoriów git [11] [1] [2]

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

Wydajność

2.1 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.

2.2 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:

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

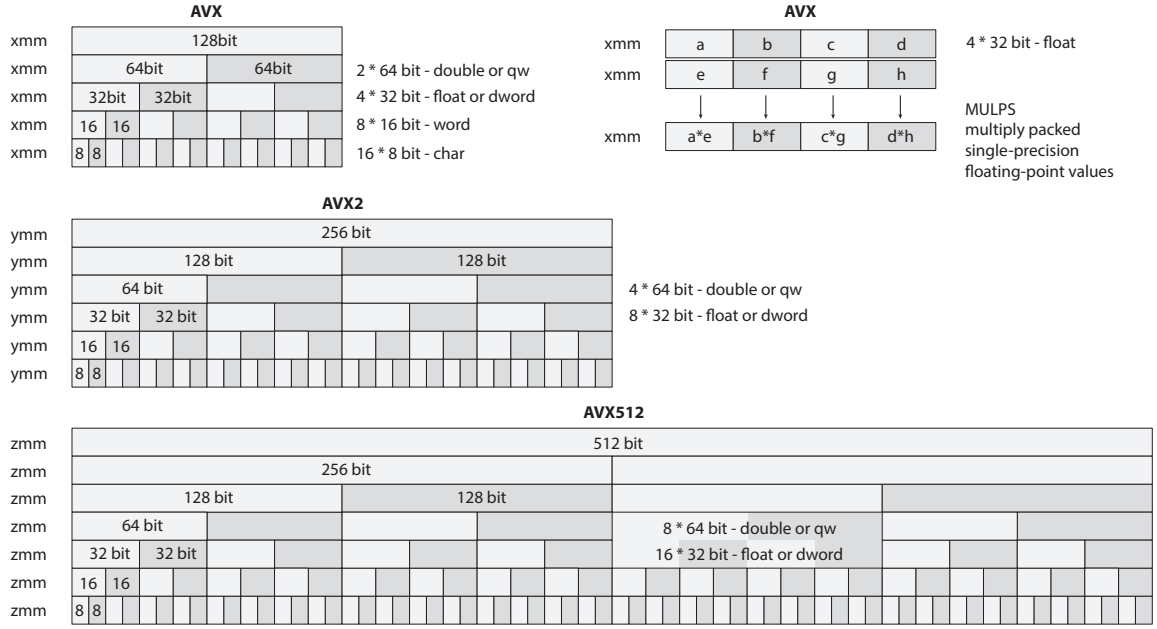
a po rozpisaniu mamy:

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

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.

2.2.1 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).[9] 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.[9]



Rysunek 1. Rejestry AVX, AVX2 i AVX512,operacja równoległego mnożenia AVX

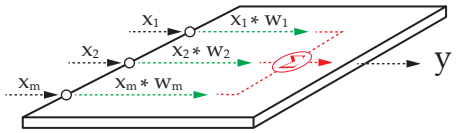
Rozkaz MULPS wykonuje jednocześnie jednocześnie 4 mnożenia zestawu 8 liczb 32 bitowych. Instrukcje w AVX512 VMPADDWD, VMPADDUBSW - wykonują mnożenie wektorowe, następnie dodawanie sąsiednich elementów. $(\mathrm{a_i} \cdot \mathrm{b_i} + \mathrm{a_{i+1}} \cdot \mathrm{b_{i+1}})$

Funkcje te przyspieszają operacje iloczynu Hadamarda (.*)[13] 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. [3]

2.3 Obliczenia na GPU

2.3.1 Równoległe mnożenie

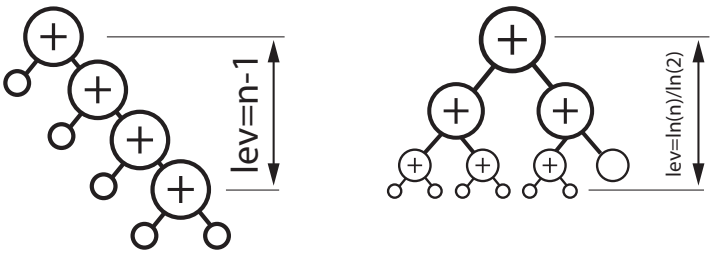
Wydajne obliczanie iloczynu Hadamarda (.*) wektorów o dużych rozmiarach mogą zapewnić procesory graficzne, w których znajdują się tysiące jednostek obliczeniowych. Czas wykonania kilku tysięcy mnożeń zajmuje tyle samo czasu co wykonanie jednego mnożenia. Użycie kart graficznych przy modelowaniu głębokich 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. [4] [5]



Rysunek 2. Mnożenie równoległe i wielokanałowe dodawanie

2.3.2 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 3. drzewo zdegenerowane i niezdegenerowane

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

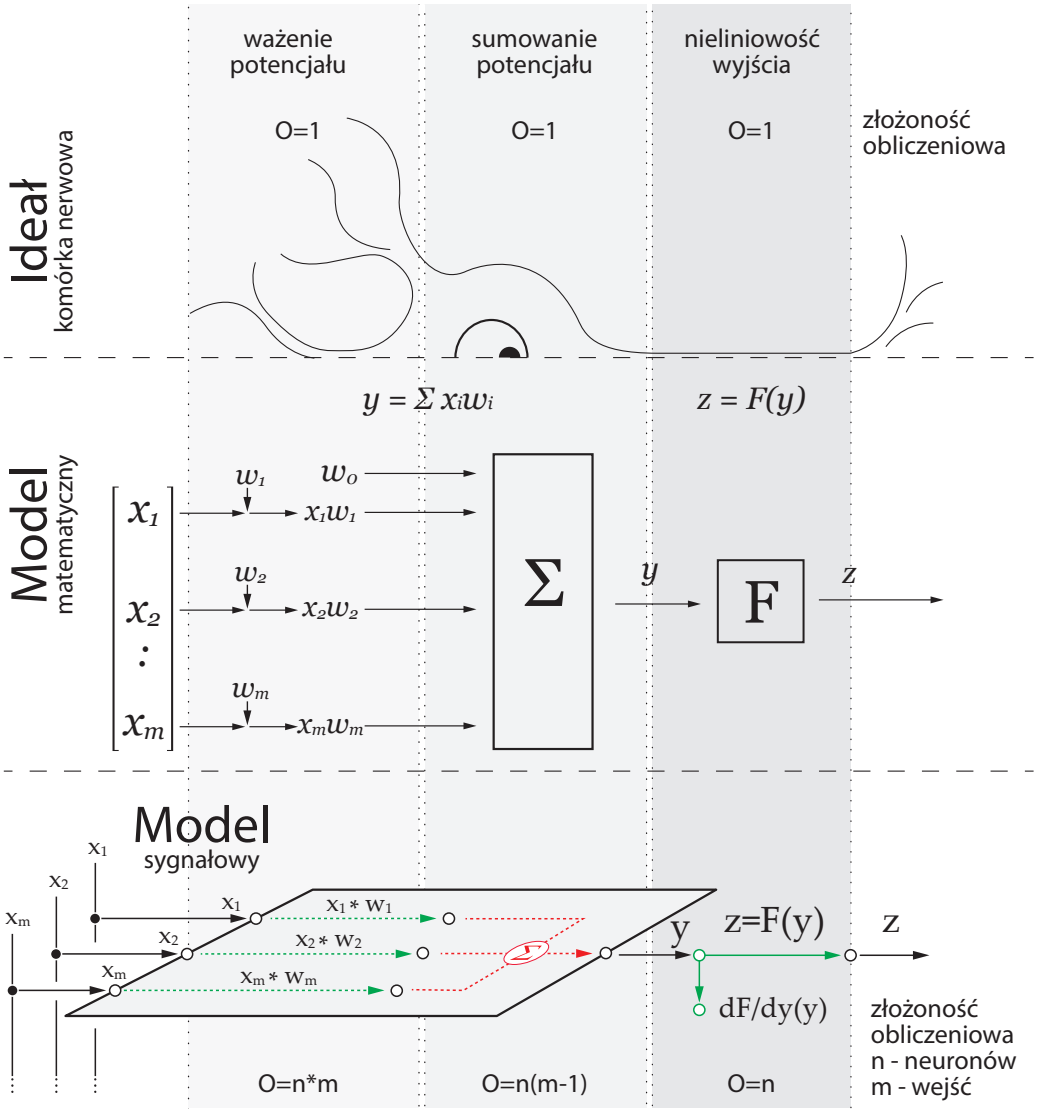
Uzyskamy dwukrotne zwiększenie szybkości przez dodanie nawiasów. Przy 32 składnikach mamy 16 + 8 + 4 + 2 + 1 dodawań, jednak pierwsze 16 może być wykonane równolegle, kolejna 8 także jest od siebie niezależna, podobnie 4 i 2. Czyli dla 32 elementów mamy 31 dodawań w 5 poziomach. Dla 1024 składników mamy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

2.4 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 3

Model perceprtonu warstwowego MLP



Rysunek 4. Neuron, model matematyczny i sygnałowy

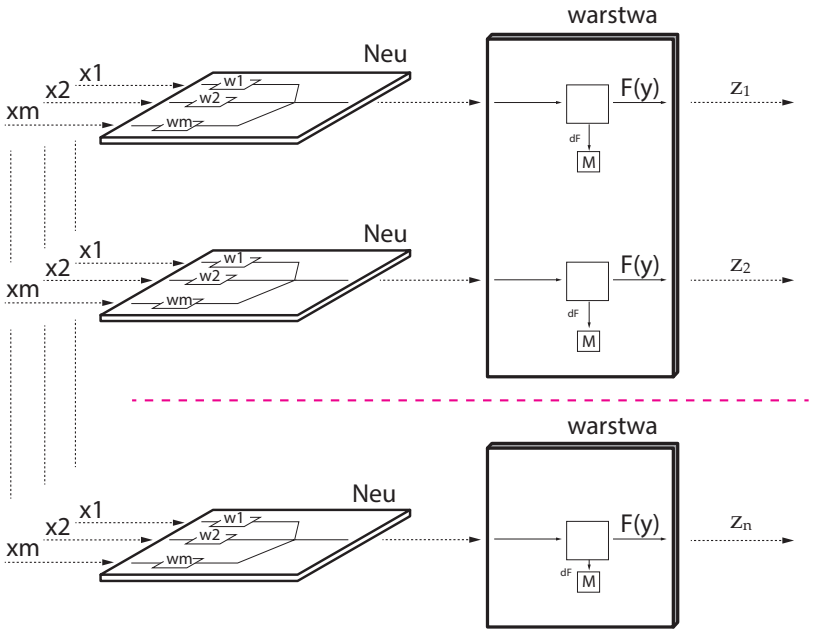
Neuron biologiczny przetwarza impulsy wejściowe w trzech krokach:

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

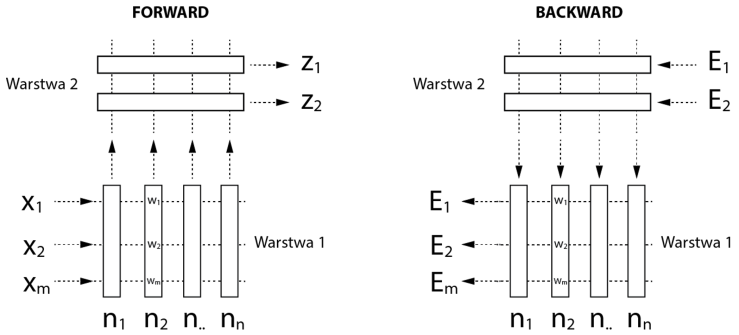
rodzaj warstwy	funkcja aktywacji $\mathrm{z} = \mathrm{F(y)}$	wartość pochodnej $\frac{\partial \mathrm{F}}{\partial \mathrm{y}}$
logistyczna (sigmoid)	$\frac{1}{1+\mathrm{e}^{-\mathrm{y}}}$	$(\mathrm{z})(1-\mathrm{z})$
ReLU	jeśli y<0 to z=y, jeśli y>=0 to z=0	jeśli y<0 to 1, jeśli y>=0 to 0
softmax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

3.1 Perceptron warstwowy

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

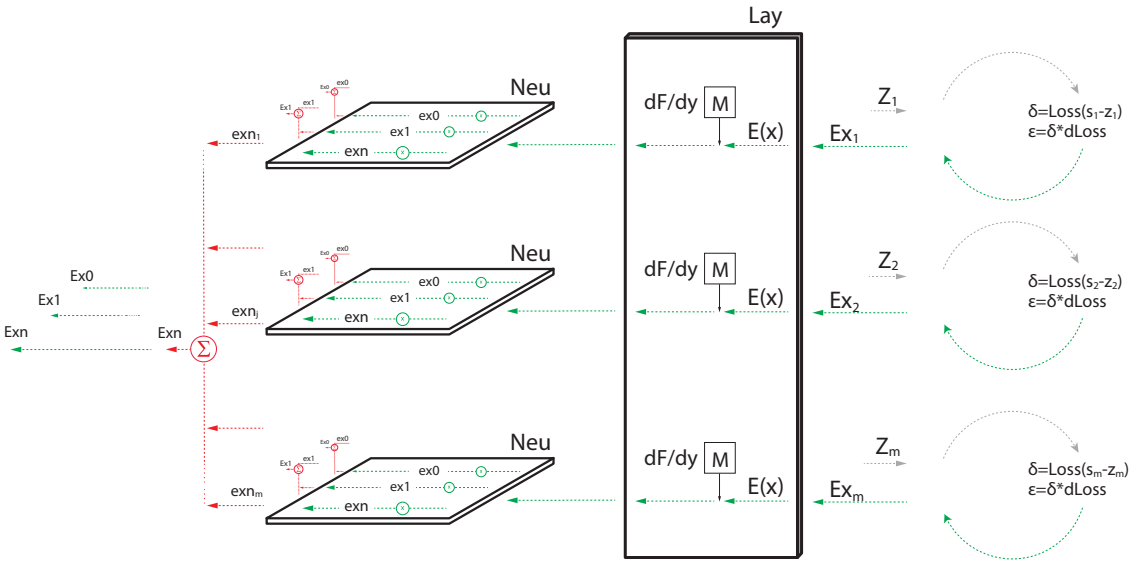


Rysunek 5. Przepływ sygnałów przez warstwę w czasie propagacji sygnału "w przód"



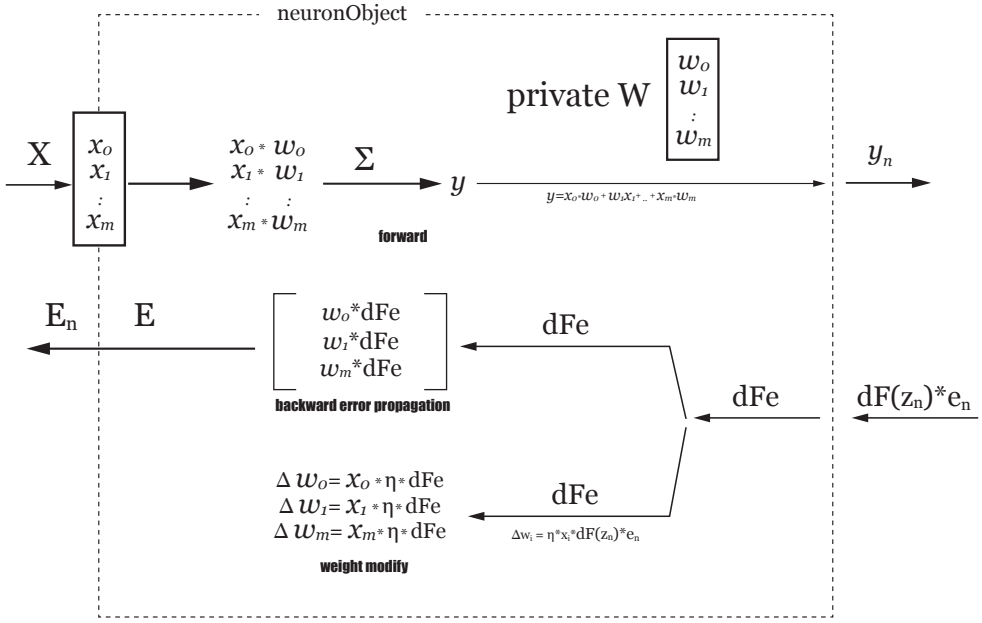
Rysunek 6. Propagacja sygnału przez zespół warstw w przód i wstecz

3.2 Proces uczenia MLP



Rysunek 7. Propagacja wstecz - uczenie sieci. (Operacje niezależne (mnożenie) oznaczone kolorem zielonym i operacje wymagające synchronizacji (dodawanie) oznaczone kolorem czerwonym)

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



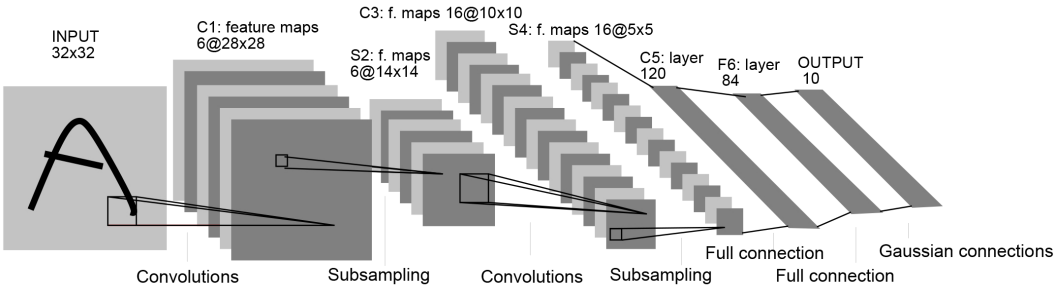
Rysunek 8. Przepływu sygnałów w obiekcie Neurosumator

Rozdział 4

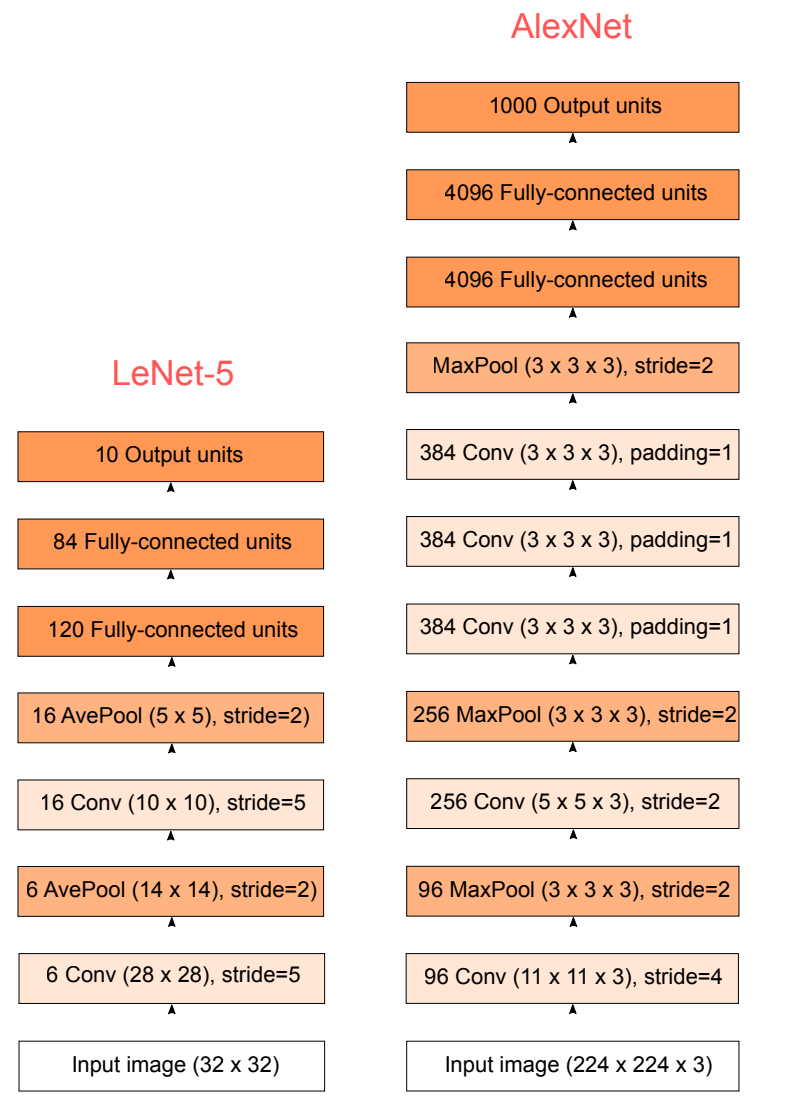
Modele sieci splotowych CNN

4.1 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 [21]. Przetwarzania obrazu przez zestaw filtrów i warstw generuje obrazy o coraz mniejszych wymiarach, lecz o zwiększającej się liczebności kanałów reprezentujących cechy charakretystyczne 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 [24] [17].



Rysunek 9. Architektura LaNet-5 [24]

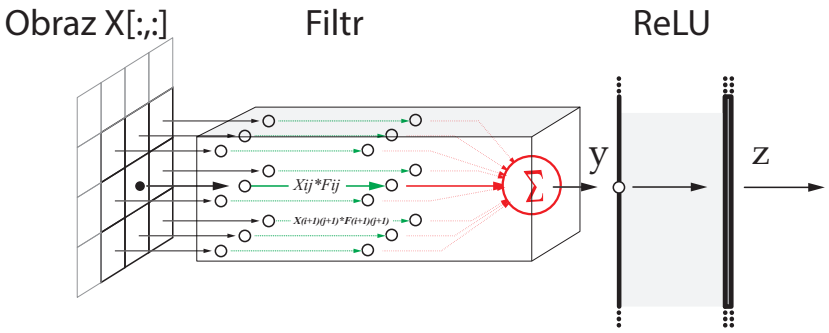


Rysunek 10. Porównanie struktury LaNet i AlexNet [7]

4.2 Operacja splotu

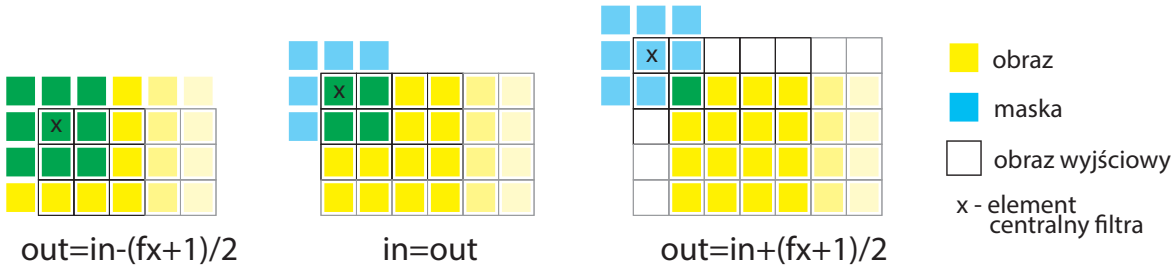
Jeśli wartości pikseli obrazu wejściowego X oznaczymy $\mathrm{X}_{(\mathrm{i,j})}$, a wartości filtra F oznaczymy jako $\mathrm{F}_{(\mathrm{m,n})}$ obraz wejściowy Y, $\mathrm{Y}_{(\mathrm{o,p})}$, wówczas każdy piksel obrazu wyjściowego obliczymy:

$$\mathrm{Y}_{(\mathrm{O,P})} = \sum_{(\mathrm{n}=1)}^{(\mathrm{N})} \sum_{(\mathrm{m}=1)}^{(\mathrm{M})} \mathrm{F}_{(\mathrm{m,n})} * \mathrm{X}_{(\mathrm{m+O,n+P})} \tag{1}$$



Rysunek 11. 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.

4.2.1 Padding



Rysunek 12. Wpływ punktu startowego maski na wymiar obrazu wynikowego operacji splotu

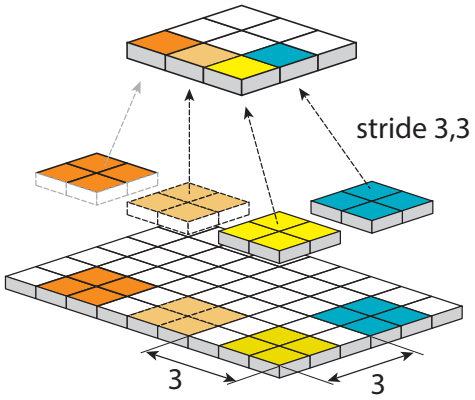
Padding oznacza wysunięcie maski poza obszar obrazu.

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).

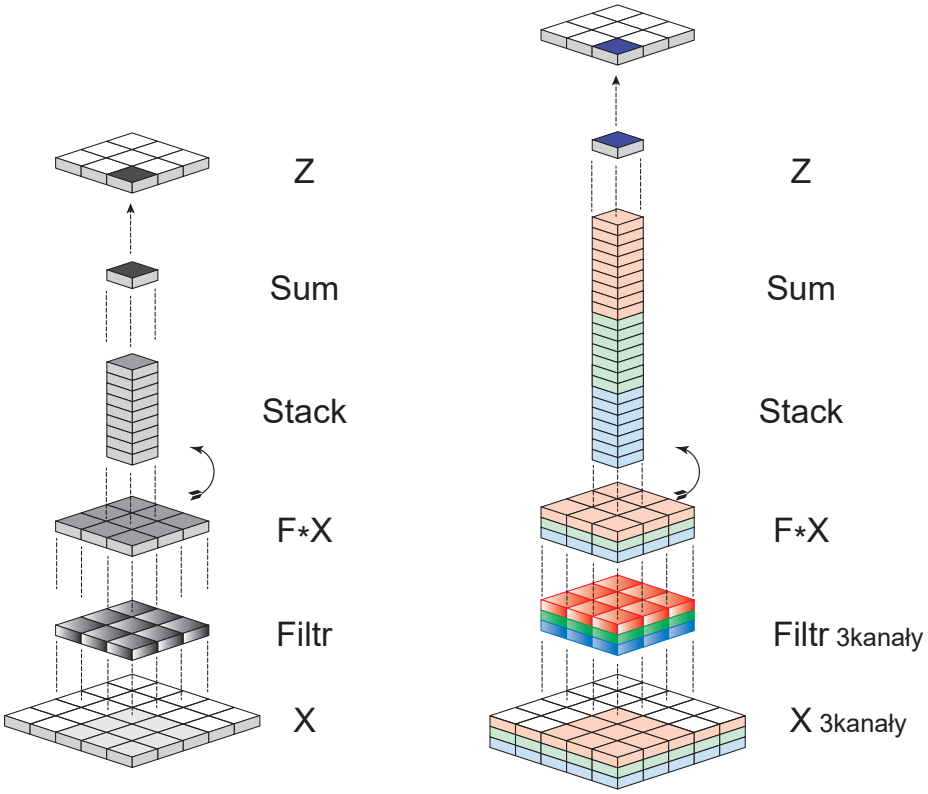
W założeniu procesu uczenia, każdy przesyłany sygnał wejściowy i wyjściowy jest sprzężony z powrotnym sygnałem określającym wielkość błędu tego sygnału. Zatem jeśli w operacji splotu padding jest dodatni, wówczas przy obliczeniu błędu wyjściowego zastosowany będzie padding ujemny.

4.2.2 Stride



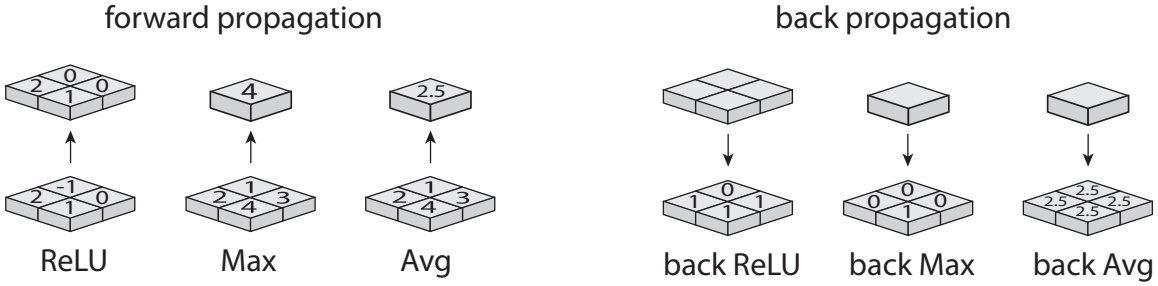
Rysunek 13. Stride, krok przesunięcia maski

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



Rysunek 14. Konwolucja jedno i wielokanałowa

4.3 ReLU i warstwy redukujące wymiar



Rysunek 15. 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 $1/\mathrm{n}^2$ 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.

4.4 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.

4.5 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.

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

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

4.6 Propagacja błędu przez warstwę splotową

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

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

Rozdział 5

Zadania

5.1 Zadanie 1 - regresia liniowa

Zadanie zaczerpnięto z [21] 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:

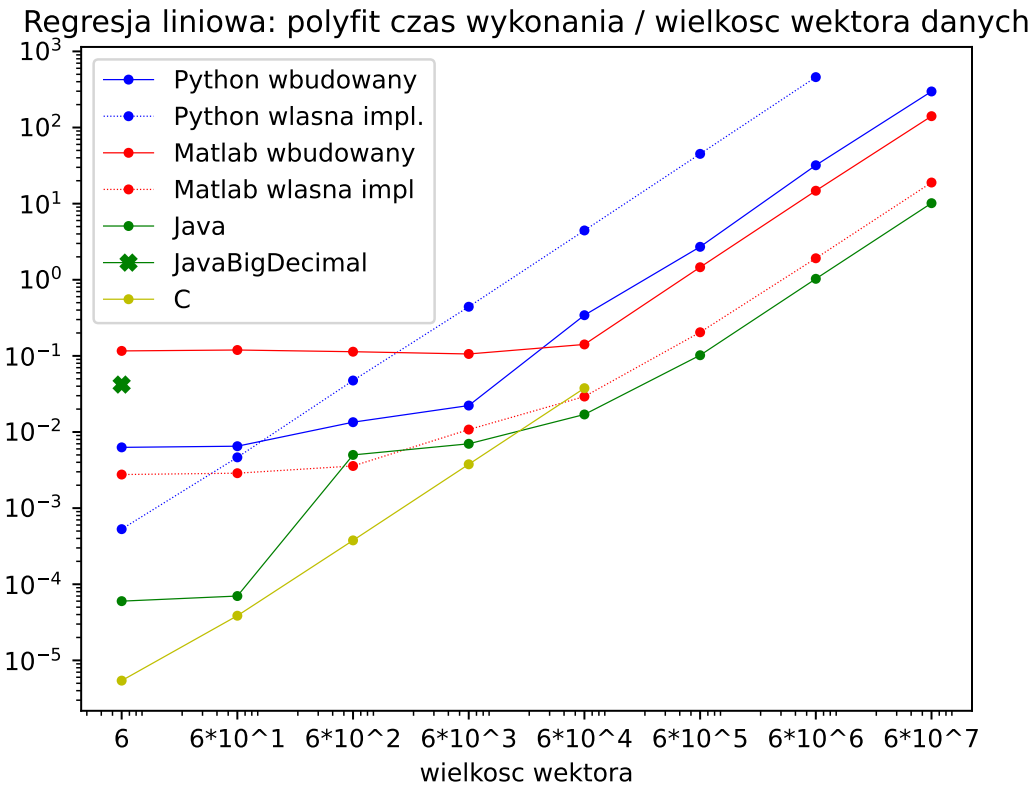
Matlab

```
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 warstwowy

To zadanie również zaczerpnięto z [21]. 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 :

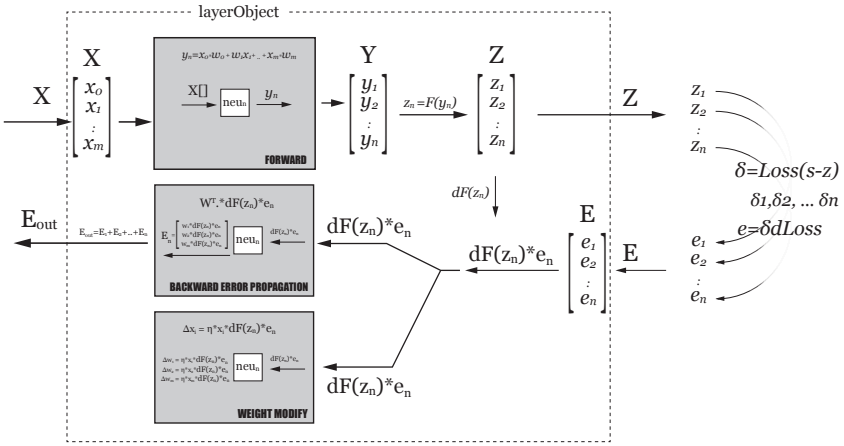
- Warstwa 1: 64 perceptronów, wejścia 784, aktywacja - funkcja sigmoidalna
- Warstwa 2: 64 perceptronów, wejścia 64, aktywacja - funkcja sigmoidalna
- Warstwa 3: 10 perceptronów, wejścia 64, aktywacja - funkcja softmax

W procesie wykorzystano zestaw obrazów treningowych i testowych ze zbiorów MNIST [8]

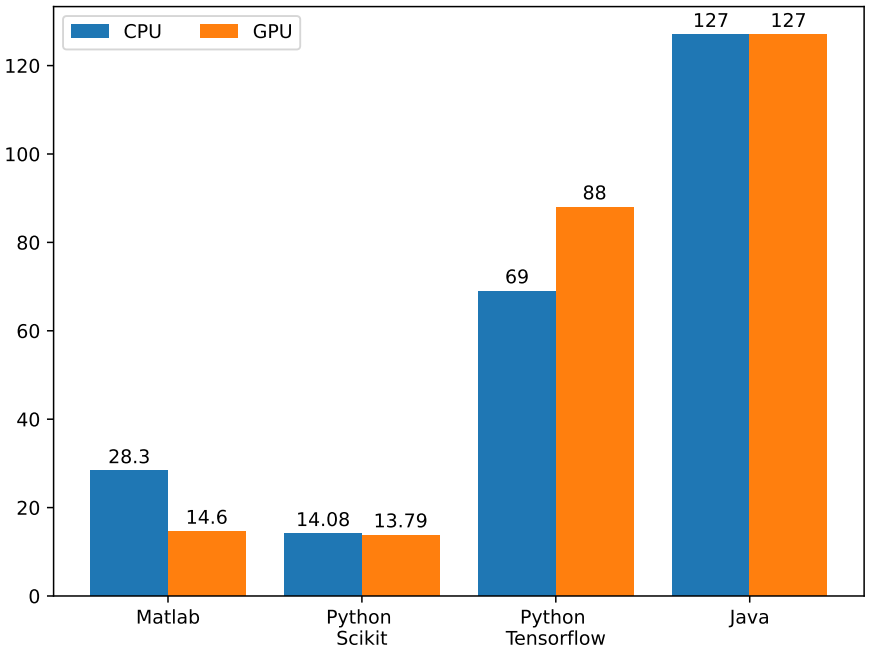
W Matlabie wykorzystano dostępne dodatki, a w Pythonie zainstalowano biblioteki Scikit-learn, zaś w kolejnym rozwiązaniu Tensorflow.

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

5.3 Zadanie 3 - sieć splotowa

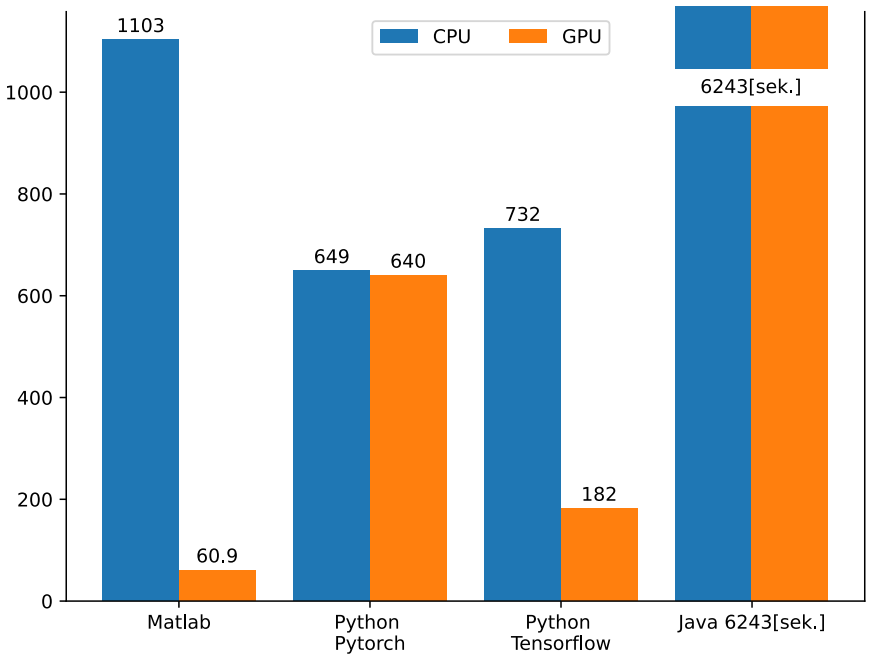
Zadanie polegało na zbudowaniu modelu sieci splotowej CNN połączonej z siecią MLP, wg. struktury LaNet-5 zaproponowanej przez prof. Yann LeCun [24] przedstawionej m.in. w [21].

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

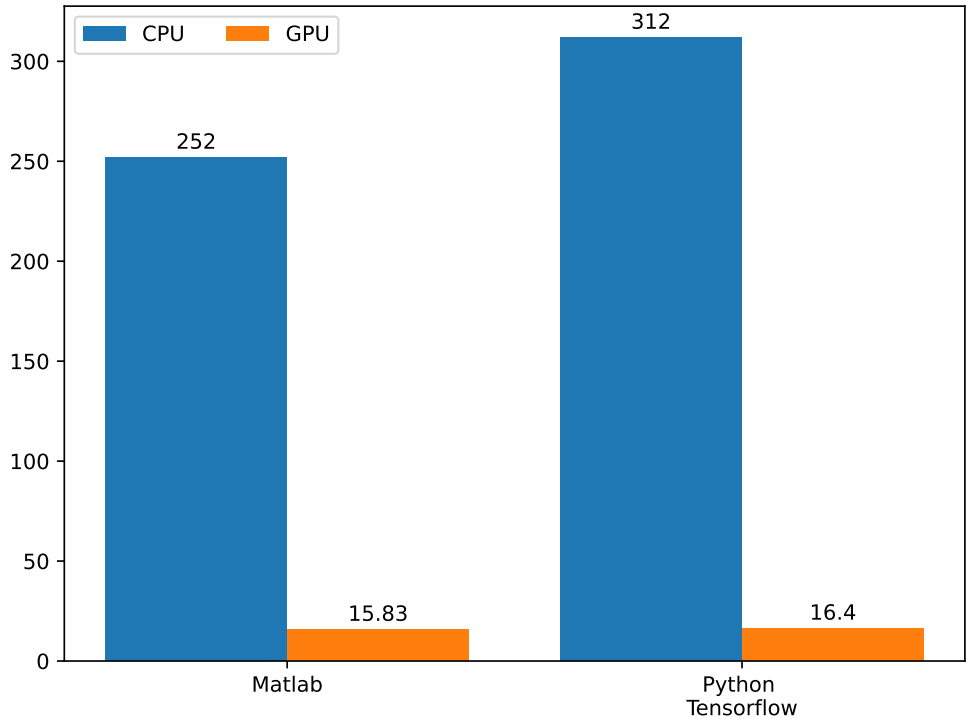
W zadaniu wykorzystano dane z zadania 2, Dwa rozwiązania zostały napisane w Pythonie z wykorzystaniem bibliotek PyTorch oraz Tensorflow. Jedno rozwiązanie powstało w Matlabie z wykorzystaniem bibliotek systemowych. W języku Java napisano własne rozwiązanie, w którym przyjęto te same założenia i strukturę 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 18. 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.



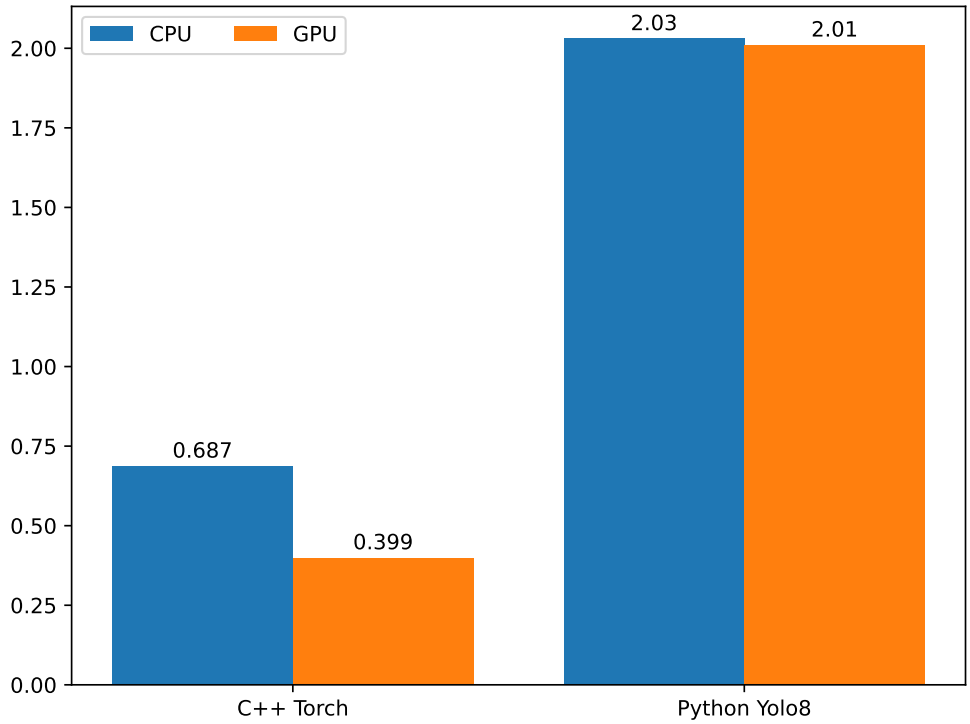
Rysunek 19. Zadanie 4 - zestawienie czasów uczenia sieci głębokiej CNN

```
>> matlab
Initializing input data normalization.
|=========================================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Mini-batch  |  Base Learning  |
|         |             |   (hh:mm:ss)   |   Accuracy   |     Loss     |      Rate       |
|=========================================================================================|
|       1 |           1 |       00:00:01 |        8.89% |       2.3034 |          0.0010 |
|      50 |          50 |       00:00:06 |       95.56% |       0.1285 |          0.0010 |
|     100 |         100 |       00:00:12 |       95.56% |       0.2177 |          0.0010 |
|=========================================================================================|
```

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 20. 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.

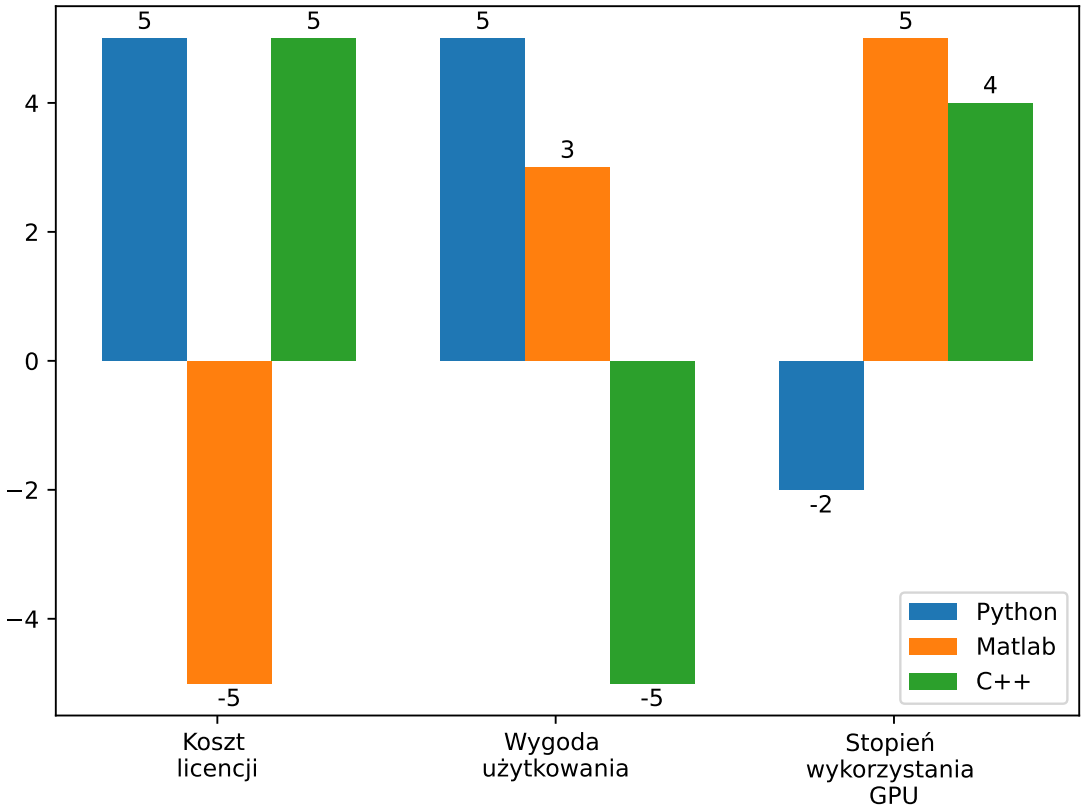
6.1.5 Koszt licencji

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 21. Zestawienie aspektów języków

Według bulldogjob.pl najpopularniejszym językiem programowania jest Python [16]. 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. Należą do nich m.in. Tensorflow, Scikit-learn, PyTorch i wiele innych. Istnieje wiele rozwiązań pomagających zainstalować środowisko, a także budować wiele wirtualnych środowisk na jednej maszynie. Dla poznawania, trenowania modeli AI nadaje się idealnie.

Niestety instalacja systemu z obsługą najnowocześniejszych kart graficznych może być kłopotliwa, a stopień wykorzystania kart graficznych może być różny. Nie jest to także najszybszy z języków [6].

6.3 Matlab

Środowisko Matlab nie jest ujęte w zestawieniu bulldogjob.pl [16]. 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.

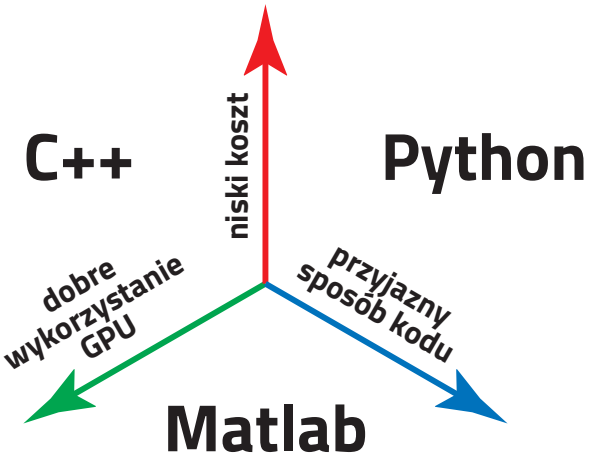
6.4 C++

Język C/C++ wg. bulldogjob.pl osiąga popularność na poziomie 6.6% [16]. 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% [16]. Java jest językiem podobnym do C++, jest to język kompilowany do kodu bajtowego uruchamianego w maszynie wirtualnej. 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 22. 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.

Bibliografia

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

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

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

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

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

[6] bulldogjob.pl. adr.: https://bulldogjob.pl/readme/ktory-jezyk-programowania-jest-najszybszy.

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

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

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

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

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

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

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

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

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

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

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

[18] 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.

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

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

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

[22] 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.

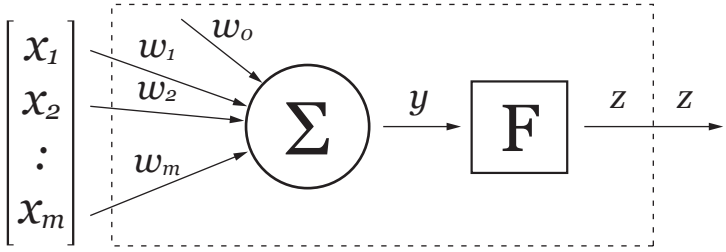
[23] Vincent Dumoulin, F. i †, F. V., *A guide to convolution arithmetic for deep learning*, 2018. adr.: https://arxiv.org/pdf/1603.07285.

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

Spis załączników

6.7 Model klasyczny - podejście matematyczne

Model neuronu [12], rys. 1 można rozpatrywać jako przetwornik sygnałów ciągłych, bardziej zbliżonych do logiki rozmytej (fuzzy logic) niż boolowskiej. Sygnały wejściowe i wyjściowe przyjmują wartości z określonych zakresów, co implikuje działania na wartościach zmiennoprzecinkowych kodowanych najczęściej w standardzie IEEE 754 i odpowiadających im typom *float* lub *double*. Na wejście neuronu podawana jest pewna liczba m sygnałów wejściowych $\mathrm{x_1}$ ... $\mathrm{x_m}$, natomiast na wyjściu pojawia się tylko jeden sygnał wyjściowy z. Odpowiedź neuronu to jedna wartość, a odpowiedź warstwy to zbiór wartości (wektor). Odpowiedź warstwy typu „softmax" jest wprost wektorem rozkładów prawdopodobieństwa przynależności do klas.



Rysunek 23. Model sztucznego neuronu [12]

Samo przetwarzanie polega na wyznaczeniu sumy ważonej sygnałów wejściowych, a następnie na wyliczeniu wartości funkcji aktywacji:

$$\mathrm{y} = \sum_{\mathrm{i}=1}^{\mathrm{m}} \mathrm{x_i} * \mathrm{w_i} - \theta, \mathrm{z} = \mathrm{F(y)} \tag{5}$$

gdzie: $\mathrm{x_i}$ - i-ty sygnał wejściowy, $\mathrm{w_i}$ - i-ta waga w neuronie, θ - energia aktywacji, F - funkcja aktywacji, z - wielkość sygnały wyjściowego neuronu.

Przy podstawieniu: $z_0 = 1$ oraz $w_0 = -\theta$ wzór przyjmuje wygodniejszą postać:

$$y = \sum_{i=0}^{m} x_i * w_i \tag{6}$$

Neurony zorganizowane są w warstwy w taki sposób, by te należące do jednej warstwy miały dostęp do tego samego wektora wejściowego X. Sygnał wyjściowy każdego neuronu to pojedyncza wartość z_j, zaś na sygnał wyjściowy warstwy składają się sygnały wszystkich neuronów należących do tej warstwy i tworzą wektor Z. Liczbę neuronów w warstwie oznaczamy n, zatem liczba wymiarów wektora Z to także n.

$$Z = \begin{bmatrix} z_1, \\ z_2, \\ : \\ z_n \end{bmatrix}$$

W najprostszych układach neurony z jednej warstwy nie komunikują się między sobą.
W zapisie wektorowym mamy:

$$X = \begin{bmatrix} x_0 = 1, \\ x_1 ... \\ : \\ x_m ... \end{bmatrix}$$

$$W = \begin{bmatrix} W_1 & W_2 & W_3 \\ \begin{bmatrix} w_0 \\ w_1 \\ w_m \end{bmatrix} & \begin{bmatrix} w_0 \\ w_1 \\ w_m \end{bmatrix} & \begin{bmatrix} w_0 \\ w_1 \\ w_m \end{bmatrix} \end{bmatrix}$$

$$z_1 = F(W_1 . * X)$$

$$Z = F(W^T * X)$$

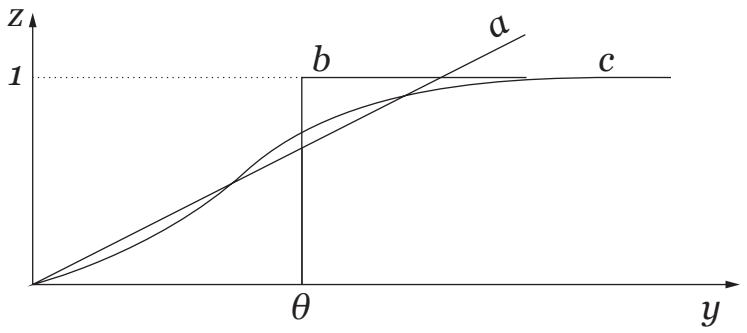
Rozpisujemy mnożenie macierzy przez macierz $W^T * X$ jak poniżej:

$$\begin{bmatrix} w_{10} \ , \ w_{11} \ , \ w_{1m} \\ w_{20} \ , \ w_{21} \ , \ w_{2m} \\ w_{n0} \ , \ w_{n1} \ , \ w_{nm} \end{bmatrix} \quad \begin{bmatrix} x_0 \\ x_1 \\ x_m \end{bmatrix} \quad \begin{bmatrix} x_0 * w_{10} + x_1 * w_{11} + x_m * w_{1m} \\ x_0 * w_{20} + x_1 * w_{21} + x_m * w_{2m} \\ x_0 * w_{n0} + x_1 * w_{n1} + x_m * w_{nm} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_m \end{bmatrix}$$

w którym X jest wektorem wejściowym, x_i jest i-tym sygnałem wejściowym, w_i jest i-tą wagą, w_{ji} jest i-tą wagą j-tego neuronu, W_j oznacza zapis wag jako wektor, zaś W oznacza macierz wag.

Symbol .* oznacza mnożenie Hadamarda czyli mnożenie pierwszego elementu z pierwszym, drugiego z drugim itd. Zapis $\mathrm{X^T}$ oznacza transpozycję macierzy lub wektora. Zapis $\mathrm{W^T} * \mathrm{X}$ oznacza mnożenie macierzowe $\mathrm{X^T}$ przez W. **Aby wyliczyć odpowiedz jednej warstwy musimy wykonać** $\mathrm{m} * \mathrm{n}$ **mnożeń i** $(\mathrm{m}-1) * \mathrm{n}$ **dodawań oraz musimy wyliczyć** n **razy wartości funkcji** $\mathrm{F(y_j)}$**.**

6.7.1 Funkcje aktywacji



Rysunek 24. Funkcje aktywacji

Sygnał y przetwarzany przez blok aktywacji F może być opisany różnymi funkcjami.

- Może być to np. prosta funkcja liniowa (a):

$$\mathrm{z} = \mathrm{ky}, \text{gdzie k jest zadanym stałym współczynnikiem.} \tag{7}$$

- ReLU - funkcja liniowa, która w części dodatniej ma współczynnik k=1, oraz współczynnik k=0 w części ujemnej:

$$\mathrm{z} = \mathrm{F(y)} = \mathrm{y^+} = \max(0, \mathrm{y}) = \mathrm{z}\begin{cases} \mathrm{x} \text{ jeśli } \mathrm{y} > 0, \\ 0 \text{ jeśli } \mathrm{y} \leq 0, \end{cases} \tag{8}$$

- Funkcja skokowa Heaviside'a, skok jednostkowy (b):

$$\mathrm{y} = \mathrm{H(z)} = \begin{cases} 1 \text{ jeśli } \mathrm{y} > \theta, \\ 0 \text{ jeśli } \mathrm{y} \leq \theta, \end{cases} \tag{9}$$

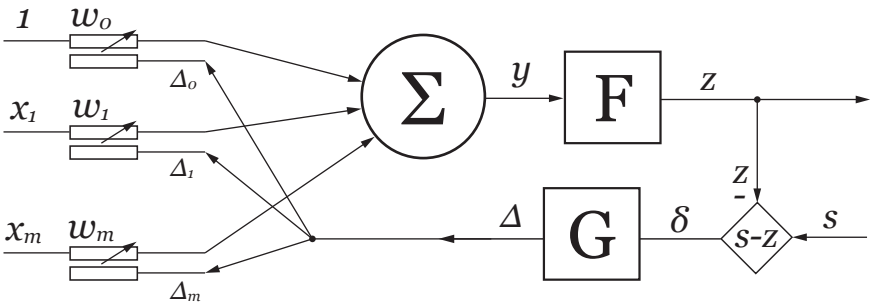
- Funkcja bipolarna:

$$\mathrm{y} = \mathrm{sgn(z)} = \begin{cases} 1 \text{ jeśli } \mathrm{y} > \theta, \\ -1 \text{ jeśli } \mathrm{y} \leq \theta, \end{cases} \tag{10}$$

- Funkcja logistyczna (sigmodalna) (c):

$$\mathrm{z} = \frac{1}{1+\exp(-\beta \mathrm{y})}, \text{ dla } \beta\text{=1 pochodna: } \frac{\partial \mathrm{z}}{\partial \mathrm{y}} = \mathrm{z}(1-\mathrm{z}) \tag{11}$$

6.7.2 Proces uczenia



Rysunek 25. Proces uczenia elementu perceptronowego

Proces uczenia pojedynczego neuronu polega na:

- obliczeniu wartości sygnału wyjściowego y;
- obliczeniu wartości sygnału wyjściowego $\mathrm{z} = \mathrm{F}(\mathrm{y})$;
- porównaniu wektora wyjściowego z z wartością oczekiwaną s oraz obliczenie różnicy $\delta = \mathrm{s} - \mathrm{z}$;
- zależnie od metody uczenia wyznaczenia wartości $\Delta = \mathrm{G}(\delta)$;
- wyznaczenie wielkości poprawek dla poszczególnych wag Δ_i;
- aktualizacja wagi: $\mathrm{w}(\mathrm{k}+1) = \mathrm{w}(\mathrm{k}) + \Delta\mathrm{w}(\mathrm{k})$.

Możemy to opisać wzorem:

$$\mathrm{w}(\mathrm{k}+1) = \mathrm{w}(\mathrm{k}) + \mathrm{G}(\mathrm{s}-\mathrm{z}) \tag{12}$$

gdzie: $\mathrm{w}(\mathrm{k}+1)$ jest wartością uaktualnioną, $\mathrm{w}(\mathrm{k})$ wartością przed aktualizacją. Funkcja $\mathrm{G}(\delta)$ jest funkcją, na podstawie której obliczamy wielkości poprawek.

6.7.3 Algorytm propagacji wstecznej

W algorytmie propagacji wstecznej [12], [22], wysyłamy do warstwy wejściowej próbkę danych X, wyjście warstwy łączy się z wejściem następnej, a wynik przetwarzania warstwy poprzedniej jest wysyłany do warstwy następnej i tak aż do wyjścia. Wynik odczytany z warstwy wyjściowej porównujemy z oczekiwaną wartością wyjściową. Różnica tych wartości stanowi błąd δ. Wartość błędu jest przesyłana w kierunku odwrotnym, czyli od warstwy wyjściowej aż do wejściowej i na podstawie wielkości błędu korygowane są wielkości wag neuronów w kolejnych warstwach.
Aby proces był optymalny, zakładamy pewną funkcję straty „Loss" „L" której parametrami są wagi warstw, wektory wejściowe i powiązane z nimi oczekiwane wartości wyjściowe. Wartość tej funkcji określa wielkość błędu odpowiedzi sieci dla pojedynczego sygnału wejściowego X. Chcemy znaleźć

takie wagi warstw, dla których suma błędów dla wszystkich próbek w serii (zwanej epoką) będzie jak najmniejsza. Zapiszemy

$$\mathrm{Loss(S,Z)} = \mathrm{Loss(S,F(Y))} = \mathrm{Loss(S,F(x_1 * w_1, x_2 * w_2 ... x_m * w_m))}$$

$$\sum_{n=1}^{N} \mathbb{L}(\mathrm{S,F(y)}) = \sum_{n=1}^{N} \mathbb{L}(\mathrm{s1_1, ..., s_m, F(x_1 * w_1, ..., x_m * w_m)}) \tag{13}$$

gdzie $\mathbb{L}$ jest funkcją straty, F jest funkcją aktywacji, a X jest wektorem wejściowym, S oczekiwaną odpowiedzią, zaś n liczbą próbek X.

Z analizy matematycznej wiemy, że w miejscach, w których występuje extremum funkcji, jej pierwsze pochodne się zerują. Dla funkcji wielu zmiennych metoda obliczania punktów zerowania pochodnych jest niepraktyczna, wygodniejsza jest iteracyjna metoda *spadku gradientowego*, w której korygujemy wagi o pewien krok w kierunku największego spadku funkcji $\mathbb{L}$. W tym przypadku korekty wyniosą:

$$\mathrm{w_{ji}} = \mathrm{w_{ji}} + \eta * \frac{\partial}{\partial \mathrm{w_j}} \mathbb{L} \tag{14}$$

Jeśli przyjmiemy miarę $\mathbb{L}$ jako miarę błędu kwadratowego $\mathbb{L} = (\mathrm{s-z})^2 = (\mathrm{s-F(y)})^2$
wówczas:

$$\frac{\partial}{\partial \mathrm{w_j}} \mathbb{L} = \frac{\partial}{\partial \mathrm{F}} \mathbb{L} * \frac{\partial}{\partial \mathrm{y_i}} \mathrm{F} * \frac{\partial}{\partial \mathrm{w_i}} \mathrm{y}$$

po podstawieniu:

$$\frac{\partial}{\partial \mathrm{w_j}} \mathbb{L} = 2(\mathrm{s-F(y)}) * \frac{\partial}{\partial \mathrm{y_i}} \mathrm{F} * \frac{\partial}{\partial \mathrm{w_i}} \mathrm{y}$$

oraz założywszy: $\mathrm{F(y)} = \mathrm{z}$, $\frac{\partial}{\partial \mathrm{y}} \mathrm{F(y)} = 1$ a także: $\mathrm{z} = (\mathrm{w_1 * x_1 + \cdots + w_m * x_m})$ oraz $\frac{\partial}{\partial \mathrm{w_i}} \mathrm{z} = \mathrm{x_i}$

$$\mathrm{w_{ji}} = \mathrm{w_{ji}} + \eta * 2(\mathrm{s-F(y)}) * \frac{\partial}{\partial \mathrm{y}} \mathrm{F}$$

ostatecznie uzyskamy:

$$\mathrm{w_{ji}} = \mathrm{w_{ji}} + \eta(\mathrm{s-z}) * 1 * \mathrm{x_i} \tag{15}$$

gdzie η jest współczynnikiem uczenia, j numerem neuronu, zj odpowiedzią neuronu j, i numerem zmiennej x, zaś m numerem warstwy.

Nazwa	Funkcja $F(y)$	Pochodna d/dyF	Zmiana wagi Δw_i $w_i = w_i + \eta * \Delta * x^T$
dla funkcji straty	$L = \sum((s-z)^2)$	pochodna:	$\frac{\partial}{\partial z}L = 2(s-z)$
Funkcja logistyczna (sigmodalna)	$z = \frac{1}{1+\exp(-y)}$	$z(1-z)$	$\Delta = (s-z) * z(1-z)$ $\Delta = W^{L+1^T}\Delta^{L+1} * z(1-z)$
funkcja liniowa	$z = ky$	k	$(s-z) * 1 * x_i$
dla funkcji straty: entropia krzyżowa Z(z,1-z)	$L(S-Z) =$ $= -\sum_{n=1}^{2} S_i * lnZ_i$ $-(sln.z + \ldots$ $\ldots (1-s)ln(1-z))$	pochodna:	$\frac{\partial}{\partial z}L = \frac{(s-z)}{z(1-z)}$
Funkcja logistyczna (sigmodalna)	$z = \frac{1}{1+\exp(-y)}$	$z(1-z)$	$\Delta = (s-z)$
dla f. SOFTMAX $s_k = \exp(y_k - y_{max})/$ $\sum_{j=1}^{K}\exp(y_j - y_{max})$	wejście S, wyjście Z $S = [s1, s2..]$ $y = [y_1, y_2, \ldots]$	$\frac{\partial}{\partial y_i}s_k = s_k * (\delta_{ki} - s_i)$ $\delta_{ki} = (k == i)?1 : 0$ $L = -lns_l : K_l = [1, 0 \ldots]^T$ $\frac{\partial}{\partial y_i}L = y_i - k_i$	$\Delta = (y_i - k_i)$
ReLU	nieciągła	0,-,1	ustalony krok, $(s-z) * x_i$
Skoku jednostkowego	nieciągła	0,-,0	ustalony krok, $(s-z) * x_i$
Funkcja bipolarna	nieciągła	0,-,0	ustalony krok, $(s-z) * x_i$

Tabela 1. Zestawienie funkcje aktywacji, pochodnych oraz zmian wag w zależności od funkcji straty [13]