

Politechnika Warszawska

W Y D Z I A Ł E L E K T R Y C Z N Y



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 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 realizując operacje 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, Matlab, 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 języku 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, Matlab, Java, C ++. To build the YOLO8 network was used Libraries: Torch, Ultralytics. The CNN network designs were used: tensorflow, scikit-learn, pytorch. 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 perceptronu 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 - regresja liniowa	27
5.2	Zadanie 2 - perceptron warstwowy	29
5.3	Zadanie 3 - sieć splotowa	31

5.4	Zadanie 4 - głęboka sieć spłotowa	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 [11]. Na wybór konkretnego tematu i dobór języków miała wpływ także lektura [20] oraz [14]. [17] [22] [9]

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, wzorując się przede wszystkim na metodach 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-tych 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.[11]

Sztuczne sieci neuronowe zdobyły sobie 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 pod tym względem szczytowe osiągnięcie sztucznej inteligencji.[14]

Za protoplastę tych (głębokich) sieci można uznać zdefiniowany na początku lat '90 wielowarstwowy neocognitron prof. Kunihiko Fukushima. 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)[20]

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 wersja 12.[20]

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 [11] oraz CNN [13] [21] 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 opisano 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. A także zaprezentowano 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: [20] [14] [18] [19]
- instrukcji i przykładów załączonych do bibliotek
- otwartych repozytoriów git [10] [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 tworzenia przetwarzającego informacje całą objętością w sposób jednoczesny używając 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 znormalizowanego punktu przez macierz translacji:

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

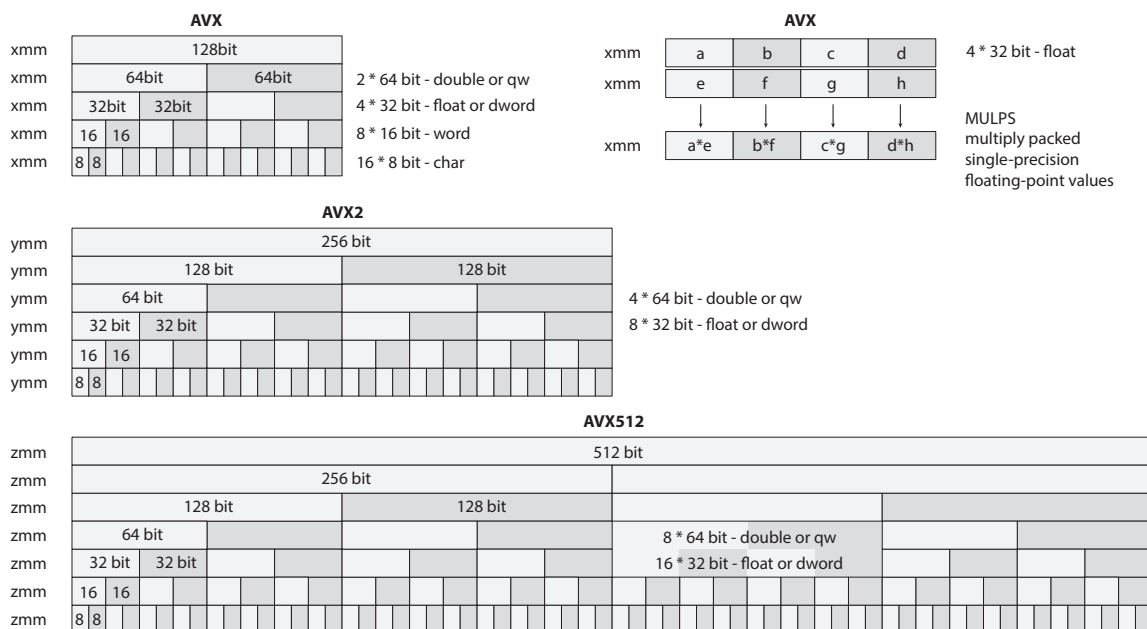
a po rozpisaniu mamy:

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

Pierwszą odpowiedzią na to zapotrzebowanie był wprowadzony na rynek w 1980 roku koprocessor 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).[8] 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.[8]



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

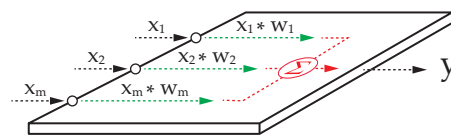
Rozkaz MULPS wykonuje jednocześnie 4 mnożenia zestawu 8 liczb 32 bitowych. Instrukcje w AVX512 VMPADDWD, VMPADDUBSW - wykonują mnożenie wektorowe, następnie dodawanie sąsiednich elementów. $(a_i \cdot b_i + a_{i+1} \cdot b_{i+1})$

Funkcje te przyspieszają operacje iloczynu Hadamarda (\cdot) [12] 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

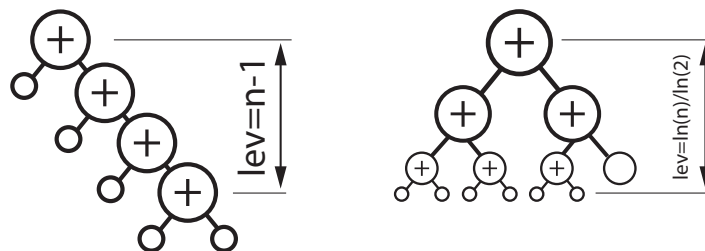
Wydajne obliczanie iloczynu Hadamarda (\cdot) wektorów o dużych rozmiarach mogą zapewnić procesory graficzne, w których znajdują się tysiące jednostek obliczeniowych. Czas wykonania kilku tysięcy mnożeń zajmuje tyle samo czasu co wykonanie jednego mnożenia. Użycie kart graficznych przy modelowaniu głębokich 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 analogicznie do zdegenerowanego drzewa binarnego czyli $((a+b)+c)+d$. Przy dużej ilości składników możemy spróbować zastosować zwykłe drzewo binarne nie zdegenerowane, a operacja dodawania może stać 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

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

Ponad dwukrotne zwiększenie szybkości przez dodanie nawiasów. przy 32 składnikach mamy $16 + 8 + 4 + 2 + 1$ dodawań, jednak pierwsze 16 może być wykonane równoległe, 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, $512+256+128+\dots+1$ w 10 poziomach. [4]

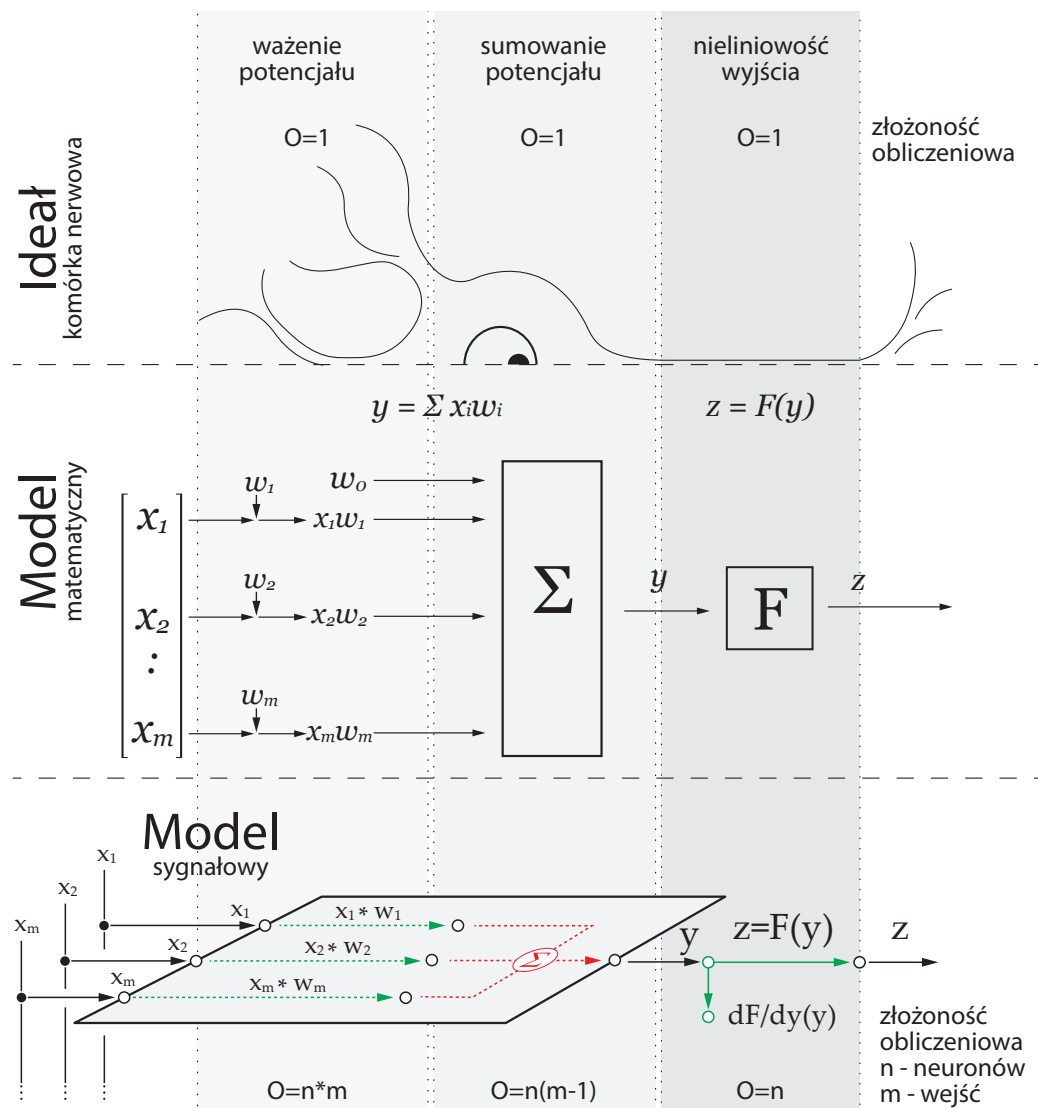
2.4 Matlab

Dodatek Parallel computing w Matlab umożliwia wykonywanie obliczeń równoległe, w osobnych wątkach, procesach a także z wykorzystaniem kart graficznych (obecnie tylko NVidia). Dodatki Optimization Toolbox i Optimization 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 perceptronu warstwowego MLP



Rysunek 4. Neuron, model matematyczny i sygnałowy

Neuron biologiczny przetwarza impulsy wejściowe w 3 krokach.

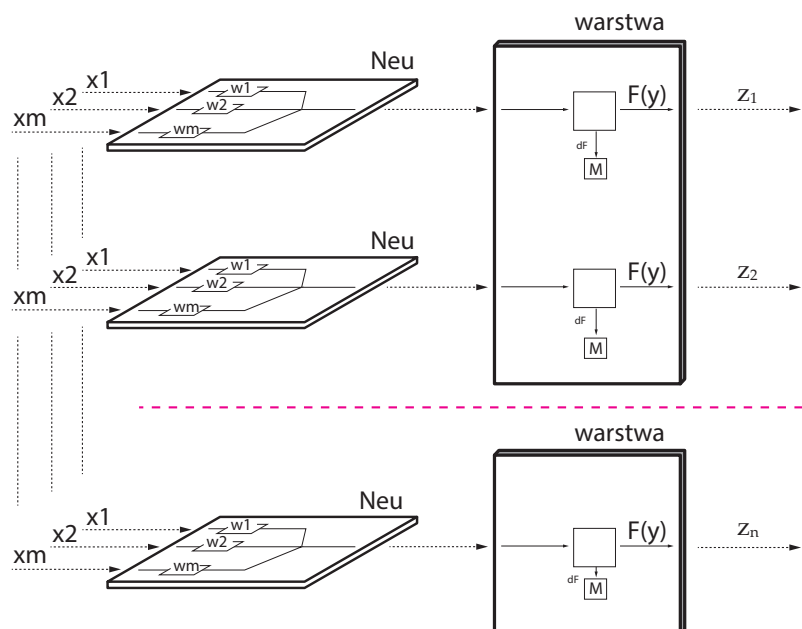
- ważenie wejściowych sygnałów, jest to skomplikowany proces, zmienny w czasie i zależny od historii wcześniejszych impulsów. W modelu matematycznym odpowiada mu mnożenie sygnału wejściowego x_i przez wielkość wagi w_i przypisanej do tego sygnału. $x_i w_i = x_i * w_i$
- jednocześnie sumowanie potencjałów sygnałów wejściowych; W modelu matematycznym odpowiada mu algebraiczna suma ważonych sygnałów: $y = \sum x_i w_i$
- nieliniowa aktywacja. W modelu matematycznym $z = F(y)$

W kodzie dla zwiększenia wydajności zapamiętano wartość pochodnej $\frac{\partial F}{\partial y}$ w punkcie z . Zostanie ona wykorzystana w późniejszych obliczeniach w procesie uczenia.

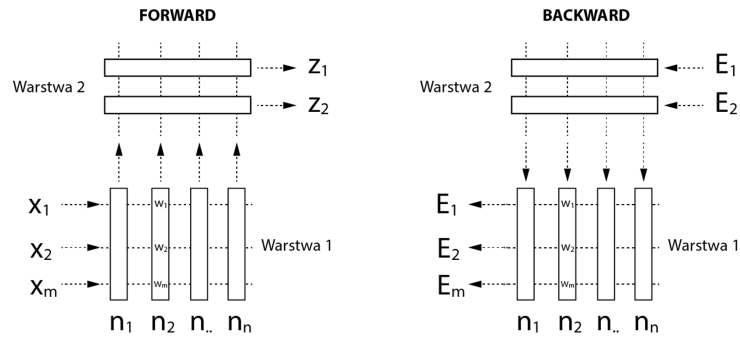
rodzaj warstwy	funkcja aktywacji $z = F(y)$	wartość pochodnej $\frac{\partial F}{\partial y}$
logistyczna (sigmoid)	$\frac{1}{1+e^{-y}}$	$(z)(1-z)$
ReLU	jeśli $y < 0$ to $z=y$, jeśli $y \geq 0$ to $z=0$	jeśli $y < 0$ to 1, jeśli $y \geq 0$ to 0
softmax	$z_i = \frac{e^{x_i}}{\sum e^{x_i}}$	dla prawidłowej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

3.1 Perceptron warstwowy

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

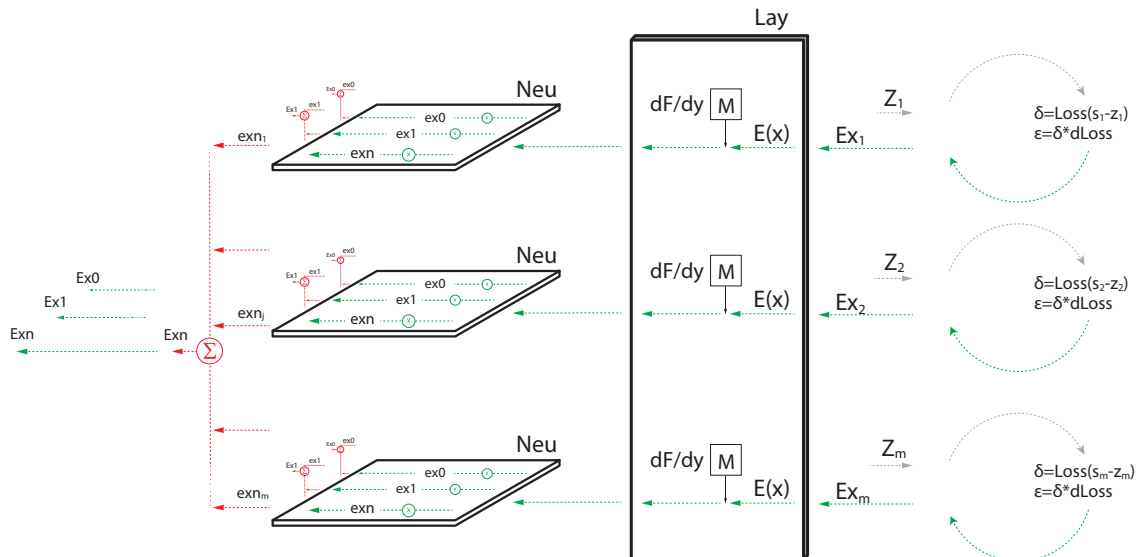


Rysunek 5. Przepływ sygnałów przez warstwę w czasie propagacji sygnału "w przód"



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

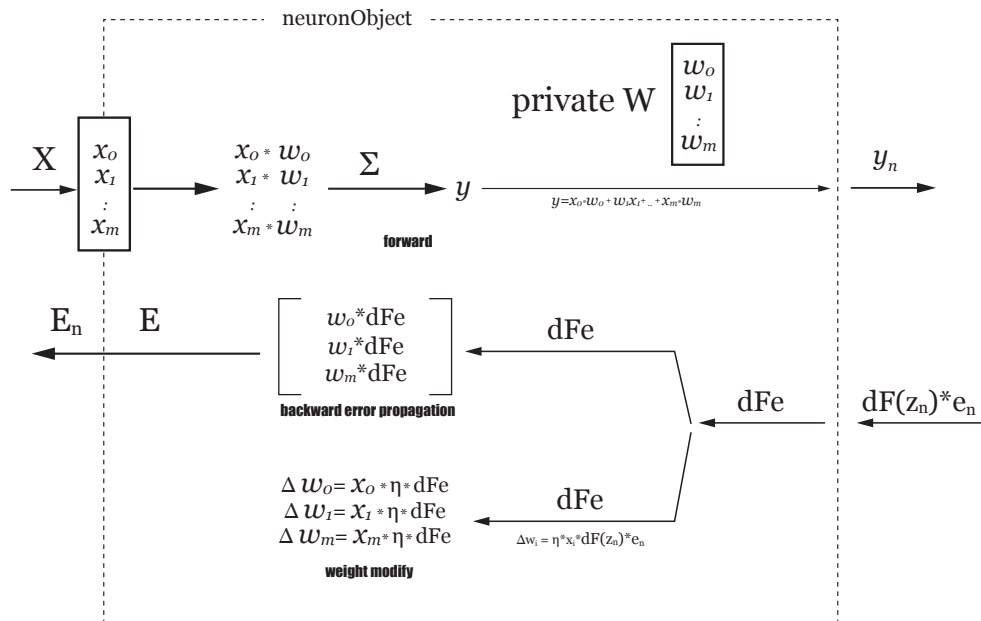
3.2 Proces uczenia MLP



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.

$[e_1 = s_1 - z_1, e_2 = s_2 - z_2, \dots, e_n = s_n - z_n]$, lub w zapisie wektorowym: $E = S - Z$, lub gdzie S jest wielkością oczekiwaną. Dla każdej wartości wyjściowej z_i zostaje zwrócona wielkość błędu e_i , ponadto dla każdej wartości z_i zapamiętano wcześniej wartość pochodnej $\frac{\partial F}{\partial y}$. W procesie uczenia do każdego z neurosumatorów zostaje przekazana odpowiadająca mu wielkość błędu: $dFe = e_i * \frac{\partial F}{\partial y}$.



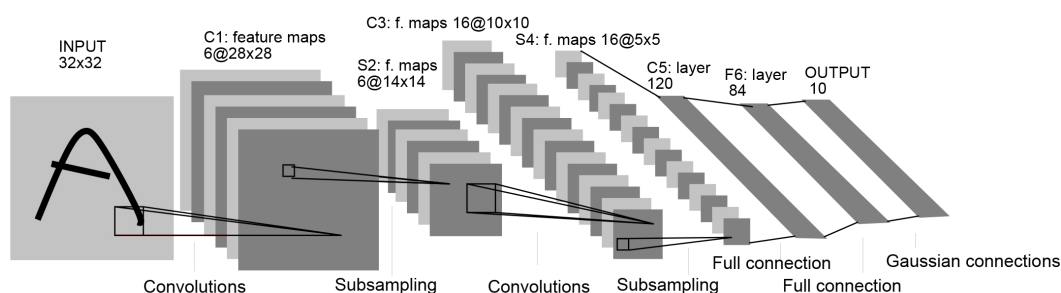
Rysunek 8. Przeptywu sygnałów w obiekcie Neurosumator

Rozdział 4

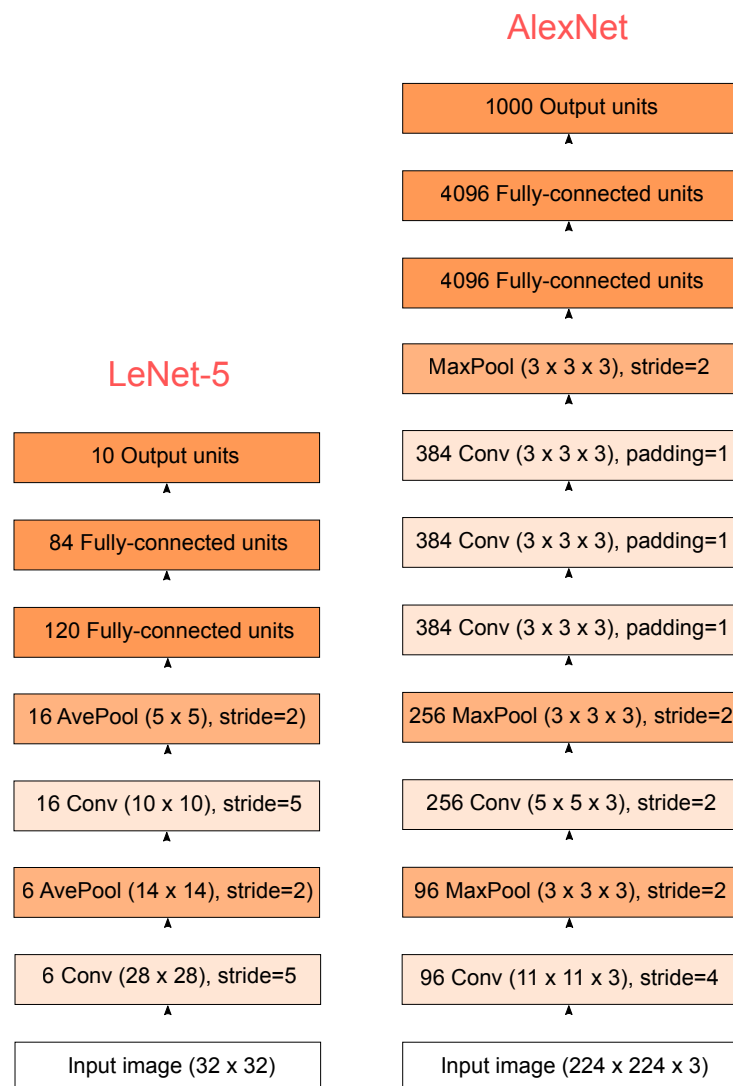
Modele sieci splotowych CNN

4.1 Wstęp

Cytat z [20]. 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. Przetwarzania obrazu przez zestaw filtrów i warstw generuje obrazy o coraz mniejszych wymiarach lecz o zwiększającej się liczbie kanałów, reprezentujące cechy charakterystyczne dla przetwarzanego zbioru. Próbką danych w kolejnych etapach jest reprezentowana przez tensory (struktury 3-wymiarowe, których szerokość i wysokość odpowiada wymiarom obrazu, a głębokość jest równa liczbie kanałów obrazu). Wyjście z ostatniej warstwy sieci w procesie wypłaszczania jest przetwarzane są na postać wektorową (1-wymiarowa tablica liczb), a następnie przekazywane na wejście układu klasyfikującego - sieci MLP z wyjściem Softmax. [23] [16]



Rysunek 9. Architektura LaNet-5 [23]



Notation:

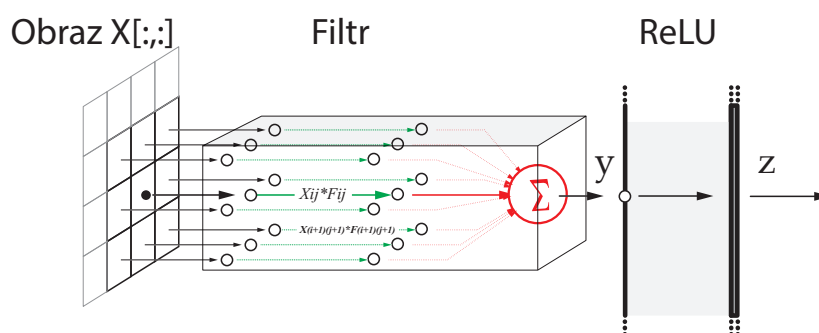
<number of planes> **type of layer** <width x height x RGB>, **stride/padding** <size>

Rysunek 10. Porównanie struktury LaNet i AlexNet [6]

4.2 Operacja splotu

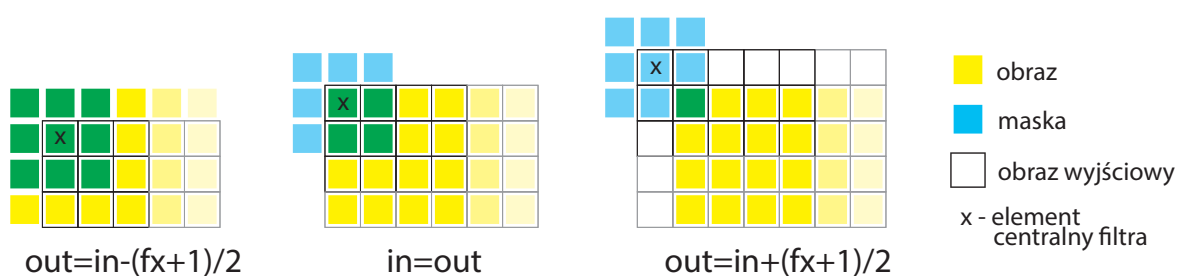
Jeśli wartości pikseli obrazu wejściowego X oznaczmy $X_{(i,j)}$ a wartości filtra F oznaczmy jako $F_{(m,n)}$ obraz wyjściowy Y , $Y_{(o,p)}$ wówczas każdy piksel obrazu wyjściowego obliczymy:

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



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

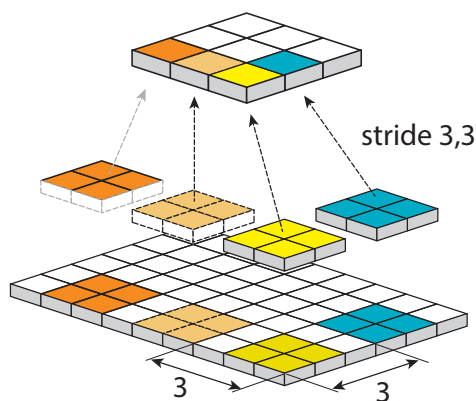
Padding oznacza wysunięcie maski poza obszar obrazu. Jeśli w procesie konwolucji centralny piksel maski:

- znajduje się przed granicą obrazu, wówczas obraz wynikowy będzie mniejszy. ($p=1$)
- 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. ($p=-1$)

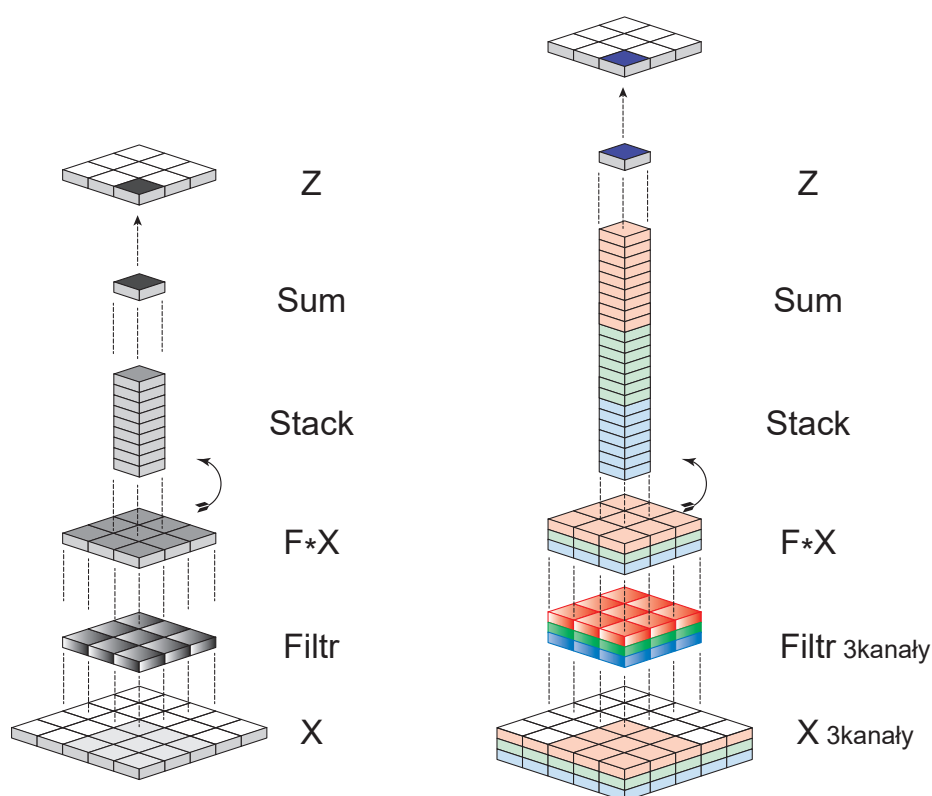
W założeniu procesu uczenia, każdy przesyłany sygnał wejściowy i wyjściowy jest sprzężony z powrotnym sygnałem określającym wielkość błędu tego sygnału. Zatem jeśli w operacji splotu padding jest dodatni, wówczas przy obliczeniu błędu wyjściowego zastosowany będzie padding ujemny.

4.2.2 Stride

Stride oznacza skok o ile pikseli przesuwa się maska w jednym kroku.

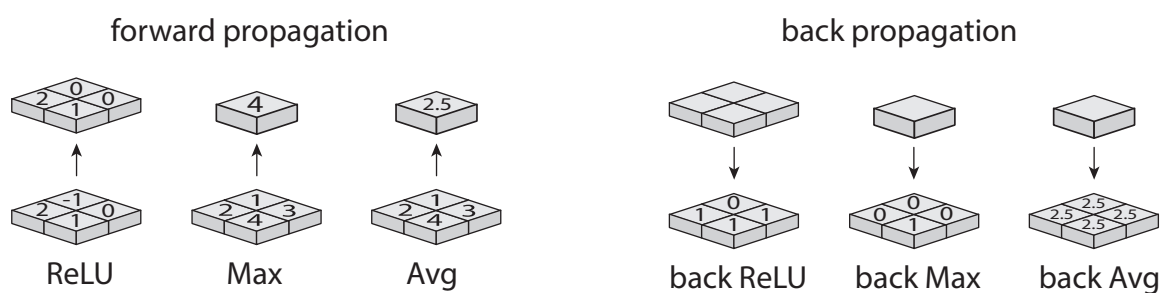


Wymiar wyjściowy obrazu $\text{dimOut} = (\text{dimIn} + 1 - \text{dimF} + 2 * \text{padding}) / \text{stride}$



Rysunek 13. konwolucja jedno i wielokanałowa

4.3 ReLU i warstwy redukujące wymiar



Rysunek 14. wynik działania warstw ReLU, Avg, Max przy przepływie wprost i wstecz.

Po operacji splotu można zastosować warstwę ReLU. Funkcja aktywacji $\text{ReLU} = \max\{x, 0\}$. 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 przypisująca wartość maksymalną z sąsiednich pól. 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, a warstwa redukująca zmniejsza rozmiar obrazu, w efekcie otrzymujemy coraz mniejsze obrazy wyjściowe.

Wyjściem każdego filtra jest jeden kanał obrazu. W miarę zmniejszania obrazów stosowana jest większa liczba 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 spłotowej

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

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

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

4.6 Propagacja błędu przez warstwę spłotową

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

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

Rozdział 5

Zadania

5.1 Zadanie 1 - regresja liniowa

Zadanie zaczerpnięto z [20] i polegało na wyznaczeniu czasu obliczeniu funkcji polyfit:

$$p(x) = p_1x^n + p_2x^{n1} + \dots + p_nx + p_{n+1}$$

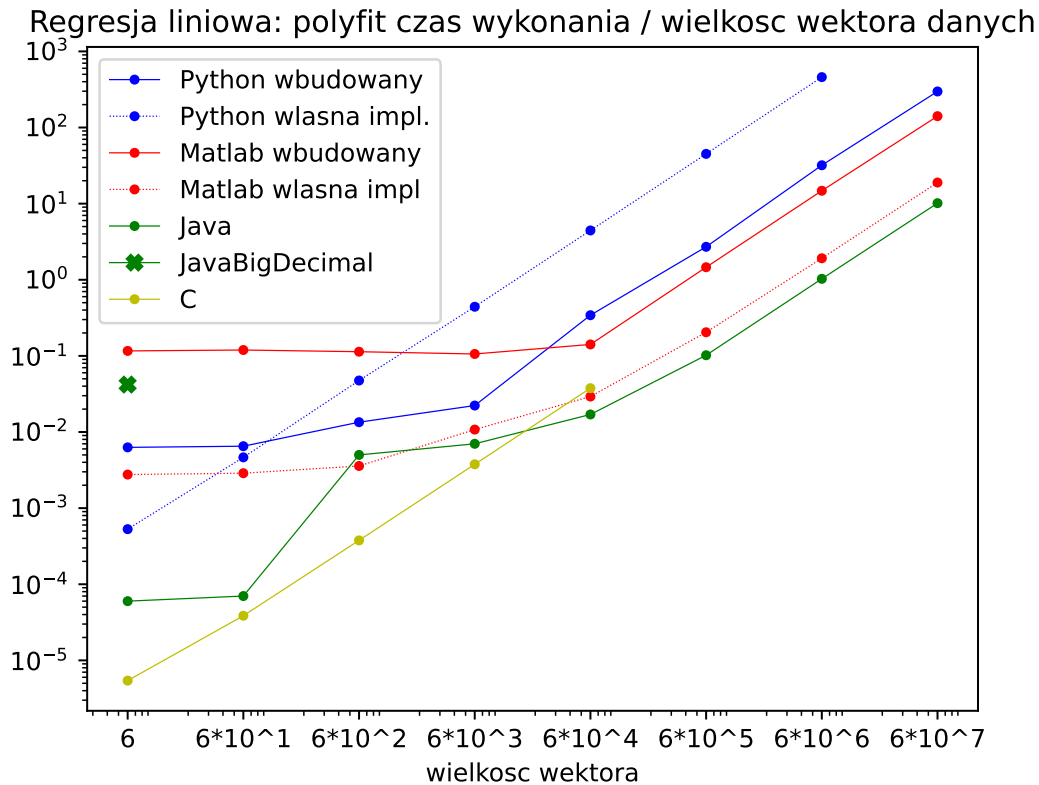
Dane wejściowe wygenerowano losowo i zapisano w pliku. Użyto tych samych danych we wszystkich realizacjach. W językach C++ i Java napisano własny kod:

C++	Python
<pre>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) ; }</pre>	<pre>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; }</pre>

W językach Python i Matlab wykorzystano dostępne funkcje języka:

Matlab	Python
<pre>for i = 1:cycles a = polyfit(x,y,1); end</pre>	<pre>for c in range(cycles): a = np.polyfit(x,y,1)</pre>

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



Rysunek 15. Zadanie 1, zestawienie czasu obliczeń w zależności od liczby próbek w serii, skala logarytmiczna

W kolejnych seriach wielkość zbioru zwiększano o 10. Na wykresie zauważono momenty załamania krzywej (Matlab i Java), po którym następuje zmiana charakterystyki "złożoności obliczeniowej" wykonania programu. Odnotowano, że przy dużych zbiorach danych program w C++ naruszał ochronę pamięci a system kończył jego pracę zgłaszając błąd.

Zestawienie ogólnej wydajności języków w kolejności od najszybszego:

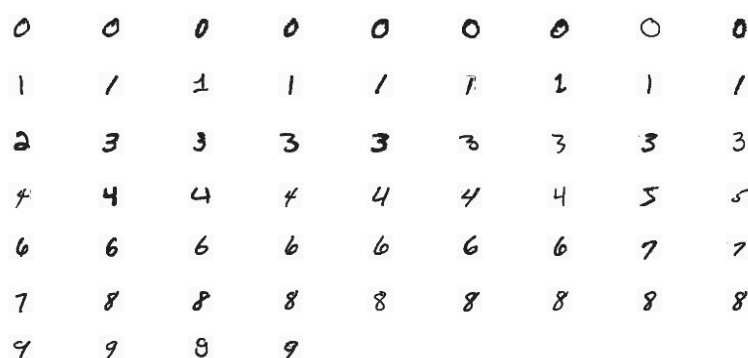
1. C++
2. Java
3. Matlab
4. Python

5.2 Zadanie 2 - perceptron warstwowy

Zadanie również zaczerpnięto z [20], polegało na zbudowaniu modelu sieci MLP. Przeprowadzeniu procesu uczenia sieci, polegającego na rozpoznawaniu cyfr pisanych ręcznie. Założono wykonanie 100 epok uczenia, przy wykorzystaniu 50% objętości zbioru danych uczących.

Zaproponowano sieć MLP o strukturze :

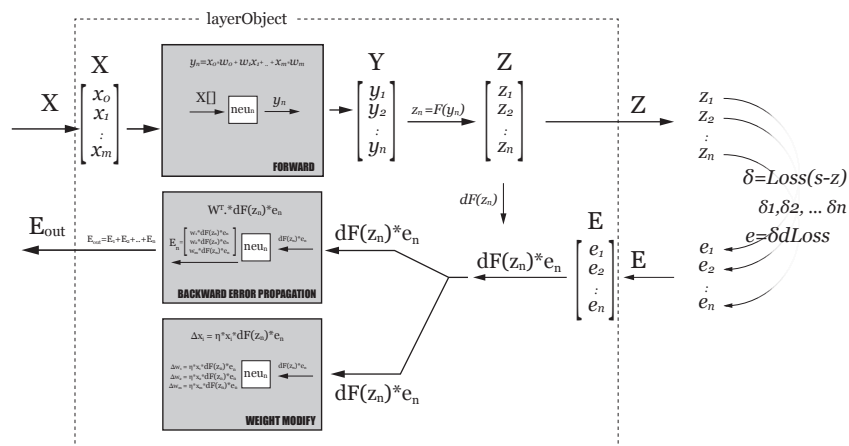
- Warstwa 1: 64 perceptronów, wejścia 784, aktywacja - funkcja sigmoidalna
- Warstwa 2: 64 perceptronów, wejścia 64, aktywacja - funkcja sigmoidalna
- Warstwa 3: 10 perceptronów, wejścia 64, aktywacja - funkcja softmax



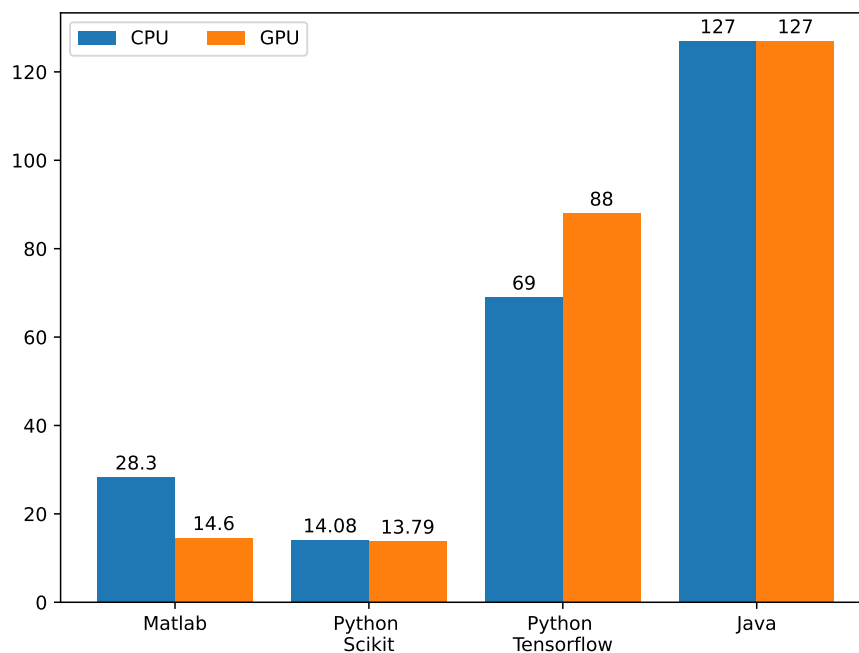
W procesie wykorzystano zestaw obrazów treningowych i testowych ze zbiorów MNIST [7]

W Matlab wykorzystano dostępne funkcje, a w Python zastosowano biblioteki Scikit-learn oraz w kolejnym rozwiązaniu Tensorflow.

Matlab	Python Scikit-learn
<pre> 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');</pre>	<pre> 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)</pre>



Rozwiązanie w Java oparto na własnej implementacji sieci MLP. Szczególny nacisk położono na czytelność kodu i idei, kosztem wydajności. Wykorzystano wiedzę z poprzednich rozdziałów i zaimplementowano zgodnie z zasadami programowania obiektowego, zwłaszcza zasady hermetyzacji. Dlatego spotykana często w literaturze "Warstwa" została rozbita na dwa współpracujące obiekty: Neuron oraz Warstwę będącą kontenerem neuronów. Własnością warstwy jest rodzaj funkcji aktywacji, a metodą obliczenie pochodnej funkcji aktywacji w punkcie. Nie stosowano bibliotek obsługujących GPU w rozwiązaniu Java.



Rysunek 16. Zadanie2, zestawienie czasów uczenia perceptronu MLP

5.3 Zadanie 3 - sieć splotowa

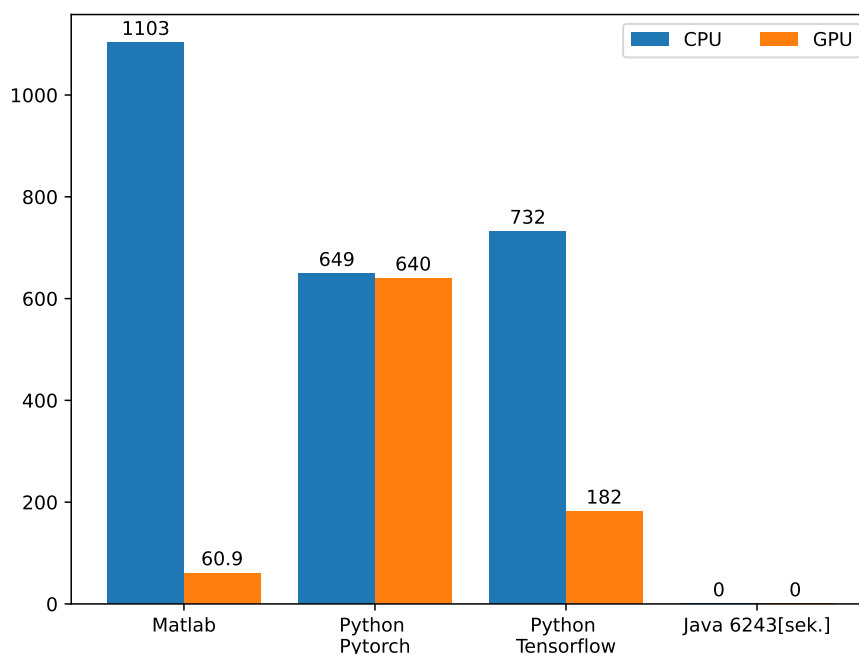
Zadanie polegało na zbudowaniu modelu sieci splotowej CNN połączonej z siecią MLP, wg. struktury LaNet-5 zaproponowanej przez prof. Yann LeCun [23] przedstawionej m.in. w [20].

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

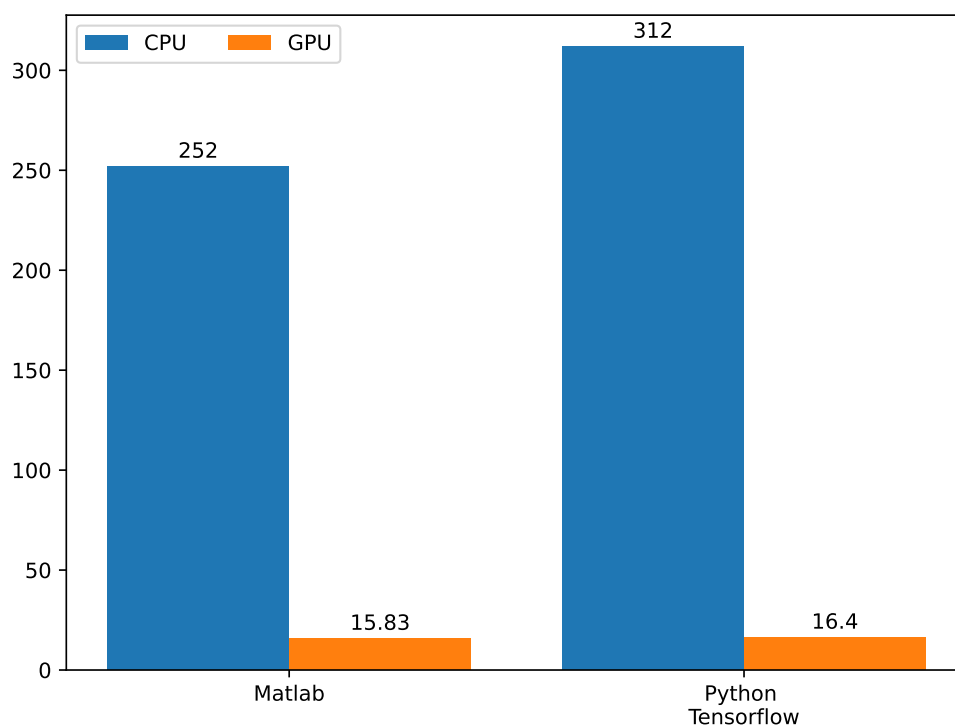
W zadaniu wykorzystano dane z zadania 2, Przygotowano dwa rozwiązania w Python z wykorzystaniem PyTorch oraz Tensorflow. Jedno w Matlab z wykorzystaniem bibliotek. Wykonując implementację własnego rozwiązania w języku Java, przyjęto te same założenia i strukturę jak w zadaniu 2, wprowadzając rozszerzenia dla klas Warstwa i Neuron tak by realizowały sieć splotową CNN oraz działanie warstw pomocniczych, ReLU, Flatten, Max i Avg.



Rysunek 17. Zadanie3, zestawienie czasów uczenia sieci CNN

5.4 Zadanie 4 - głęboka sieć spłotowa

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



Rysunek 18. Zadanie 4, zestawienie czasów uczenia sieci głębokiej CNN

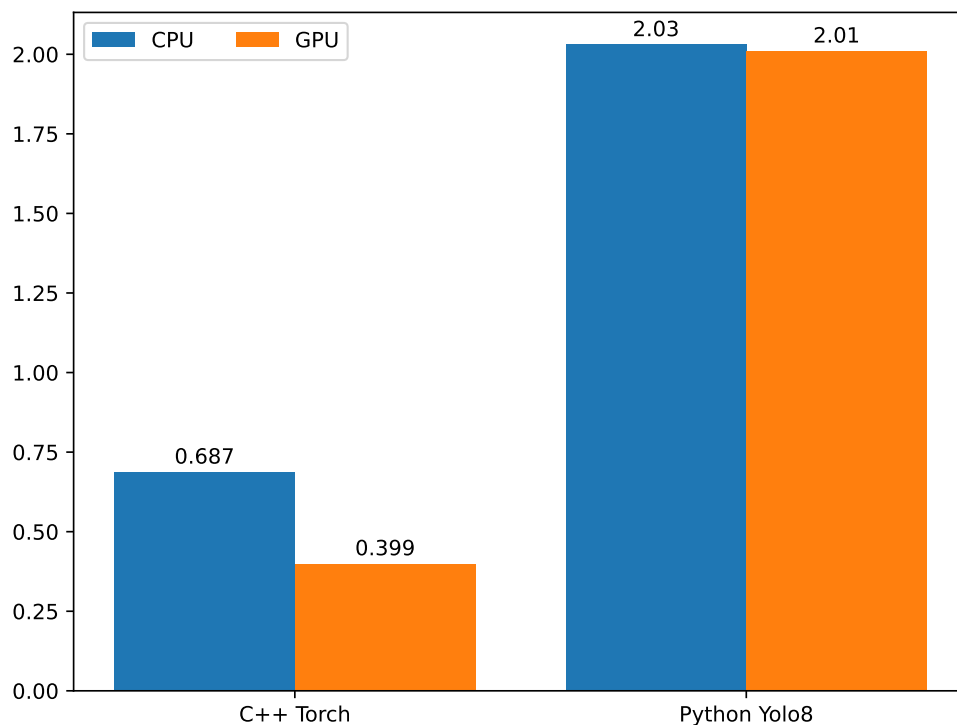
```
>> matlab
Initializing input data normalization.
```

Epoch	Iteration	Time Elapsed (hh:mm:ss)	Mini-batch Accuracy	Mini-batch Loss	Base Learning 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 19. Zadanie 4, zestawienie czasów rozpoznawania obiektów Yolo8

Rozdział 6

Porównanie języków

6.1 Aspekty języków

6.1.1 Popularność języka

Duża popularność rozwiązania zapewnia łatwy dostęp do dużej społeczności osób pracujących z językiem i nad rozwojem języka. Dla dynamicznie rozwijających się języków powstaje wiele bibliotek, środowisk IDE, narzędzi a także literatury i przykładów zastosowania w rozwiązywaniu konkretnych zadań. Przy wyborze języka 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 zależnie 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.

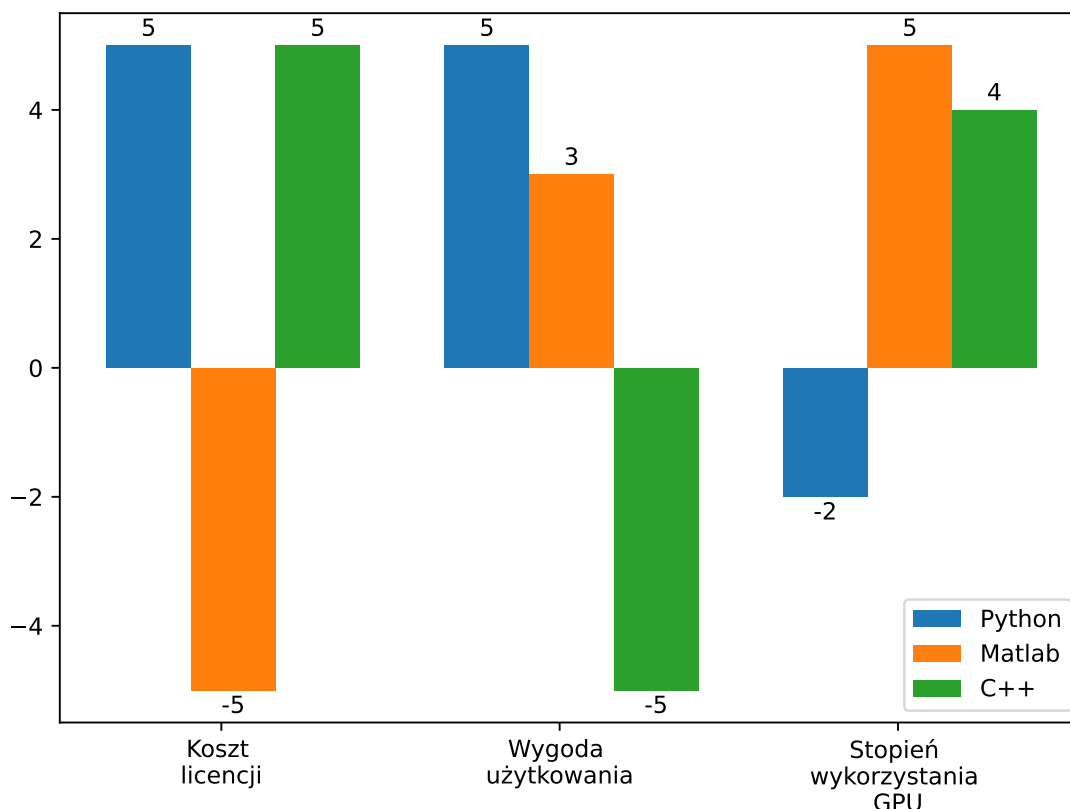
6.1.5 Koszt licencji

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 wymagają od programisty / użytkownika 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++ gdzie występuje ścisła kontrola typów, jednocześnie występują wskaźniki, referencje, ramki stosu. Nie wspominając o języku asemblera gdzie niezbędna jest wiedza nie tylko z zakresu np. reprezentacji liczb w pamięci, przeliczania liczb szesnastkowych na dziesiętne, ale i pracy na rejestrach, stosie, wywołaniach przerwań sprzętowych czy funkcji systemowych.

6.2 Python



Rysunek 20. Zadanie 4, zestawienie czasów rozpoznawania obiektów Yolo8

Według bulldogjob.pl najpopularniejszym językiem programowania jest Python 34.3% [15]. Jest to język młody, jednak pręźnie rozwijany i bardzo modny. Składa się na to kilka przyczyn, po pierwsze jest to język darmowy, (z wyjątkiem niektórych bibliotek). Po drugie ma bardzo niski próg wejścia. Aby pisać kod w Python nie trzeba posiadać specjalistycznej wiedzy technicznej, ani dogłębnego rozumienia 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 bardzo dużo przykładów, samouczków i dokumentacji. Istnieje bardzo 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 najszybszych z języków.

6.3 Matlab

Środowisko Matlab nie jest ujęte w zestawieniu bulldogjob.pl[15]. 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 to rozpoczynanie indeksów od 1.

Dodatek Parallel computing w Matlab umożliwia wykonywanie obliczeń równoległe, w osobnych wątkach, procesach a także z wykorzystaniem kart graficznych (obecnie tylko NVidia). Dodatki Optimization Toolbox i Optimization Computing Toolbox optymalizują tworzony kod zwiększając jego wydajność. Dodatek Deep Learning Toolbox dostarcza gotowych metod do obliczeń sieci neuronowych.

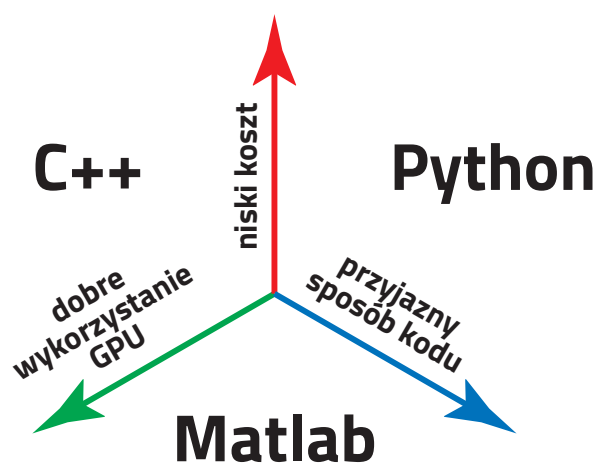
6.4 C++

Język C/C++ wg. bulldogjob.pl osiąga popularność na poziomie 6.6% [15] 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 Python czy Matlab, jednak skompilowana aplikacja działa bardzo szybko. Język dobrze obsługuje karty graficzne. Język ten 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% [15]. 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 ich.

6.6 Podsumowanie i wnioski



Rysunek 21. Priorytety przy wyborze języka

Przy wyborze języka trzeba zdecydować się na dwa z trzech priorytetów, jeden niestety trzeba odrzucić.

Bibliografia

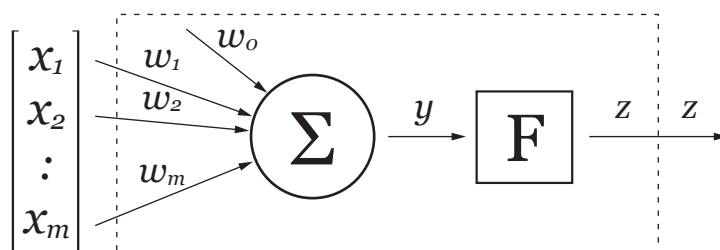
- [1] Adr.: <https://docs.ultralytics.com/models/yolov8/#how-do-i-train-a-yolov8-model>.
- [2] Adr.: <https://github.com/ultralytics/ultralytics/tree/main/examples/YOLOv8-CPP-Inference>.
- [3] Adr.: <https://www.youtube.com/watch?v=x9Scb5Mkulg>.
- [4] Adr.: http://www.cs.put.poznan.pl/rwalkowiak/pliki/2019/pr/programowanie_PKG_CUDA_2019.pdf.
- [5] Adr.: <https://zapisy.ii.uni.wroc.pl/courses/kurs-obliczenia-rownolege-na-kartach-graficznych-cuda-q2-201920-zimowy>.
- [6] Caceres, P. adr.: <https://com-cog-book.github.io/com-cog-book/features/conv-net.html>.
- [7] Cun, Y. A. L. adr.: <http://yann.lecun.com/exdb/mnist/>.
- [8] Hoey, J. V., *Programowanie w asemblerze x64*, Werner, T. G., red. Helion S.A., 2024, ISBN: 978-83-289-0109-4.
- [9] Jefkine, *Backpropagation In Convolutional Neural Networks*, 2016. adr.: <https://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/>.
- [10] Jocher, G., Chaurasia, A. i Qiu, J., *Ultralytics YOLOv8*, ver. 8.0.0, 2023. adr.: <https://github.com/ultralytics/ultralytics>.
- [11] Józef Korbicz Andrzej Obuchowicz, D. U., *Sztuczne sieci neuronowe: podstawy i zastosowania*. Akademicka Oficyna wydawnicza PLJ, 1994, ISBN: 83-7101-197-0.
- [12] Kasprzak, W., *Metody sztucznej inteligencji C6.pdf*, materiał dydaktyczny, 2024.
- [13] Mieczysława Muraszkiewicza i Roberta Nowaka, praca zbiorowa pod redakcją, *Sztuczna inteligencja dla inżynierów*. Oficyna Wydawnicza Politechniki Warszawskiej, 2023, ISBN: 978-83-8156-584-4.
- [14] Osowski, S., *Sieci neuronowe do przetwarzania informacji*. Oficyna Wydawnicza Politechniki Warszawskiej, 2020, ISBN: 978-83-7814-923-1.
- [15] bulldogjob.pl. adr.: <https://bulldogjob.pl/it-report/2025/technologie>.

- [16] Qi, H., „Derivation of Backpropagation in Convolutional Neural Network (CNN)”, 2016. adr.: <https://api.semanticscholar.org/CorpusID:37819922>.
- [17] 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.: <https://hal.science/hal-04409232v1>.
- [18] Sarah Guido, A. M., *Machine learning, Python i data science*. Helion S.A., 2021, ISBN: 978-83-8322-751-1.
- [19] Sebastian Raschka, V. M., *Machine learning, Python i data science*. Helion S.A., 2021, ISBN: 978-83-283-7001-2.
- [20] Stanisław Osowski, R. S., *Matematyczne modele uczenia maszynowego w językach MATLAB i PYTHON*. Oficyna Wydawnicza Politechniki Warszawskiej, 2023, ISBN: 978-83-8156-597-4.
- [21] Stuart Russell, P. N., *Artificial Intelligence: A Modern Approach, 4th Edition*, Grażyński, T. A., red. Pearson Education, Inc., Polish language by Helion S.A. 2023, 2023, ISBN: 978-83-283-7773-8.
- [22] Vincent Dumoulin, F. i ħ, F. V., *A guide to convolution arithmetic for deep learning*, 2018. adr.: <https://arxiv.org/pdf/1603.07285>.
- [23] 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 [11], 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 $x_1 \dots x_m$, 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 22. Model sztucznego neuronu [11]

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

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

gdzie: x_i - i -ty sygnał wejściowy, 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:

$$y = \sum_{i=0}^m x_i * w_i \quad (6)$$

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 .

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

W najprostszych układach neurony z jednej warstwy nie komunikują się między sobą.

W zapisie wektorowym mamy:

$$X = \begin{bmatrix} x_0 = 1, \\ x_1 \dots \\ \vdots \\ x_m \dots \end{bmatrix}$$

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

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

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

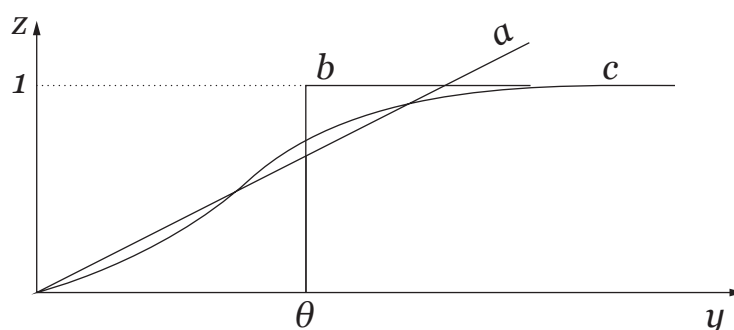
Rozpisujemy mnożenie macierzy przez macierz $W^T * X$ jak poniżej:

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

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

6.7.1 Funkcje aktywacji



Rysunek 23. Funkcje aktywacji

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

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

$$z = ky, \text{ gdzie } k \text{ jest zadany stałym współczynnikiem.} \quad (7)$$

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

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

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

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

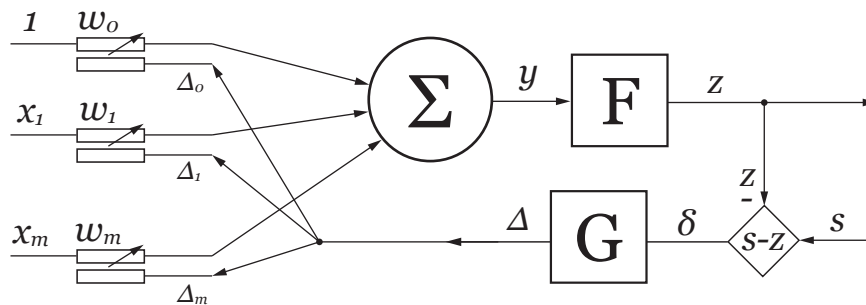
- Funkcja bipolarna:

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

- Funkcja logistyczna (sigmodalna) (c):

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

6.7.2 Proces uczenia



Rysunek 24. Proces uczenia elementu perceptronowego

Proces uczenia pojedynczego neuronu polega na:

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

Możemy to opisać wzorem:

$$w(k+1) = w(k) + G(s - z) \quad (12)$$

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

6.7.3 Algorytm propagacji wstecznej

W algorytmie propagacji wstecznej [11], [21], 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

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

$$\sum_{n=1}^N \mathbb{L}(S, F(y)) = \sum_{n=1}^N \mathbb{L}(s_{1_1}, \dots, s_{1_m}, F(x_1 * w_1, \dots, x_m * w_m)) \quad (13)$$

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 ekstremum 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:

$$w_{ji} = w_{ji} + \eta * \frac{\partial}{\partial w_j} \mathbb{L} \quad (14)$$

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

wówczas:

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

po podstawieniu:

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

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

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

ostatecznie uzyskamy:

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

gdzie η jest współczynnikiem uczenia, j numerem neuronu, z_j 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)	$z = \frac{1}{1+\exp(-y)}$	$z(1 - z)$	$\Delta = (s - z) * z(1 - z)$ $\Delta = W^{L+1T} \Delta^{L+1} * z(1 - z)$
funkcja liniowa	$z = ky$	k	$(s - z) * 1 * x_i$
dla funkcji straty: entropia krzyżowa $Z(z, 1-z)$	$L(S - Z) =$ $= -\sum_{n=1}^2 S_i * \ln Z_i$ $-(s \ln z + \dots$ $\dots (1 - s) \ln(1 - z))$	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 $S = [s_1, s_2, \dots]$ $y = [y_1, y_2, \dots]$	$\frac{\partial}{\partial y_i} s_k = s_k * (\delta_{ki} - s_i)$ $\delta_{ki} = (k == i) ? 1 : 0$ $L = -\ln s_l : K_l = [1, 0, \dots]^T$ $\frac{\partial}{\partial y_i} L = y_i - k_i$	$\Delta = (y_i - k_i)$
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 [12]