

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

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

Piotr Heinzelman

numer albumu 146703

promotor

dr inż. Witold Czajewski

WARSZAWA 2024

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

Streszczenie

Idea jest taka, by sprawdzić jak w różnych językach programowania wygląda jakieś kompleksowe zastosowanie sieci neuronowych do przetwarzania obrazów. Dwa najpopularniejsze języki to C++ i Python, potem jest Matlab. Dość powszechne jest też uruchamianie kodu Pythona w Colabie. Chodzi o zrobienie dokładnie kilku takich samych aplikacji w kilku językach i porównanie ich pod różnymi kątami. Przykładowe aplikacje to: klasyfikacja obrazów, rozpoznawanie obiektów na obrazach, śledzenie obiektów, segmentacja obiektów, modyfikacja obrazów czy ich fragmentów.

Jeśli taki zakres projektu Panom odpowiada, to proszę dać znać, a założę dedykowany projektowi kanał na Teamsach, gdzie będziemy się dalej komunikować.

Słowa kluczowe: A, B, C

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

Abstract

This is abstract. This one is a little too short as it should occupy the whole page.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetur adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Keywords: X, Y, Z

Spis treści

1	Wstęp	9
2	Teoria	13
2.1	Model klasyczny - podejście matematyczne	13
2.1.1	Funkcje aktywacji	15
2.1.2	Proces uczenia	16
2.1.3	Algorytm propagacji wstecznej	16
3	Model obiektowy	19
3.1	Obiekt klasy Layer	19
3.2	Propagacja sygnału pomiędzy obiektami	21
4	Przetwarzanie numeryczne	23
4.1	Kodowanie	23
4.2	operacje równoległe	24
4.3	Dodawanie równoległe	25
4.4	Wątki	25
4.5	HyperWątki HT	26
4.6	Pamięć operacyjna	26
4.7	Rdzenie CUDA	26
4.8	Matlab	27
4.9	Python	27
5	Praktyka	29
5.1	Metodyka	29
5.2	Realizacja obliczeń - Jednowymiarowa regresja liniowa	30
6	Sieci głębokie	33
6.1	CNN	33
6.2	Python TensorFlow	33
7	Podsumowanie	35

Bibliografia	37
Spis rysunków	39
Spis tabel	41
Spis załączników	43

Rozdział 1

Wstęp

Dobór algorytmu do zadania jest bardzo ważny, zdecydowanie ważniejszy niż dobór języka - jednak nie będzie on głównym tematem tej pracy. Tu zakładamy użycie tych samych algorytmów i porównujemy wydajności implementacji algorytmu w różnych "językach". Teoretycznie wyniki powinny być zbliżone przy założeniu, że programy doskonale wykorzystują możliwości sprzętu. Celem pracy jest porównanie, które języki ogólnego przeznaczenia liczą szybciej, które lepiej wykorzystują dodatkową infrastrukturę taką jak *wątki*, *procesy*, *hyperThreading*, czy rdzenie *CUDA* na kartach graficznych. Jeśli gdzieś zaobserwujemy różnice - to będą one wynikały zastosowania innych języków, które w odmienny sposób zapisują i odczytują liczby, kolejkują zadania czy optymalizują wygenerowany kod. Jednak zanim przejdziemy do porównania, przyjrzymy się modelom matematycznym, a z ich pomocą zbudujemy prosty model obiektowy. Następnie przetestujemy model obiektowy, prześledzimy przetwarzanie na najniższych poziomach, aż do pojedynczych operacji. Te działania pomogą nam przygotować zadania numeryczne, do rozwiązania których użyjemy maszyn cyfrowych.

```
1  /*
2
3  4. Realizacje oblicze Jednowymiarowej regresji liniowej
4  a) Matlab - wbudowana funkcja
5  b) Matlab - w Ćasna implementacja
6  c) Python - wbudowana funkcja
7  d) Python - w Ćasna implementacja
8  e) Java - w Ćasna implementacja
9  f) C++ - w Ćasna implementacja
10
11 Cel podstawowy: por wnanie wydajno Źci podstawowych operacji
    arytmetycznych w zale no Źci od J Źzyka.
12 Cele dodatkowe: potwierdzenie poprawno Źci w Ćasnej implementacji przez
    por wnanie wynik w ,
13 Sprawdzenie poprawno Źci funkcji wczytuj cej dane z plik w ,
14
15 oraz opcjonalnie je Źli wystarczy czasu:
```

- 16 g) Python tensorflow.keras lub PyTorch, lub CuPy //https://www.
geeksforgeeks.org/tools-and-libraries-to-leverage-gpu-computing-in-
python/
17 h) C++ na GPU Cuda
18
19 Cel podstawowy: porównanie realnego wzrostu wydajności oblicze na
GPU.
20
21 4. Realizacje oblicze Klasyfikowanie pisma odręcznego MLP z
wykorzystaniem danych MNIST
22 a) Matlab
23 b) Python -numpy, -sklearn
24 c) Python -tensorflow
25 d) w C++ implementacja w Java
26
27 Cel podstawowy: porównanie wydajności realizacji w zależności od
Języka.
28 Cele dodatkowe: potwierdzenie poprawności w C++ implementacji przez
porównanie wyników,
29
30
31 5) realizacja Klasyfikowanie pisma odręcznego sieci głębokie CNN
realizacja sieci z wykorzystaniem bibliotek.
32 a) Matlab
33 b) Python tensorflow.keras
34 c) w C++ implementacja Java lub C++ libtorch
35
36 Cel podstawowy: porównanie wydajności implementacji. Zbadanie
wydajności uczenia sieci.
37 Cele dodatkowe: potwierdzenie poprawności w C++ implementacji w Java
lub C++
38
39
40 6) Rozpoznawanie twarzy sieci głębokie CNN
41 a) Matlab
42 b) Python tensorflow.keras
43 c) w C++ implementacja Java lub C++ libtorch
44
45 Cel podstawowy: porównanie wydajności implementacji.
46
47
48 7) Klasyfikacji obraz w z wykorzystaniem sieci głębokiej CNN
49 a) Matlab
50 b) Python tensorflow.keras
51 c) C++ libtorch
52

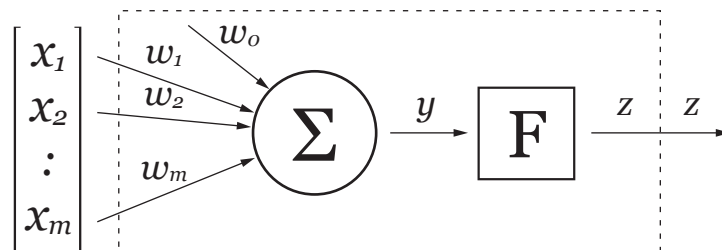
```
53 Cel podstawowy: porównanie wydajności implementacji i wydajności
    procesu uczenia nadzorowanego.
54
55 8) detekcja i segmentacji obiekt w sieci głębszej CNN
56 b) Python tensorflow.keras
57 c) C++ libtorch
58
59 Cel podstawowy: porównanie wydajności implementacji.
60
61
62 */
```

Rozdział 2

Teoria

2.1 Model klasyczny - podejście matematyczne

Model neuronu [1], 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 1. Model sztucznego neuronu [1]

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

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

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:

$$y = \sum_{i=0}^m x_i * w_i \quad (2)$$

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:

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

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

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

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

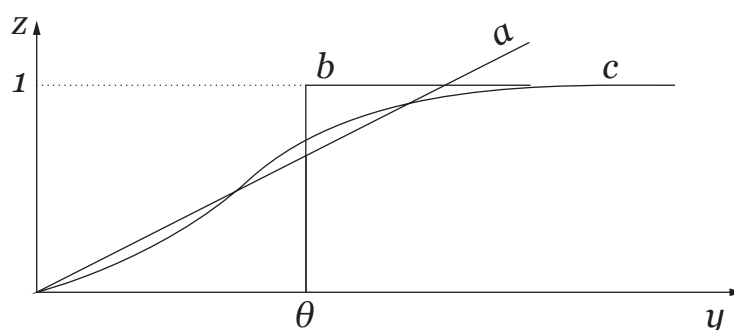
Rozpisujemy mnożenie macierzy przez macierz $W^T * X$ jak poniżej:

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

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 $W^T * X$ oznacza mnożenie macierzowe X^T przez W . **Aby wyliczyć odpowiedź jednej warstwy musimy wykonać $m * n$ mnożeń i $(m - 1) * n$ dodawań oraz musimy wyliczyć n razy wartości funkcji $F(y_j)$.**

2.1.1 Funkcje aktywacji



Rysunek 2. Funkcje aktywacji

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

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

$$z = ky, \text{ gdzie } k \text{ jest zadany stałym współczynnikiem.} \quad (3)$$

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

$$z = F(y) = y^+ = \max(0, y) = \begin{cases} y & \text{jeśli } y > 0, \\ 0 & \text{jeśli } y \leq 0, \end{cases} \quad (4)$$

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

$$y = H(z) = \mathbb{1}(z) = \begin{cases} 1 & \text{jeśli } y > \theta, \\ 0 & \text{jeśli } y \leq \theta, \end{cases} \quad (5)$$

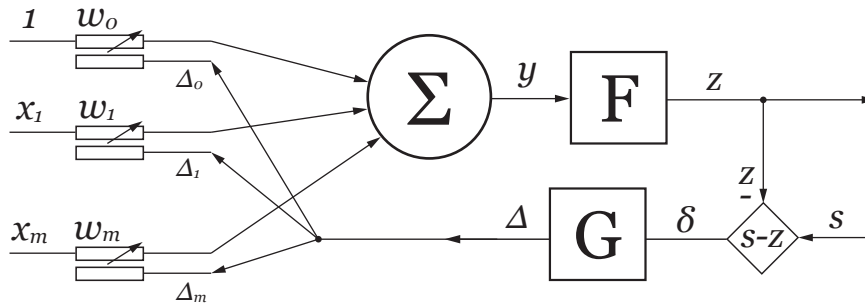
- Funkcja bipolarna:

$$y = \text{sgn}(z) = \begin{cases} 1 & \text{jeśli } y > \theta, \\ -1 & \text{jeśli } y \leq \theta, \end{cases} \quad (6)$$

- Funkcja logistyczna (sigmodalna) (c):

$$z = \frac{1}{1 + \exp(-\beta y)}, \text{ dla } \beta=1 \text{ pochodna: } \frac{\partial z}{\partial y} = z(1 - z) \quad (7)$$

2.1.2 Proces uczenia



Rysunek 3. 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:

$$w(k+1) = w(k) + G(s - z) \quad (8)$$

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

2.1.3 Algorytm propagacji wstecznej

W algorytmie propagacji wstecznej [1], [4], 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” „ \mathbb{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

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

$$\sum_{n=1}^N \mathbb{L}(S, F(y)) = \sum_{n=1}^N \mathbb{L}(s_1, \dots, s_m, F(x_1 * w_1, \dots, x_m * w_m)) \quad (9)$$

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:

$$w_{ji} = w_{ji} + \eta * \frac{\partial}{\partial w_j} \mathbb{L} \quad (10)$$

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 założywszy: $F(y) = z$, $\frac{\partial}{\partial y} F(y) = 1$ 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:

$$w_{ji} = w_{ji} + \eta(s - z) * 1 * x_i \quad (11)$$

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, $(s - z) * x_i$

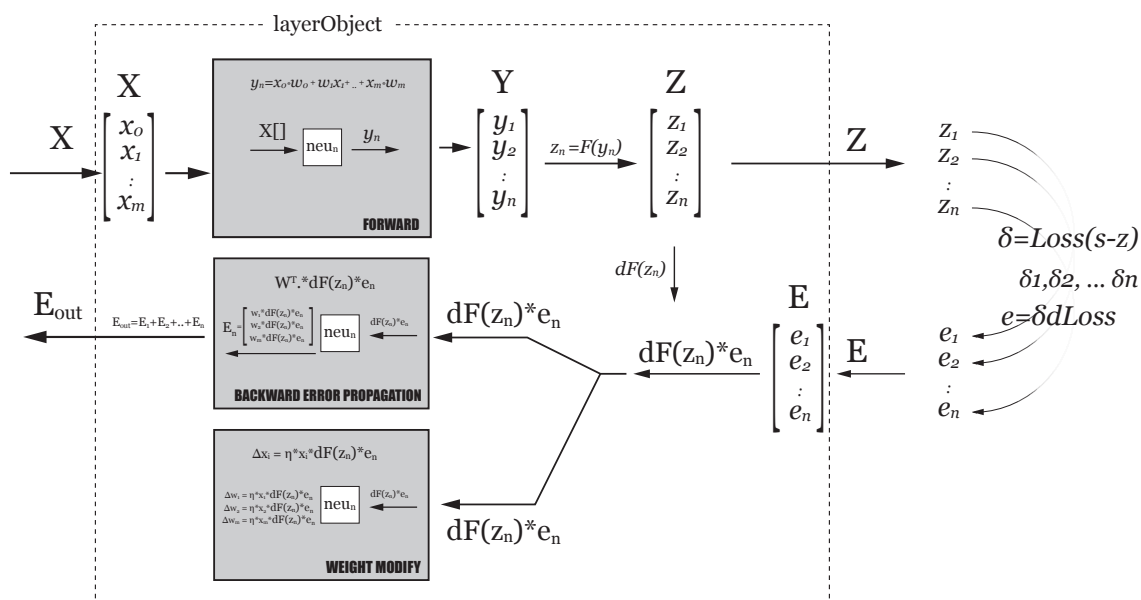
Tabela 1. Zestawienie funkcje aktywacji, pochodnych oraz zmian wag w zależności od funkcji straty [5]

Rozdział 3

Model obiektowy

Proces przetwarzania możemy opisać i analizować jako współpracę obiektów dwu klas: neuronu (*Neuron*) i warstwy (*Layer*). Ujęcie obiektowe umożliwia ściślejszą hermetyzację, ułatwia realizację współbieżności przetwarzania. Realizacja w tym modelu umożliwia wykorzystanie wzorców projektowych w tym: *dziedziczenia*, *interfejsu* czy *obserwatora*.

3.1 Obiekt klasy Layer

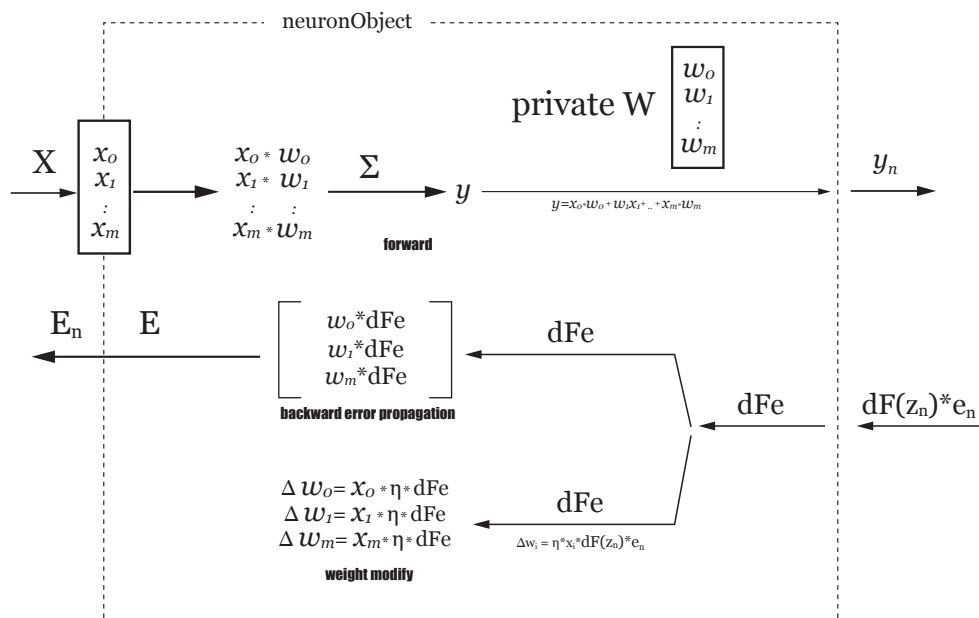


Rysunek 4. Przepływu sygnałów w obiekcie Layer

Każdy obiekt Layer ma własną kolekcję obiektów klasy Neuron. Dysponuje też polami danych np. X , które są zbiorami wartości. I tak pole danych X jest zbiorem wartości x_0, x_1, \dots, x_m . pole danych Y

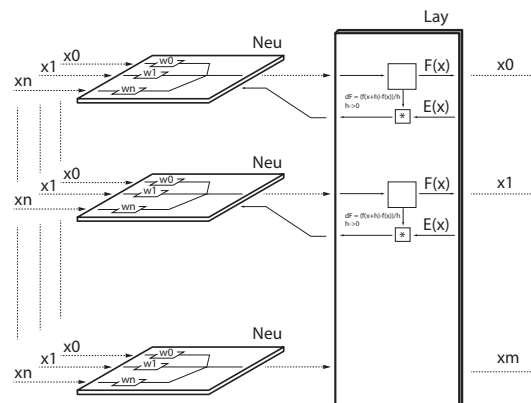
jest zbiorem wartości y_0, y_1, \dots, y_m . Z matematycznego punktu widzenia, można powiedzieć że wektor X jest wektorem zmiennych niezależnych $X = [x_0, x_1, \dots, x_m]$, skoro tak, to i **pochodne składników tych wektorów będą niezależne względem siebie**. Z informatycznego punktu widzenia pole X jest zbiorem wartości x_0, x_1, \dots, x_m , które może być realizowane przez takie struktury danych jak *zbiór*, *lista*, *tablica* czy *wektor*. Zależnie od języka programowania pewne struktury będą wygodniejsze do wykorzystania od innych. Struktura *tablica* jest najprostsza, kolejne wartości są indeksowane liczbą całkowitą, a sama tablica po utworzeniu nie może zmieniać swojego rozmiaru.

Obiekty klasy Layer wywołuje dla wszystkich neuronów ze swojej kolekcji żądania wykonania obliczeń. Obiekty klasy Neuron mają dostęp do obiektu rodzica, a dzięki temu mają także dostęp do niektórych pól danych obiektu Layer. Neurony nie mają jednak bezpośrednich połączeń między sobą.



Rysunek 5. Przepływu sygnałów w obiekcie Neuron

Obiekty klasy Neuron są bardzo proste i lekkie, realizują kilka operacji matematycznych wywoływanych na żądanie warstwy rodzica. Do obliczeń używają wewnętrznych zmiennych wag zorganizowanych w tablicę $W = [w_0, w_1, \dots, w_m]$, oraz dostarczonych przez rodzica zmiennych skalarnych oznaczanych małymi literami np. η , lub tablic skalarów oznaczanych dla rozróżnienia wielkimi literami np. pole X .



Rysunek 6. współpraca klas Neuron / Layer

3.2 Propagacja sygnału pomiędzy obiektami

Pojedynczy neuron odczytuje wartości wektora wejściowego - tablicy X o określonym rozmiarze m , przekazanego przez rodzica. Oblicza iloczyn odpowiednich wag i wartości, a następnie sumuje uzyskane iloczyny. Obliczoną **wartość skalarną zmiennoprzecinkową** - zwraca jako wynik operacji.

```

1 public float forward( float[] X ) {
2     float sum=0;
3     for ( int m=0; m<W.length; m++ ) {
4         yi = X[m]*W[m];
5         sum = sum + yi;
6     }
7     return sum;
8 }

```

obiekt Layer dla otrzymanych wartości y_1, y_2, \dots, y_n oblicza wielkość funkcji aktywacji $z_i = f(y_i)$. Metoda wyliczająca wielkość funkcji aktywacji zależnie do rodzaju warstwy:

```

1 private float F ( float y ){
2     float z;
3     switch (this.lType) {
4         case sigmod: { return ( 1/(1 + Math.exp( -y ))); }
5         case linear:
6             default: { z=y; break; }
7     }
8     return z;
9 }

```

Wielkości te zebrane w tablicę tworzą wartość wyjściową Z . Przy okazji obliczamy także wartość pochodnych funkcji F w punktach y_i i zapisujemy w tablicy $dFofZ$).

```
1 public void forward(){
2     for ( int n=0; n<neurons.length; n++ ) {
3         Y[n] = neurons[n].forward( X );
4         Z[n] = F ( Y[n] );
5         dFofZ[n] = dF( Z[n] );
6     }
7 }
```

Zestawienie pochodnych funkcji straty $Loss$ oraz funkcji aktywacji $dF(y)$ zależnie od rodzajów warstw 1.

```
1 private float dF (float z ){
2     float df;
3     switch (lType) {
4         case sigmod: { df = z*(1-z); break; }
5         case linear:
6         default: { df=1; break; }
7     }
8     return df;
9 }
```

Pierwsze programy napisane przy użyciu modelu wykonują poprawnie przykładowe zadania z [5].

Rozdział 4

Przetwarzanie numeryczne

Z punktu widzenia matematyków różnica pomiędzy 1.0000000000000001 a 1.0 jest spora, ponieważ pomiędzy tymi wartościami mamy nieskończenie wiele liczb, natomiast przy obliczeniach, których dokonujemy używając maszyn cyfrowych musimy zważyć, czy zwiększać dokładność zwiększając ilość bitów zajmowanych przez liczby, jednocześnie zmniejszać szybkość operacji, czy wykorzystać mniej dokładne reprezentacje zyskując na szybkości pewnych operacji.

4.1 Kodowanie

Najczęściej używaną reprezentacją liczb rzeczywistych jest opisana przez normę IEEE 754. Norma ta definiuje zapis liczby w 32, 64 lub 128 bitach. W językach programowania dostępne jako klasy *Float*, *Double*.

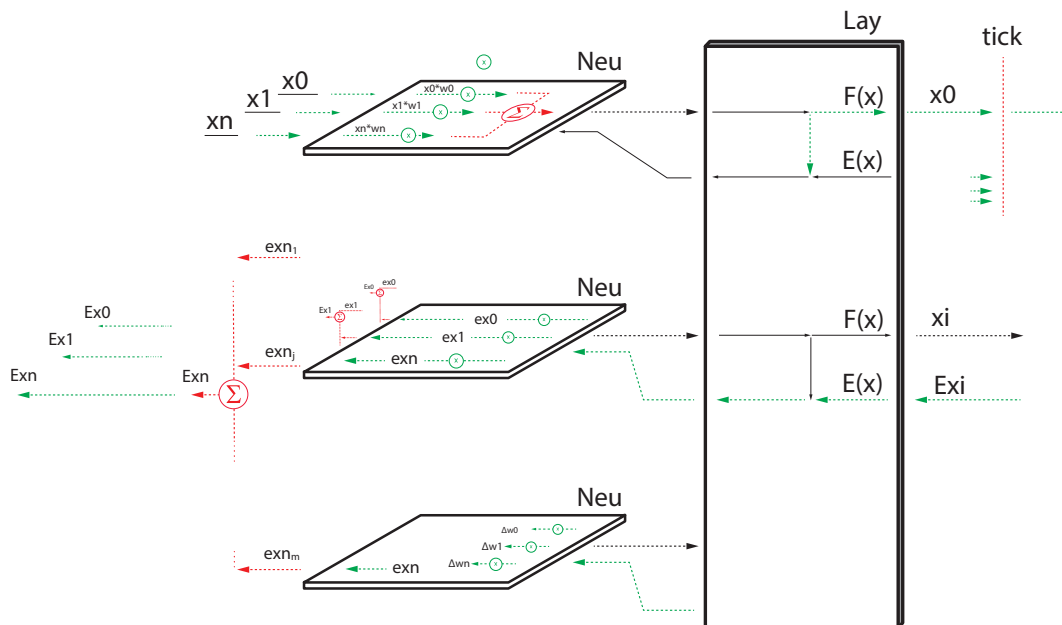
- Matlab: przypisanie $a=1.0000000000000001$ // $a=1$;
- Python: `print (.0006000000000000000001)` // 0.0006.
- Java: $0.1 + 0.2 = 0.30000000000000004$.

Do zastosowań specjalnych możemy użyć specjalnych formatów zapisu np. *BigDecimal* w Java o większej dokładności (większej liczbie cyfr znaczących), lecz o dużo dłuższym czasie operacji.

Float - 0.002 [sek.] / *BigDecimal* 0.042 [sek.]

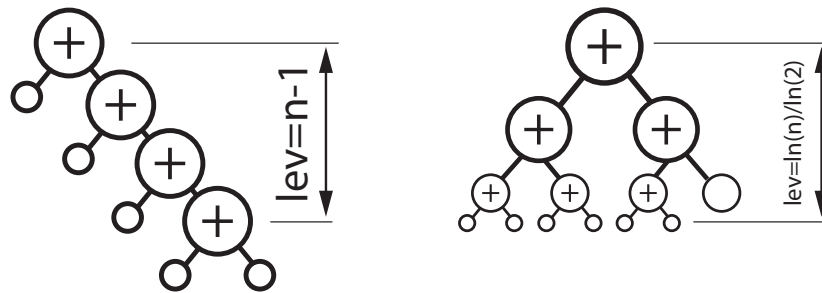
4.2 operacje równoległe

operacje niezależne - na rysunku oznaczone kolorem zielonym - takie jak mnożenie wag przez wartości wejściowe możemy wykonywać równoległe, wykorzystując osobne wątki, procesy, rdzenie. Operacje sumowania nie są operacjami niezależnymi.



Rysunek 7. operacje niezależne i operacje wymagające synchronizacji

Sumowanie $a+b+c+d$ realizowana jest analogicznie do zdegenerowanego drzewa binarnego czyli $((a+b)+c)+d$... nie jest operacją równoległą. Przy dużej ilości składników możemy spróbować zastosować zwykłe drzewo binarne nie zdegenerowane, a operacja dodawania może stać się częściowo równoległa $((a+b) + (c+d) \dots)$. Maszyny cyfrowe obecnie nie są wyposażone w sumatory wielokanałowe pozwalające dodawać więcej niż 2 liczby jednocześnie. Najprostsza realizacja fizyczna takiego sumatora jest możliwa do realizacji w układach FPGA, można także zastosować nawiasy.



Rysunek 8. drzewo zdegenerowane i niezdegenerowane

4.3 Dodawanie równoległe

różnice w szybkości dodawania: (C++)

$$b = a1 + a2 + a3 + a4 + \dots + a1024;$$

czas wykonania: **0.247875 [sek.]**

$$b = (((((a1 + a2) + (a3 + a4)) + ((a5 + a6) \dots$$

czas wykonania: **0.09568 [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. Podsumowując 1023 dodawania możemy wykonać w czasie takim jak 10 dodawań korzystając z osobnych 512 rdzeni CUDA zakładając że dostęp do pamięci będzie niezależny. Korzystając ze zwykłego CPU poszczególne rdzenie będą czekały się w kolejce po odczyt wartości składników oraz w drugiej kolejce do zapisu wyników w pamięci. Jeśli kod będzie "optymalizowany" np. przez procesor i zmieni się kolejność niektórych działań, wynik sumowania może być nieprawidłowy.

4.4 Wątki

Rozdzielenie przetwarzania w ramach procesu na wątki umożliwia równoległe przetwarzanie danych i jest wspierane przez nowoczesne procesory. Proces będzie miał do dyspozycji kilka niezależnych jednostek liczących ALU, więc niezależne operacje będą wykonywane w tym samym czasie. A zatem pojedyncze mnożenie zajmie tyle czasu co kilka mnożeń na kilku rdzeniach.

4.5 HyperWątki HT

Technologia wprowadzone przez Inter HyperThread(R) tworzy dwie kolejki rozkazów dla jednego rdzenia, co dla systemu operacyjnego wygląda na dwa niezależne rdzenie. W rezultacie w czasie gdy jedna kolejka wykorzystuje jednostkę liczącą, w drugiej kolejce może być bez przeszkód wykonana instrukcja odczytania lub zapisu danych do pamięci - a te instrukcje zajmują nawet kilkanaście cykli zegara. W tej sytuacji mamy zwiększone wykorzystanie potencjału pojedynczego rdzenia.

4.6 Pamięć operacyjna

Ponieważ dostęp do *magistrali pamięci* jest synchroniczny, to nawet jeśli samo przetwarzanie przez ALU będzie równoległe, to odczyt i zapis w pamięci będzie miał charakter zbliżony do synchronicznego. Nawet przy dwu niezależnych kanałach po 128 bitów w każdym (DDR5).

4.7 Rdzenie CUDA

Procesory ogólnego przeznaczenia, mimo wielu wysiłków, nie są w stanie zapewnić w pełni równoległego przetwarzania. Równoległe przetwarzanie mogą zapewnić np. procesory graficzne, w których znajduje się wiele jednostek obliczeniowych, ale przede wszystkim dostęp do pamięci jest rzeczywiście równoległy. Procesory graficzne projektowane były do obliczeń translacji punktów w przestrzeni, a operacje te sprowadzały się do mnożenia macierzy współrzędnych znormalizowanych punktu przez macierz translacji:

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

a po rozpisaniu mamy:

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

i są to te same obliczenia, które wykonujemy wielokrotnie podczas pracy z matematycznymi modelami sieci neuronowych:

$$y = x_0 * w_0 + x_1 * w_1 + x_2 * w_2 \dots + x_n * w_n;$$

a jeśli będą one wykonane przez osobne rdzenie CUDA, a wyniki będą zapisane w pamięci VRam o dostępie równoległym - wtedy wiele takich operacji zajmie dokładnie tyle samo czasu co pojedyncza operacja. Idealnie byłoby gdyby cała praca odbywała się w VRAM i na rdzeniach CUDA bez konieczności przesyłania danych w każdym cyklu do pamięci operacyjnej RAM. Jeśli zbiór danych w całości znajdowałby się w VRAM, to przejście sygnałów przez jedną warstwę sieci trwałoby tyle co jedna operacja, a propagacja przez sieć trwałaby tyle razy dłużej ile warstw mamy w sieci. Powrót sygnału w postaci wielkości błędu podobnie zajmowałby mniej więcej tyle czasu. (O ile operacje mnożenia możemy wykonać równolegle to operacje wielokrotnego dodawania już nie do końca). Równoległe zmiany kilku wag również byłyby dokonywane w takim czasie jak zmiana pojedynczej wagi.

Wprowadzanie na rynek gier sieciowych wymagających do zabawy coraz wydajniejszych kart graficznych oraz pojawienie się wirtualnych walut typu Bitcoin wydobywanych - obliczanych właśnie przy użyciu takiego sprzętu zwiększyła zapotrzebowanie na coraz wydajniejsze jednostki mimo sporych kosztów jak dla użytkownika indywidualnego.

Dostępność cenowa coraz wydajniejszych jednostek, które przyspieszają obliczenia nawet o kilka rzędów wielkości, spowodowała, że budowanie prawdziwie używalnych modeli stało się możliwe. A temat „sztucznej inteligencji” znany wcześniej jedynie garstce naukowców, stał się nagle tematem bardzo popularnym.

4.8 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.

4.9 Python

Metody do obliczeń sieci neuronowych w języku Python dostarczone są w bibliotekach scikit-learn i TensorFlow2 a dostarczają one gotowych metod przydatnych w obliczeniach sieci neuronowych. Wykorzystują one możliwości równoległego przetwarzania, które oferuje sprzęt na którym uruchamiany jest kod.

<https://www.geeksforgeeks.org/matplotlib-tutorial/> matplotlib

Rozdział 5

Praktyka

Przykłady obliczeń Python i Matlab zostały zaczerpnięte z [3]. **Pełen kod dostępny na github:** <https://github.com/piotrHeinzelman/inz/tree/main/MixedProj/01.polyfit> w przypadku Matlab i Python korzystam z dostępnych funkcji, w przypadku Java obliczam wg. wzoru 28. Obliczenia różnymi metodami dają zbliżone wyniki, więc zakładam że moje implementacje są poprawne.

5.1 Metodyka

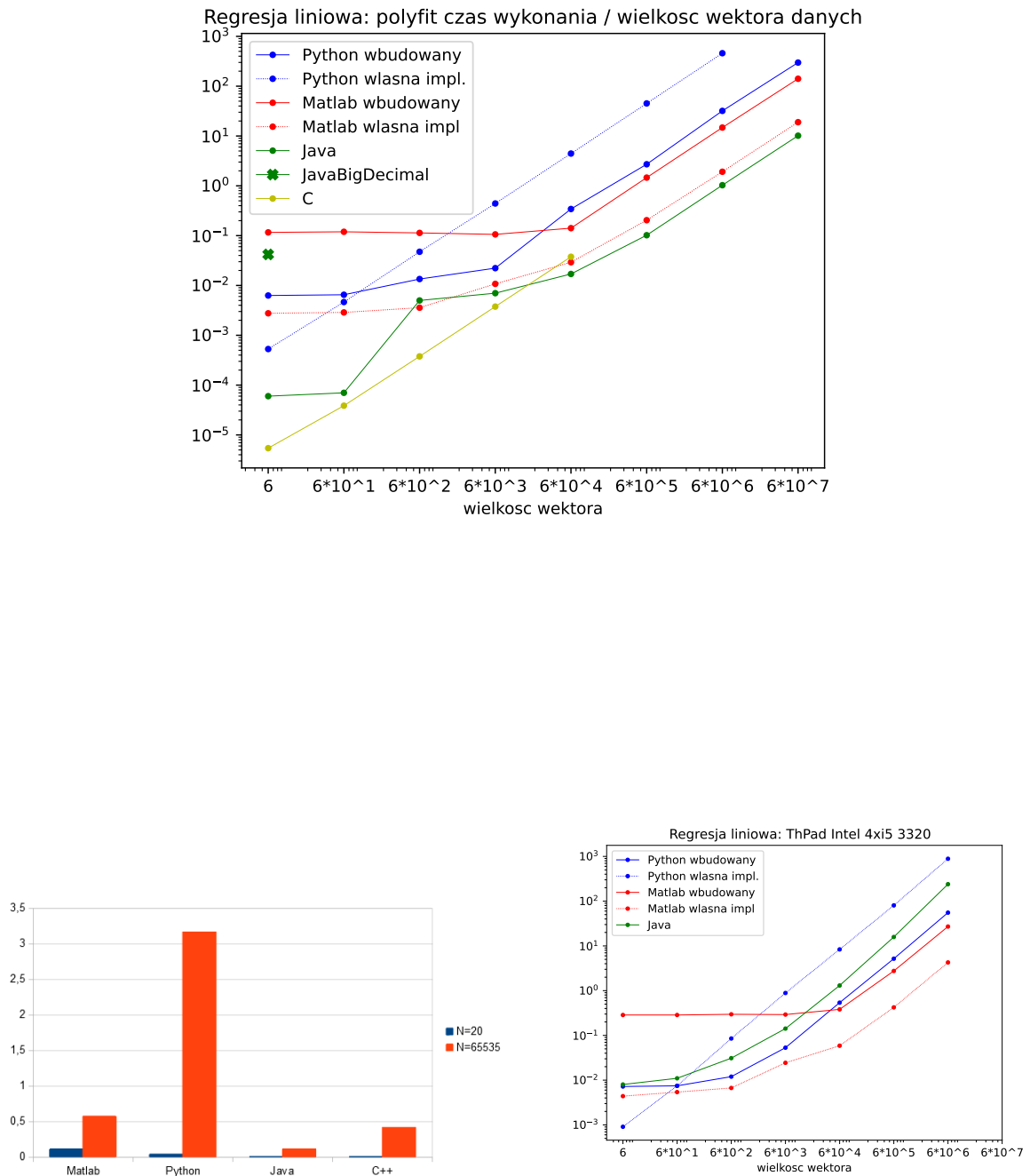
W pracy tam gdzie możliwe staram się wykorzystać dostarczone funkcje liczące. I tak dla Matlab i dla Python wykorzystałem zaimplementowaną funkcję "polyfit". W kodzie Java musiałem sam zaimplementować funkcję liczącą dla porównania wydajności. W programach nie porównuję czasu ładowania i przygotowania danych ponieważ chcę wykonywać obliczenia dla tych samych wartości czytanych z tych samych plików, natomiast nie jest moim celem optymalizacja wczytywania danych z pliku. Programy w kolejnych językach uruchamiam poniższym kodem:

```
1 ./matRun >> output.txt
2 ./pyRun >> output.txt
3 ./javaRun >> output.txt
4
5 //matRun
6 echo "--- Matlab app start: ---"
7 /usr/local/MATLAB/R2024a/bin/matlab -nodisplay -nosplash -nodesktop -
   batch 'run matlab.m'
8
9 //pyRun
10 echo "--- Python app start: --- "
11 python main.py
12
13 // javaRun
14 javac Main.java
15 echo "--- Java app start: ---"
16 java Main
```

5.2 Realizacja obliczeń - Jednowymiarowa regresja liniowa

Kod Matlab:

```
1 //          --- MATLAB ---
2
3 a = polyfit(x,y,1);
4
5 //          --- Python ---
6
7 a = np.polyfit(x,y,1)
8
9 //          --- Java ---
10
11 for (int i = 0; i < x.length; i++) {
12     xsr += x[i];
13     ysr += y[i];
14 }
15 xsr = xsr / x.length;
16 ysr = ysr / y.length;
17
18 for (int i = 0; i < x.length; i++) {
19     sumTop += ((x[i] - xsr) * (y[i] - ysr));
20     sumBottom += ((x[i] - xsr) * (x[i] - xsr));
21 }
22 w1 = sumTop / sumBottom;
23 w0 = ysr - w1 * xsr;
24
25 //          --- C++ ---
26
27 for ( int i=0; i<len; i++ ){
28     xsr += X[i];
29     ysr += Y[i];
30 }
31 xsr=xsr / len; ysr=ysr / len;
32 double sumTop=0.0;
33 double sumBottom=0.0;
34
35 for ( int i=0;i<len;i++ ){ // xtmp = X[i]-sr ! ;
36     sumTop += ((X[i]-xsr)*(Y[i]-ysr));
37     sumBottom += ((X[i]-xsr)*(X[i]-xsr));
38 }
39 w1 = sumTop / sumBottom;
40 w0 = ysr -(w1 * xsr) ;
41
42 .
```



Rysunek 9. Porównanie czasów obliczania regresji liniowej

Rozdział 6

Sieci głębokie

Za protoplastę sieci głębokich można uznać zdefiniowany na początku lat dziewięćdziesiątych wielowarstwowy **neocognitron** Fukushima, Prawdziwy rozwój tych sieci zawdzięczamy jednak profesorowi LeCun, który zdefiniował podstawową strukturę i algorytm uczący specjalizowanej **sieci konwolucyjnej** *Convolutional Neural Network* skracanej tradycyjnie do CNN [3].

6.1 CNN

```
1 https://www.mathworks.com/matlabcentral/fileexchange/74760-image-  
   classification-using-cnn-with-multi-input-cnn?s_tid=  
   srchtitle_support_results_1_CNN
```

6.2 Python TensorFlow

.. // derative of convolution

<https://www.physicsforums.com/threads/how-do-you-derive-the-derivative-of-a-convolution.403002/>
<https://dsp.stackexchange.com/questions/46746/how-to-evaluate-derivative-of-convolution-integral>
<https://en.wikipedia.org/wiki/Convolution>

Rozdział 7

Podsumowanie

- Teoria: wiemy jak ma działać, ale jednak nie działa.
- Praktyka: działa, ale nie wiemy – dlaczego?
- Łączymy teorię z praktyką: nic nie działa i nie wiemy, dlaczego.

Więcej informacji na temat \LaTeX :

- <https://www.overleaf.com/learn> – przystępny tutorial na stronie Overleaf,
- <https://www.latex-project.org/> – strona domowa projektu,
- <https://www.tug.org/begin.html> – dobry zbiór odnośników do innych materiałów.

Powodzenia!

Bibliografia

- [1] Józef Korbicz Andrzej Obuchowicz, D. U., *Sztuczne sieci neuronowe: podstawy i zastosowania*. Akademicka Oficyna wydawnicza PLJ, 1994, ISBN: 83-7101-197-0.
- [2] R., W., *Metody programowania nieliniowego*. Warszawa: WNT, 1986.
- [3] Stanisław Osowski, R. S., *Matematyczne modele uczenia maszynowego w językach MATLAB i PYTHON*. Oficyna Wydawnicza Politechniki Warszawskiej, 2023, ISBN: 978-83-8156-597-4.
- [4] 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.
- [5] Włodzimierz Kasprzak, prof. dr hab. inż., „Metody sztucznej inteligencji (2024L), MSI-C6.pdf,” materiał dydaktyczny.

Spis rysunków

1	Model sztucznego neuronu [1]	13
2	Funkcje aktywacji	15
3	Proces uczenia elementu perceptronowego	16
4	Przepływu sygnałów w obiekcie Layer	19
5	Przepływu sygnałów w obiekcie Neuron	20
6	współpraca klas Neuron / Layer	21
7	operacje niezależne i operacje wymagające synchronizacji	24
8	drzewo zdegenerowane i niezdegenerowane	25
9	Porównanie czasów obliczania regresji liniowej	31

Spis tabel

1	Zestawienie funkcje aktywacji, pochodnych oraz zmian wag w zależności od funkcji straty [5]	18
---	---	----

Spis załączników

1	Uogólniona reguła delty	45
2	Algorytm propagacji wstecznej	47
3	Jednowymiarowa regresja liniowa	49

Załącznik 1

Uogólniona reguła delty

Rozważmy sieć jednowarstwową z elementami przetwarzającymi o nieliniowej, lecz niemalejącej i różniczkowalnej funkcji aktywacji F wówczas zmianę wag przy prezentacji μ -tego wzorca można opisać równaniem:

$$\Delta w_{ji} = -\eta \frac{\partial \xi}{\partial w_{ji}} = -\eta \frac{\partial \xi}{\partial y_j} \frac{\partial y_j}{\partial w_{ji}} = -\eta \frac{\partial \xi}{\partial z_j} \frac{\partial z_j}{\partial y_j} \frac{\partial y_j}{\partial w_{ji}}, \quad (12)$$

przy czym:

$$\frac{\partial \xi}{\partial z_j} = (s - z_j), \text{ z def. } ((\frac{1}{2}(x - y)^2)' = (x - y), \quad (13)$$

$$\frac{\partial z_j}{\partial y_j} = F'(y_j), \quad (14)$$

$$\frac{\partial y_j}{\partial w_{ji}} = x_i; \quad (15)$$

stąd ostatecznie wzór przyjmuje postać:

$$\Delta w_{ji} = \eta F'(y)(S - z_j)x_i = \eta F'(y)(S - F(y))x_i = \eta F'(y)(s_1, s_2 \dots s_m - F(x_1 * w_1, x_2 * w_2 \dots))x_i \quad (16)$$

oraz dla warstwy ukrytej:

$$\Delta w_{ji} = \eta F'(y_j) \sum_{i=1}^{n_{m+1}} F'(y_j)(s^{m+1} - z_j^{m+1})w_{ji} \quad (17)$$

Załącznik 2

Algorytm propagacji wstecznej

Algorytm ten [1], podaje on przepis na zmianę wag w_{ij} dowolnych połączeń elementów przetwarzających rozmieszczonych w sąsiednich warstwach sieci jednokierunkowej. Jest on oparty na minimalizacji sumy kwadratów błędów uczenia z wykorzystaniem optymalizacyjnej metody największego spadku [2]. Dzięki zastosowaniu specyficznego sposobu propagowania błędów uczenia sieci powstałych na wyjściu, tzn. przesyłania ich do warstwy wyjściowej od wejściowej, algorytm propagacji wstecznej stał się jednym z najskuteczniejszych algorytmów uczenia sieci. Rozważamy sieć jednowarstwową o liniowych elementach przetwarzających. Załóżmy, że mamy P -elementowy zbiór wzorców. Przy prezentacji μ -tego wzorca możemy zdefiniować błąd:

$$\delta_j^\mu = s_j^\mu - z_j^\mu = s^\mu - y_j^\mu = s^\mu - \sum_{i=0}^m w_{ij} x_i^\mu, \quad (18)$$

gdzie s_j^μ , y_j^μ oznaczają odpowiednio oczekiwane i aktualne wartości wyjścia j -tego elementu oraz ważoną sumę wejść wyznaczoną w jego sumatorze przy prezentacji μ -tego wzorca. x_i^μ i -ta składowa μ -tego wektora wejściowego, w_{ji} - oznacza wagę połączenia pomiędzy j -tym elementem warstwy wyjściowej a i -tym elementem warstwy wejściowej. m -liczba wejść.

Jako miarę błędu sieci ξ wprowadzimy sumę po wszystkich wzorcach błędów powstałych przy prezentacji każdego z nich:

$$\xi = \sum_{\mu=0}^P \xi_\mu = \frac{1}{2} \sum_{\mu=1}^P \sum_{j=1}^n (s^\mu - y^\mu)^2, \quad (19)$$

gdzie

$$\xi_\mu = \frac{1}{2} \sum_{j=1}^n (s^\mu - y^\mu)^2, \quad (20)$$

Problem uczenia sieci to zagadnienie minimalizacji funkcji błędu ξ . Jedną z najprostszych metod minimalizacji jest gradientowa metoda największego spadku [2]. Jest to metoda iteracyjna, która poszukuje kolejnego lepszego punktu w kierunku przeciwnym do gradientu funkcji celu w danym punkcie. Stosując powyższą metodę do uczenia sieci, zmiana Δw_{ji} wagi połączenia winna spełniać relację:

$$\Delta w_{ji} = -\eta \frac{\partial \xi}{\partial w_{ji}} = -\eta \sum_{\mu=1}^P \frac{\partial \xi_\mu}{\partial w_{ji}} = -\eta \sum_{\mu=1}^P \frac{\partial \xi_\mu}{\partial z_j^\mu} \frac{\partial z_j^\mu}{\partial w_{ji}} \quad (21)$$

gdzie η oznacza współczynnik proporcjonalności. W przypadku elementów liniowych mamy:

$$\frac{\partial \xi_\mu}{\partial z_j^\mu} = -(s_j^\mu - z_j^\mu) = -\delta_j^\mu, \quad (22)$$

$$\frac{\partial z_j^\mu}{\partial w_{ji}} = \frac{\partial y_j^\mu}{\partial w_{ji}} = x_i^\mu \quad (23)$$

stąd otrzymujemy:

$$\Delta w_{ji} = \eta \sum_{\mu=1}^P \delta_j^\mu x_i^\mu \quad (24)$$

ostatecznie pełną regułę zapiszemy:

$$w_{ji}(k+1) = w_{ji}(k) + \Delta w_{ji}, \quad (25)$$

Konsekwentna realizacja metody największego spadku wymaga dokonywania zmian wag dopiero po zaprezentowaniu sieci pełnego zbioru wzorców. W praktyce stosuje się jednak zmiany wag po każdej prezentacji wzorca zgodnie ze wzorem:

$$\Delta^\mu w_{ji} = -\eta \frac{\partial \xi_\mu}{\partial w_{ji}} = \eta \delta_j^\mu x_i^\mu, \quad (26)$$

Załącznik 3

Jednowymiarowa regresja liniowa

[4] Funkcja liniowa jednej zmiennej to funkcja w postaci $y = w_1x + w_0$; współczynniki w_0 i w_1 możemy traktować jak wagi, i możemy je traktować łącznie jako wektor $\mathbf{W} = \langle w_0, w_1 \rangle$ a samo przekształcenie można utożsamić z iloczynem skalarnym $y = \mathbf{W}^* \langle 1, x \rangle$. Zadanie dopasowania najlepszej hipotezy hw wiążącej te dwie wielkości nosi nazwę regresji liniowej. Matematycznie dopasowanie to sprowadza się do znalezienia wektora \mathbf{W} minimalizującego funkcję straty, zgodnie z teorią Gaussa jako miarę tej straty przyjmuje się sumę miar dla wszystkich przykładów:

$$Loss(h_w) = \sum_{j=1}^N L_2(y_j, hw(x_j)) = \sum_{j=1}^N L_2(y_j - hw(x_j))^2 = \sum_{j=1}^N L_2(y_j - (w_1x + w_0))^2,$$

Naszym celem jest znalezienie optymalnego wektora \mathbf{W}

$$\mathbf{W} = \operatorname{argmin} Loss(h_w)$$

Gdy funkcja ciągła osiąga minimum w danym punkcie, pierwsze pochodne cząstkowe po argumentach tej funkcji zerują się w tym punkcie; w kontekście regresji liniowej nasza funkcja $Loss(h_w)$ jest funkcją dwu zmiennych: w_0 i w_1 , których wartości w punkcie minimum określone są przez układ równań:

$$\begin{cases} \frac{\partial}{\partial w_0} \sum (y_j - (w_1x + w_0))^2 = 0, \\ \frac{\partial}{\partial w_1} \sum (y_j - (w_1x + w_0))^2 = 0, \end{cases}$$

Rozwiązaniem takiego układu są wartości:

$$w_1 = \frac{N(\sum x_j y_j) - (\sum x_j)(\sum y_j)}{N(\sum x_j^2) - (\sum x_j)^2}, w_0 = \frac{\sum y_j - w_1(\sum x_j)}{N}, \quad (27)$$

Dla dużych N [4] musimy użyć następującej, równoważnej postaci rzeczonych wzorów:

$$w_1 = \frac{\sum (x_j - \bar{x})(y_j - \bar{y})}{\sum (x_j - \bar{x})^2}, w_0 = \bar{y} - w_1 \bar{x}, \quad (28)$$

gdzie \bar{x} i \bar{y} są średnimi arytmetycznymi:

$$\bar{x} = \frac{\sum x_j}{N}, \bar{y} = \frac{\sum y_j}{N}, \quad (29)$$