Algorytmy macierzowe
Sprawozdanie z laboratorium nr. 1

Szymon Nowak-Trzos, Dominik Adamczyk

13 listopada 2023

1 Treść zadania

Zadanie polegało na zaimplementowaniu u porównaniu trzech algorytmów mnożenia macierzy:

- Rekurencyjnie metodą Binet'a
- Rekurencyjnie metodą Strassena
- Metodą znalezioną przez AI

2 Wstęp

Zgodnie z treścią zaimplementowane zostały wszystkie wskazane algorytmy, a także naiwny iteracyjny algorytm w celach porównawczych. Całe zadanie zostało napisane w języku Python 3. Algorytmy zostały przystosowane do mnożenia macierzy o wymiarach $2^n \times 2^n$. Dodatkowo przeprowadziliśmy testy na mnożeniu macierzy rozmiarów $4 * 2^n \times 4 * 2^n$ z $4 * 2^n \times 5 * 2^n$ dla algorytmu zaproponowanego przez AI. Umożliwiło to pokazanie mniejszej liczby wykonywanych operacji przez ten algorytm dla niektórych przypadków.

3 Użyte rozwiązania

Aby zliczyć ilość oberacji addytywnych i multiplikatywnych stworzona została nowa klasa Num() dziedzicząca po pythonowej klasie float. Umożliwiło nam to wykonywanie zliczanie ilości operacji w liczniku zawartym w tej klasie.

```
class Num(float):
    counts = Counter(mul=0, add=0)
    def __add__(self, other):
        self.counts["add"] += 1
        return Num(super(Num, self).__add__(other))

    def __radd__(self, other):
        self.counts["add"] += 1
        return Num(super(Num, self).__radd__(other))

    def __mul__(self, other):
        self.counts["mul"] += 1
        return Num(super(Num, self).__mul__(other))

    ### Other overloaded methods

    def reset_counter(self):
        self.counts["mul"] = 0
        self.counts["add"] = 0
```

Wszystkie algorytmy napisane zostały w klasie Matrix MatrixMultiplier. Obiekt takiej klasy przyjmuje dwie macierze, które chcemy wymnożyć, następnie odpowiednia metoda odpowiada za mnożenie. W klasie tej znajdują się też pomocnicze metody (np. do dzielenia macierzy na 4 części)

4 Algorytmy

Poniżej prezentujemy opisy i kody implementowanych algorytmów

4.1 Algorytm Iteracyjny

Dla celów porównawczych naiwny algorytm iteracyjny. Jego złożoność to $O(n^3)$.

```
    def iterative_wrap(self, A, B):
        C = np.zeros((A.shape[0], B.shape[1]))
        for i in range(A.shape[0]):
            for j in range(B.shape[1]):
                s = Num(0)
                for k in range(B.shape[0]):
                    s += A[i, k] * B[k, j]
                C[i, j] = s
        self.result = C
        return C
```

4.2 Algorytm Binet'a

Ten algorytm bazuje na podzieleniu macierzy na 4 podmacierze i potraktowania tych podmacierzy jak zwykłych liczb w mnożeniu dwóch macierzy 2×2, z tą różnicą że zamiast wykonywać mnożenie na liczbach wykonujemy rekurencyjnie mnożenie macierzy tą metodą.

Złożoność obliczeniowa tego algorytmu wynosi $O(n^3)$ - gdyż przy mnożeniu dwóch macierzy o wymiarach $n \times n$ wykonuje się 8 mnożeń na każdym etapie. Czyli możemy zapisać:

$$O(8^{\log_2 n}) = O((2^{\log_2 n})^3) = O(n^3)$$

```
        def binet_wrap(self, A, B):
        C = np.zeros((1, 1), dtype=A.dtype)

        if A.shape[0] == A.shape[1] and A.shape[1] == B.shape[0] and B.shape[0] == B.
                                                    shape[1]:
            if A.shape[0] == 1:
                return np.array([[Num(A[0, 0] * B[0, 0])]], dtype=Num)
            elif self.is_power_of_two(B.shape[0]):
                A11, A12, A21, A22 = self.matrix_partition_sq2(A)
                B11, B12, B21, B22 = self.matrix_partition_sq2(B)
                C = self.matrix_repartition_sq2(
                self.binet_wrap(A11, B11) + self.binet_wrap(A12, B21),
                self.binet_wrap(A11, B12) + self.binet_wrap(A12, B22),
                self.binet_wrap(A21, B11) + self.binet_wrap(A22, B21),
                self.binet_wrap(A21, B12) + self.binet_wrap(A22, B22)
                )

        return C
```

4.3 Algorytm Strassena

W algorytmie tym podobnie jak w poprzednim dzielimy macierz na 4 części. Następnie jednak tworzone są nowe podmacierze P z podzielonej wcześniej macierzy - do obliczania podmacierzy używamy rekurencyjnie algorytmu Strassena. W ostatnim kroku obliczana jest maicierz wynikowa powstała z podmacierzy. Przewagą algorytmu Strassena nad algorytmem Bineta jest fakt że wykonuje on 7 mnożeń w każdym kroku. Wobec tego możemy obliczyć:

$$O(7^{\log_2 n}) = O(n^{\log_2 7}) \approx O(n^{2.81})$$

```
        def strassen_wrap(self, A, B):
        C = np.zeros((1, 1), dtype=A.dtype)

        if A.shape[0] == A.shape[1] and A.shape[1] == B.shape[0] and B.shape[0] == B.
                                                    shape[1]:
            if A.shape[0] == 1:
                return np.array([[Num(A[0, 0] * B[0, 0])]], dtype=Num)
            elif B.shape[0] > 1 and (B.shape[0] & (B.shape[0] - 1)) == 0:
                A11, A12, A21, A22 = self.matrix_partition_sq2(A)
                B11, B12, B21, B22 = self.matrix_partition_sq2(B)
                P1 = self.strassen_wrap(A11 + A22, B11 + B22)
                P2 = self.strassen_wrap(A21 + A22, B11)
                P3 = self.strassen_wrap(A11, B12 - B22)
                P4 = self.strassen_wrap(A22, B21 - B11)
                P5 = self.strassen_wrap(A11 + A12, B22)
                P6 = self.strassen_wrap(A21 - A11, B11 + B12)
                P7 = self.strassen_wrap(A12 - A22, B21 + B22)

                C = self.matrix_repartition_sq2(P1 + P4 - P5 + P7, P3 + P5,
                                                P2 + P4, P1 - P2 + P3 + P6)

        return C
```

4.4 Algorytm zaproponowany przez AI

Duża rodzina algorytmów mnożenia macierzy została znaleziona przez sieć neuronową Alpha Tensor. Funkcja tworząca algorytmy dla konkretnych macierzy, a także opis trójwymiarowych tensorów z których tworzone są funkcje pochodzą z repozytorium: https://github.com/google-deepmind/alphatensor.

```
    def algorithm_from_factors(self, factors: np.ndarray, n, m, k):
        factors = [factors[0].copy(), factors[1].copy(), factors[2].copy()]
        rank = factors[0].shape[-1]
        factors[0] = factors[0].reshape(n, m, rank)
        factors[1] = factors[1].reshape(m, k, rank)
        factors[2] = factors[2].reshape(k, n, rank)
        factors[2] = factors[2].transpose(1, 0, 2)

        def f(a, b):
            n = a.shape[0]
            m = a.shape[1]
            l = b.shape[1]

            result = np.array([[0 for _ in range(l)] for _ in range(n)], dtype=Num)
            for alpha in range(rank):
                left = Num()
                for i in range(n):
                    for j in range(m):
                        if factors[0][i, j, alpha] != 0:
                            curr = factors[0][i, j, alpha] * a[i][j]
                            left += curr
                right = Num()
                for j in range(m):
                    for k in range(l):
                        if factors[1][j, k, alpha] != 0:
                            curr = factors[1][j, k, alpha] * b[j][k]
                            right += curr
                matrix_product = left * right
```

```
                for i in range(n):
                    for k in range(l):
                        if factors[2][i, k, alpha] != 0:
                            curr = factors[2][i, k, alpha] * matrix_product
                            result[i, k] += curr
            return result

        return f

    def alphatensor(self, A, B):
        assert A.shape[1] == B.shape[0]
        n = A.shape[0]
        m = A.shape[1]
        k = B.shape[1]
        factors = self.factorizations[str(n) + ',' + str(m) + ',' + str(k)]
        matrix_mul_algorithm = self.algorithm_from_factors(factors, n, m, k)
        return matrix_mul_algorithm(A, B)
```

Minimalna złożoność algorytmów znalezionych przez Alpha Tensor wynosi $O(n^{2.373})$, jednak nie każdy algorytm uzyska taką złożoność. Przykładowo dla macierzy rozmiarów 2x2 i 4x4 złożonych z liczb rzeczywistych Alpha Tensor osiąga takie same wyniki jak algorytm Strassena. Oznacza to że w praktyce macierze których bok jest potęgą dwójki będą mnożone z taką samą złożonością dla Strassena i Alpha Tensor.

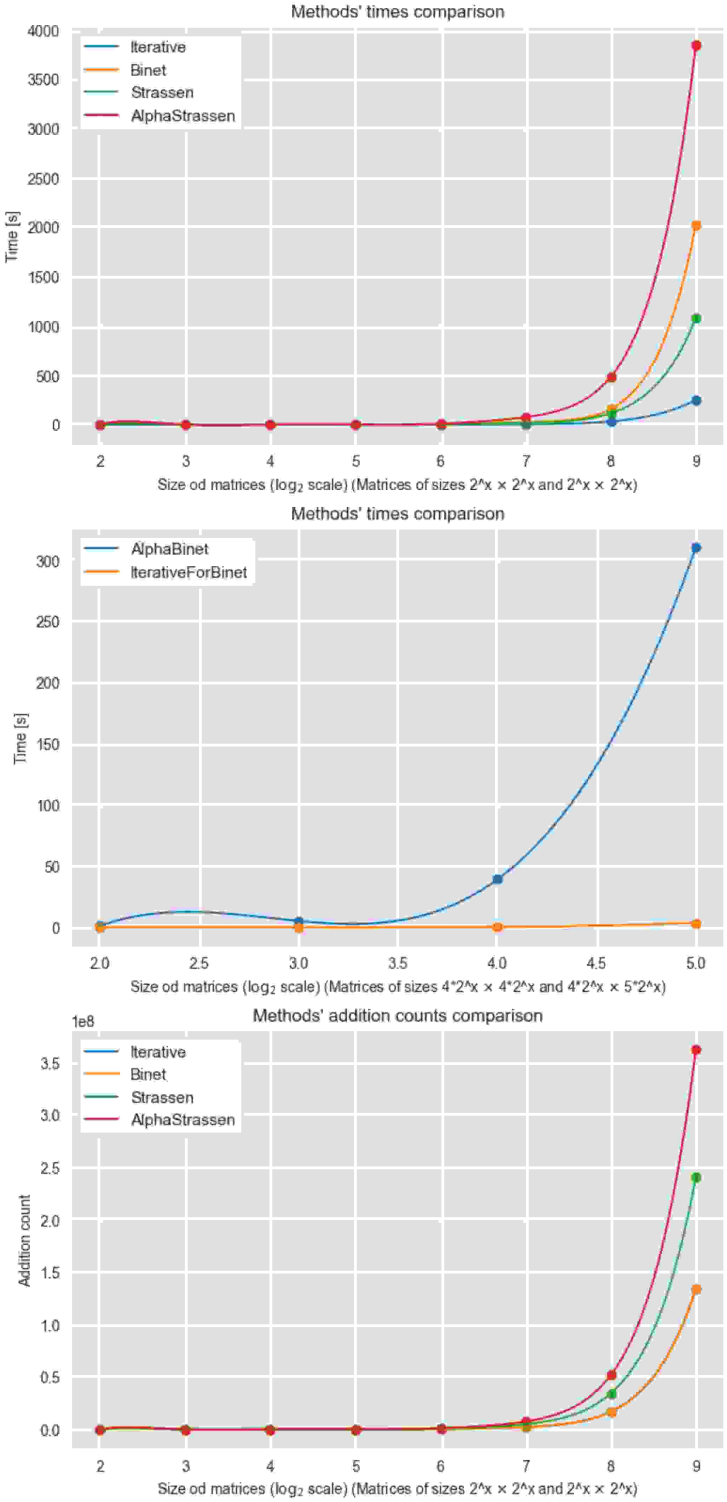
Size (n, m, p)	Best method known	Best rank known	AlphaTensor rank Modular	AlphaTensor rank Standard
(2, 2, 2)	(Strassen, 1969)[2]	7	7	7
(3, 3, 3)	(Laderman, 1976)[15]	23	23	23
(4, 4, 4)	(Strassen, 1969)[2] $(2, 2, 2) \otimes (2, 2, 2)$	49	47	49
(5, 5, 5)	(3, 5, 5) + (2, 5, 5)	98	96	98
(2, 2, 3)	(2, 2, 2) + (2, 2, 1)	11	11	11
(2, 2, 4)	(2, 2, 2) + (2, 2, 2)	14	14	14
(2, 2, 5)	(2, 2, 2) + (2, 2, 3)	18	18	18
(2, 3, 3)	(Hopcroft and Kerr, 1971)[16]	15	15	15
(2, 3, 4)	(Hopcroft and Kerr, 1971)[16]	20	20	20
(2, 3, 5)	(Hopcroft and Kerr, 1971)[16]	25	25	25
(2, 4, 4)	(Hopcroft and Kerr, 1971)[16]	26	26	26
(2, 4, 5)	(Hopcroft and Kerr, 1971)[16]	33	33	33
(2, 5, 5)	(Hopcroft and Kerr, 1971)[16]	40	40	40
(3, 3, 4)	(Smirnov, 2013)[18]	29	29	29
(3, 3, 5)	(Smirnov, 2013)[18]	36	36	36
(3, 4, 4)	(Smirnov, 2013)[18]	38	38	38
(3, 4, 5)	(Smirnov, 2013)[18]	48	47	47
(3, 5, 5)	(Sedoglavic and Smirnov, 2021)[19]	58	58	58
(4, 4, 5)	(4, 4, 2) + (4, 4, 3)	64	63	63
(4, 5, 5)	$(2, 5, 5) \otimes (2, 1, 1)$	80	76	76

Obraz 1: Typy macierzy i ilość mnożeń wymagana do policzenia ich iloczynów

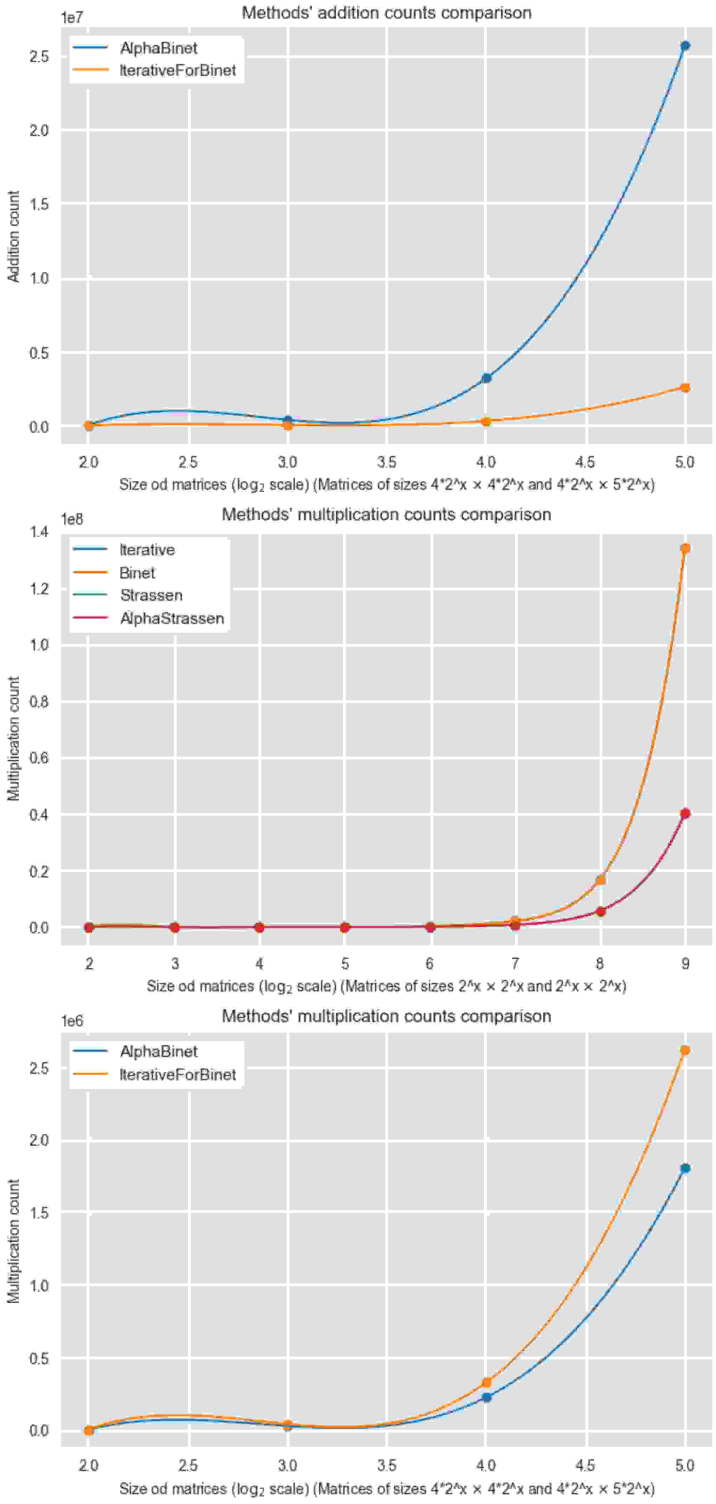
W naszych testach użyliśmy zmodyfikowanego algorytmu Strassena, który iloczyn macierzy (4, 4, 4) oblicza metodą znalezioną przez AI (ale kroki dla większych macierzy wykonuje metodą Strassena). W tych przypadkach algorytm nie będzie działał szybciej od Strassena, gdyż liczba multiplikacji jest taka sama.

Dodatkowo przetestowany zostanie algorytm iloczynu macierzy (4, 4, 5). Został on napisany poprzez modyfikację algorytmu Bineta - gdy zejdziemy do iloczynu macierzy możliwego do obliczenia przy pomocy algorytmu znalezionego przez AI, to wykonujemy taki iloczyn. W tym wypadku oczekujemy, że liczba mnożeń będzie mniejsza niż w przypadkach naiwnego algorytmu.

5 Wykresy



Obraz 2: Wykresy - cz.1



Wykres 3: Wykresy - cz.2

6 Porównanie z Octave

```
In 40  1  MMM = MatrixMultiplier(np.array([[1, 2],[3, 4]]), np.array([[1, 2], [3, 4]]))
       2  MMM.binet()
       3  MMM.result

Out 40
```

	0	1
0	7.0	10.0
1	15.0	22.0

2 rows × 2 columns Open in new tab

Wykres 4: Wynik mnożenia naszą implementacją algorytmu Bineta

```
octave:1> A = [1, 2; 3, 4];
B = [1, 2; 3, 4];

result = A * B;




octave:4> result
result =

    7   10
   15   22
```

Wykres 5: Wynik mnożenia - Octave

	Iterative	Binet	Strassen	AlphaStrassen	AlphaBinet	IterativeForBinet
2	True	True	True	True	True	True
3	True	True	True	True	True	True
4	True	True	True	True	True	True
5	True	True	True	True	True	True
6	True	True	True	True	NaN	NaN
7	True	True	True	True	NaN	NaN
8	True	True	True	True	NaN	NaN
9	True	True	True	True	NaN	NaN

Wykres 6: Sprawdzenie poprawności naszych algorytmów z wynikiem mnożenia macierzy algorytmem z numpy

7 Wnioski

Zgodnie z oczekiwaniami bardziej zaawansowane algorytmy osiągają mniejszą ilość obliczeń multiplikatywnych od tych podstawowych. Algorytm chociaż ma złożoność taką samą jak metoda iteracyjna, to pozwala zmniejszyć liczbę dodawań. Algorytmy zaproponowane przez sztuczną inteligencję potrafią mieć mniej obliczeń względem tych naiwnych, jednak czas ich wykonania jest najdłuższy. Wynika to konieczności wykonania dużej liczby dodatkowych operacji, różnych od dodawań i mnożeń, a także specyfiki samego języka programowania.