



**WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI**
POLITECHNIKI RZESZOWSKIEJ

Filip Walkowicz

Implementacja sieci neuronowej w języku C

Projekt

Opiekun pracy:
mgr inż. Kamil Gomułka

Rzeszów, 2024

Spis treści

1. Wstęp	5
2. Praca z danymi	6
2.1. Importowanie danych uczących z CSV	6
2.2. Normalizacja danych	7
2.3. Podział danych na zbiory uczący, walidacyjny oraz testowy	8
3. Implementacja w kodzie podstawowych funkcji	10
3.1. Tworzenie struktury macierzy	10
3.2. Wyświetlanie macierzy	12
3.3. Dodawanie Macierzy	13
3.4. Mnożenie Macierzy	13
3.5. Funkcja aktywacji ReLU	14
3.6. Tworzenie struktury modelu	15
3.7. Funkcja Kosztu MSE	16
3.8. Goodness of fit - R^2	16
4. Uczenie modelu	17
4.1. Pętla ucząca	17
4.2. Propagacja wsteczna	19
5. Podsumowanie	20
5.1. Problem wybuchającego gradientu	20
5.2. Rozwiązania problemu wybuchających gradientów	20
6. Źródła	21

1. Wstęp

Celem projektu jest implementacja trójwarstwowej gęsto połączonej sieci neuronowej w języku C w celu przewidywania cen złota na podstawie historycznych danych rynkowych. Projekt ten ma na celu stworzenie podstawowej infrastruktury sieci neuronowej, zdolnej do analizy danych finansowych oraz predykcję wartości ceny w przyszłości.

Do trenowania modelu wykorzystany zostanie przykładowy zbiór danych z serwisu kaggle.com, zawierający rzeczywiste wartości cen złota. Dzięki dostępowi do takich danych, możliwe będzie zarówno przeprowadzenie wstępnej analizy, jak i późniejsze oceny skuteczności modelu na podstawie rzeczywistych trendów rynkowych. Zbiór danych zawiera różne wskaźniki, które mają wpływ na wahania ceny złota. Przygotowane dane historyczne, przedstawione w tabeli 1.1, stanowią kluczowy zasób informacyjny, który umożliwi naukę sieci oraz późniejsze testowanie jej efektywności w zadaniu przewidywania cen złota. Proces ten obejmuje analizę korelacji między różnymi cechami danych oraz ich wpływu na wartość ceny zamknięcia.

- Price: Oficjalna cena zamknięcia w danym dniu, miesiącu lub roku.
- Open: Cena otwarcia w danym dniu, miesiącu lub roku.
- High: Najwyższa cena w danym dniu, miesiącu lub roku.
- Low: Najniższa cena w danym dniu, miesiącu lub roku.
- Volume: Wolumen transakcji w danym dniu, miesiącu lub roku.
- Change: Zmiana procentowa pomiędzy ceną z bieżącego i poprzedniego okresu (dnia, miesiąca lub roku).

Tablica 1.1: Przykładowe dane ze zbioru danych ceny złota

Price	Open	High	Low	Change
271.6	271.6	271.6	271.6	0.0727
256.6	256.6	256.6	256.6	-0.0552
262.4	262.4	262.4	262.4	0.0226
290.7	290.7	290.7	290.7	0.1079
293.5	293.5	293.5	293.5	0.0096

2. Praca z danymi

2.1. Importowanie danych uczących z CSV

Pierwszym krokiem w pracy z danymi jest zaimplementowanie funkcji odczytującej zawartość pliku CSV i zapisującej dane w strukturze macierzy. W tym projekcie zaimplementowano funkcję `readData`, która wczytuje kolejne wiersze z pliku CSV, przetwarza je i zapisuje w odpowiednie miejsca w tablicy przechowującej dane w formacie macierzy.

```
1 void readData(Matrix DatasetMatix) {
2     char line[1000];
3     char *token;
4     FILE *file = fopen("<path_to_file>", "r");
5     if (file == NULL) {
6         perror("Error opening file");
7     }
8
9     fgets(line, sizeof(line), file); // skip header
10
11     for (int i = 0; i < DatasetMatix.rows; i++) {
12         fgets(line, sizeof(line), file);
13         token = strtok(line, ",");
14         for (int j = 0; j < DatasetMatix.cols; j++) {
15             DatasetMatix.data[i * DatasetMatix.cols + j] = atof(
16 token);
17             token = strtok(NULL, ",");
18             if (token == NULL) {
19                 break;
20             }
21         }
22         fclose(file);
23 }
```

Listing 1: Import danych z pliku CSV

Na początku w funkcji definiuje trzy podstawowe rzeczy: bufor, token, oraz plik z którego dane mają być zczytane. Bufor przechowuje dane całej linii, a token pojedynczych wartości w linii. Od razu tutaj zabezpieczam się przed niepoprawnym otwarciem pliku z wykorzystaniem `if`.

```
1 char line[1000];
2 char *token;
3 FILE *file = fopen("<path_to_file>", "r");
4 if (file == NULL) {
5     perror("Error opening file");
6 }
```

Listing 2: Definiowanie zmiennych w `readData`

Kolejnym krokiem jest pominięcie nagłówka który nie jest potrzebny oraz zapis danych do odpowiednich komurek w macierzy. Struktura wykorzystywanej macierzy została dokładniej opisana w 3.1. W skrócie przechodzę po każdej komórce i przypisuje kolejne dane z pliku do niej.

```

1 for (int i = 0; i < DatasetMatix.rows; i++) {
2     fgets(line, sizeof(line), file);
3     token = strtok(line, ",");
4     for (int j = 0; j < DatasetMatix.cols; j++) {
5         DatasetMatix.data[i * DatasetMatix.cols + j] = atof(
6         token);
7         token = strtok(NULL, ",");
8         if (token == NULL) {
9             break;
10        }
11    }
12 }

```

Listing 3: Iterowanie po macierzy i przypisywanie danych

2.2. Normalizacja danych

Normalizacja danych jest kluczowym krokiem w procesie przetwarzania danych, ponieważ zapewnia spójny zakres wartości dla różnych cech (np. ceny, wolumenu). Dzięki temu zwiększa się stabilność procesu uczenia maszynowego, a także przyspiesza zbieżność algorytmu, co ma szczególne znaczenie w przypadku optymalizacji metodą gradientową. Jedną z powszechnie stosowanych metod normalizacji jest skalowanie min-max, opisane poniżej:

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}} \quad (2.1)$$

W języku C implementacja powyższej formuły została zrealizowana za pomocą funkcji MinMaxNormalize(), której kod przedstawiono poniżej. Funkcja ta przeprowadza normalizację danych w macierzy, przekształcając każdą kolumnę niezależnie:

```

1 void MinMaxNormalize(Matrix mat) {
2     for (int i = 1; i < mat.cols; i++) {
3         float min = findMin(mat, i);
4         float max = findMax(mat, i);
5         for (int j = 0; j < mat.rows; j++) {
6             mat.data[j * mat.cols + i] = (mat.data[j * mat.cols
7             + i] - min) / (max - min);
8         }
9     }
10 }

```

Listing 4: Normalizacja danych z wykorzystaniem MinMax

Funkcja na początku wyszukuje wartości minimalne i maksymalne dla każdej kolumny za pomocą funkcji pomocniczych findMin() i findMax(), które działają na przekazanej macierzy. Ich kod przedstawiono poniżej:

```
1 float findMax(Matrix mat, int column) {
2     float max = 0;
3     for (int i = 0; i < mat.rows; i++) {
4         if (mat.data[i * mat.cols + column] > max) {
5             max = mat.data[i * mat.cols + column];
6         }
7     }
8
9     return max;
10 }
11
12 float findMin(Matrix mat, int column) {
13     float min = mat.data[mat.cols + column];
14     for (int i = 0; i < mat.rows; i++) {
15         if (mat.data[i * mat.cols + column] < min) {
16             min = mat.data[i * mat.cols + column];
17         }
18     }
19
20     return min;
21 }
```

Listing 5: Wyszukiwanie wartości min/max

Wartości minimalne i maksymalne zwracane przez funkcje findMin() i findMax() są następnie wykorzystywane w implementacji wzoru (2.1), który przekształca dane w danej kolumnie do zakresu [0,1].

2.3. Podział danych na zbiory uczący, walidacyjny oraz testowy

W procesie uczenia maszynowego dane są zazwyczaj dzielone na trzy główne zbiory: uczący, walidacyjny i testowy. Zbiór uczący służy do trenowania modelu, umożliwiając mu naukę na podstawie wzorców i zależności zawartych w danych, podczas gdy zbiór walidacyjny pozwala na ocenę wydajności modelu i dobór odpowiednich hiperparametrów, co pomaga uniknąć przeuczenia. Po zakończeniu treningu i walidacji model jest testowany na zbiorze testowym, który służy do obiektywnej oceny jego zdolności do generalizacji na nowych, nieznanych danych.

W języku C implementacja powyższych zbiorów została zrealizowana za pomocą funkcji CutDataset(), która ma na celu podział podstawowego zbioru na trzy podzbiory. W tej sekcji została stworzona też nowa struktura która reprezentuje podział zbioru na trzy części.


```

1 typedef struct {
2     Matrix trainDataset;
3     Matrix testDataset;
4     Matrix validDataset;
5 } DatasetSplit;

```

Listing 6: Struktura przechowywująca zbiór treningowy - walidacyjny - testowy

CutDataset() dzieli dane na trzy części w oparciu o określony procentowy udział zbioru uczącego. Funkcja tworzy nową strukturę zawierającą trzy macierze. Dane są kopiowane z oryginalnej macierzy do odpowiednich zbiorów, a na końcu zwracana jest struktura zawierająca te trzy zbiory, co pozwala na ich dalsze wykorzystanie w procesie trenowania i oceny modelu.

```

1 DatasetSplit CutDataset(Matrix mat, float trainPercent) {
2     int trainDatasetSize = (int)(mat.rows * trainPercent);
3     int testDatasetSize = (mat.rows - trainDatasetSize) / 2 + 1;
4     int validDatasetSize = (mat.rows - trainDatasetSize) / 2;
5
6     printf("Sizes are: %d-%d-%d\n", trainDatasetSize,
7         testDatasetSize, validDatasetSize);
8     DatasetSplit result;
9     result.trainDataset = createMatrix(trainDatasetSize, mat.
10        cols);
11     result.testDataset = createMatrix(testDatasetSize, mat.cols)
12        ;
13     result.validDataset = createMatrix(validDatasetSize, mat.
14        cols);
15     for (int i = 0; i < trainDatasetSize; i++) {
16         for (int j = 0; j < mat.cols; j++) {
17             result.trainDataset.data[i * mat.cols + j] = mat.
18                data[i * mat.cols + j];
19         }
20     }
21     for (int i = trainDatasetSize; i < trainDatasetSize +
22        testDatasetSize; i++) {
23         for (int j = 0; j < mat.cols; j++) {
24             result.testDataset.data[(i - trainDatasetSize) * mat.
25                .cols + j] = mat.data[i * mat.cols + j];
26         }
27     }
28     for (int i = trainDatasetSize + testDatasetSize; i < mat.
29        rows; i++) {
30         for (int j = 0; j < mat.cols; j++) {
31             result.validDataset.data[(i - trainDatasetSize -
32                testDatasetSize) * mat.cols + j] = mat.data[i * mat.cols + j];
33         }
34     }
35     return result;
36 }

```

Listing 7: Podział na zbiór treningowy - walidacyjny - testowy

3. Implementacja w kodzie podstawowych funkcji

3.1. Tworzenie struktury macierzy

W sieci neuronowej operacje na dużych macierzach są kluczowe, ponieważ służą do przechowywania wag między neuronami, danych wejściowych oraz wyników przetwarzania. W tej sekcji zaimplementowano podstawową strukturę Matrix w języku C, która umożliwia reprezentację tych danych w sposób uporządkowany i umożliwiający łatwe wykonywanie operacji macierzowych.

```
1 typedef struct {  
2     float *data;  
3     int rows;  
4     int cols;  
5 } Matrix;
```

Listing 8: Implementacja struktury macierzy w C

Dzięki takiej strukturze możemy dynamicznie alokować tablice typu float, która przechowuje wszystkie elementy macierzy. Użycie jednowymiarowej tablicy jest wydajniejsze i ułatwia nam zarządzanie pamięcią lecz dodaje nam kilka kroków w implementacji. Ponieważ dane ustawiamy w tablicy jednowymiarowej potrzebne są nam informacje także o ilości wierszy i kolumn w danej macierzy. Dzięki temu jeżeli dane są ustawione w lini w pamięci możemy przeskakiwać pomiędzy wierszami w taki sposób:

```
1 for (int i = 0; i < rows; i++) {  
2     for (int j = 0; j < cols; j++) {  
3         printf("  %f", mat.data[i * mat.cols + j]);  
4     }  
5 }
```

Listing 9: Iterowanie po elementach macierzy

Następnym krokiem jest stworzenie funkcji która pozwoli nam na tworzenie przykładowej macierzy wraz z wypełnieniem jej. Funkcja createMatrix przyjmuje dwa parametry: liczbę wierszy rows oraz liczbę kolumn cols, i zwraca strukturę Matrix, która reprezentuje macierz o określonych rozmiarach. Wewnątrz funkcji:

- 1) Inicjalizacja struktury: Tworzona jest nowa instancja Matrix, a jej pola rows i cols są ustawiane na wartości podane jako argumenty funkcji.
- 2) Alokacja pamięci: Wskaźnik data jest inicjalizowany poprzez dynamiczną alokację pamięci na rows * cols elementów typu float. To pozwala na przechowywanie

wszystkich wartości macierzy w jednej, ciągłej blokowej strukturze pamięci, co ułatwia iterowanie po elementach i zwiększa wydajność.

```
1 Matrix createMatrix(int rows, int cols) {
2     Matrix mat;
3     mat.rows = rows;
4     mat.cols = cols;
5     mat.data = (float *) malloc(rows * cols * sizeof(float));
6
7     for (int i = 0; i < rows; i++) {
8         for (int j = 0; j < cols; j++) {
9             mat.data[i * cols + j] = randFloat();
10        }
11    }
12    return mat;
13 }
```

Listing 10: Funkcja tworząca macierz

W powyższym przykładzie użyłem także funkcji `randFloat()` którą wcześniej zaimplementowałem, a która ma na celu wygenerowanie pseudolosowej liczby z zakresu 0-1:

```
1 float randFloat(void) {
2     return (float) rand() / (float) RAND_MAX;
3 }
```

Listing 11: Generowanie losowej liczby

Jak było wspomniane powyżej, funkcja ma na celu inicjalizację struktury oraz alokację pamięci. Na samym początku następuje inicjalizacja struktury oraz przypisanie ilości kolumn i wierszy. Później rezerwujemy odpowiednią ilość pamięci funkcją `malloc`.

```
1 Matrix mat;
2 mat.rows = rows;
3 mat.cols = cols;
4 mat.data = (float *) malloc(rows * cols * sizeof(float));
```

Listing 12: Inicjalizacja struktury oraz alokacja pamięci

Wartość każdego elementu jest ustawiana poprzez indeksowanie jednowymiarowej tablicy `data` za pomocą formuły $i * \text{cols} + j$. Ta formuła oblicza przesunięcie danego elementu w jednowymiarowej tablicy, co pozwala na indeksowanie elementów w stylu macierzowym (wiersz-kolumna) w jednowymiarowej tablicy.

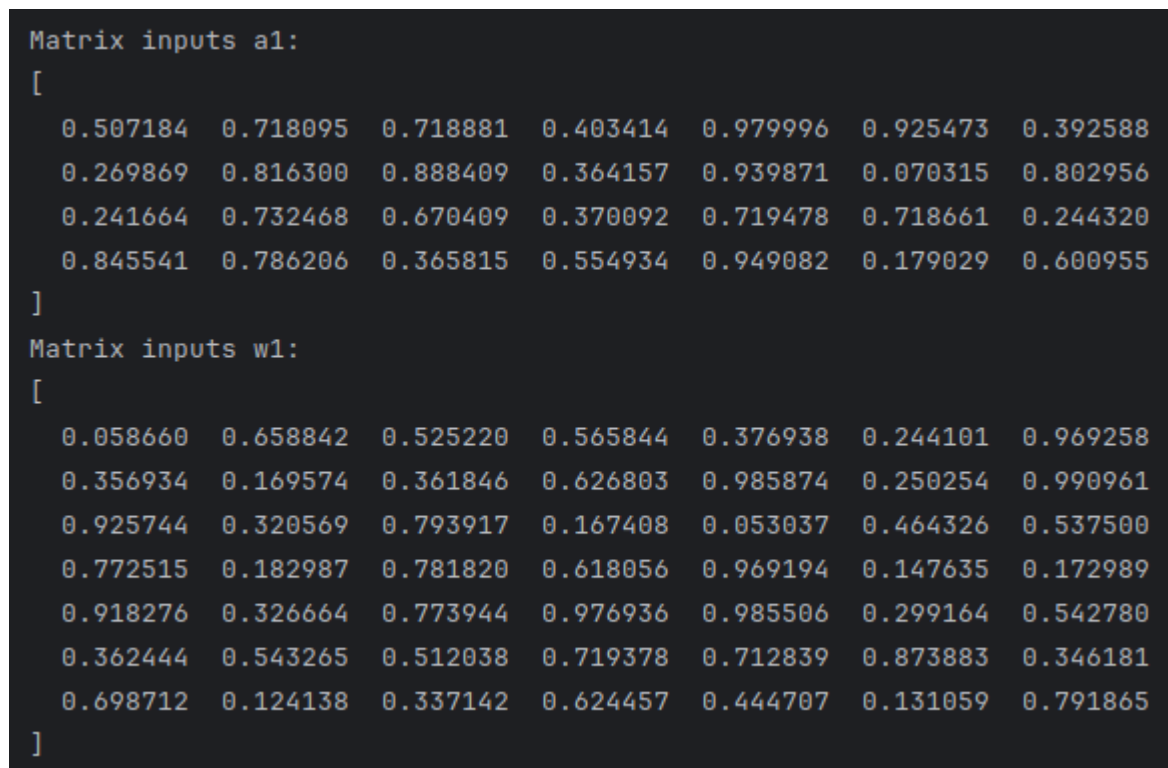
3.2. Wyświetlanie macierzy

W celu ułatwienia pracy na macierzach zaimplementowałem funkcję `displayMatrix` która ma na celu wyświetlenia zawartości macierzy w przejrzysty dla człowieka sposób. Funkcja `displayMatrix` przyjmuje jako argument strukturę `Matrix` i wypisuje jej zawartość w postaci wierszy i kolumn:

```
1 void displayMatrix(Matrix mat) {  
2     printf("[\n");  
3     for (int i = 0; i < mat.rows; i++) {  
4         for (int j = 0; j < mat.cols; j++) {  
5             printf("    %f", mat.data[i * mat.cols + j]);  
6         }  
7         printf("\n");  
8     }  
9     printf("]\n");  
10 }
```

Listing 13: Wyświetlanie macierzy

Funkcja rozpoczyna wyświetlanie macierzy od znaku `[` oraz nowej linii, aby otworzyć reprezentację macierzy i zapewnić czytelność, w ten sam sposób kończy ona wyświetlanie macierzy.



```
Matrix inputs a1:  
[  
    0.507184  0.718095  0.718881  0.403414  0.979996  0.925473  0.392588  
    0.269869  0.816300  0.888409  0.364157  0.939871  0.070315  0.802956  
    0.241664  0.732468  0.670409  0.370092  0.719478  0.718661  0.244320  
    0.845541  0.786206  0.365815  0.554934  0.949082  0.179029  0.600955  
]  
Matrix inputs w1:  
[  
    0.058660  0.658842  0.525220  0.565844  0.376938  0.244101  0.969258  
    0.356934  0.169574  0.361846  0.626803  0.985874  0.250254  0.990961  
    0.925744  0.320569  0.793917  0.167408  0.053037  0.464326  0.537500  
    0.772515  0.182987  0.781820  0.618056  0.969194  0.147635  0.172989  
    0.918276  0.326664  0.773944  0.976936  0.985506  0.299164  0.542780  
    0.362444  0.543265  0.512038  0.719378  0.712839  0.873883  0.346181  
    0.698712  0.124138  0.337142  0.624457  0.444707  0.131059  0.791865  
]
```

Rysunek 3.1: Przykład wywołania funkcji `displayMatrix`

3.3. Dodawanie Macierzy

Dodawanie macierzy jest operacją, w której wartości elementów dwóch macierzy o tych samych wymiarach są dodawane do siebie element po elemencie. Wynikiem tej operacji jest nowa macierz o tych samych wymiarach, której każdy element stanowi sumę odpowiadających sobie elementów z obu macierzy wejściowych. Przykład dodawania dwóch macierzy został zaprezentowany w poniższym przykładzie 3.2

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} + \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11} + B_{11} & A_{12} + B_{12} \\ A_{21} + B_{21} & A_{22} + B_{22} \end{bmatrix} \quad (3.2)$$

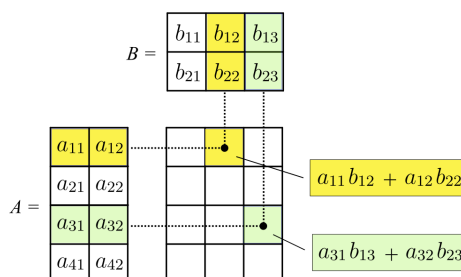
Następnym krokiem jest zaimplementowanie dodawania w języku C. Problem ten został rozwiązany przy implementacji nowej funkcji 15 która przechodzi po każdym elemencie macierzy oraz dodaje wartości z drugiej macierzy do macierzy wynikowej.

```
1 void addMatrix(Matrix result, Matrix mat) {  
2     for (int i = 0; i < result.rows; i++) {  
3         for (int j = 0; j < result.cols; j++) {  
4             result.data[i * result.cols + j] += mat.data[i *  
5                 result.cols + j];  
6         }  
7     }  
}
```

Listing 14: Dodawanie macierzy w C

3.4. Mnożenie Macierzy

Mnożenie macierzy jest operacją, w której wartości elementów dwóch macierzy o tych samych wymiarach wewnętrznych np.(2x3 i 3x4) są mnożone oraz dodawane do siebie w sposób wiersz razy kolumna. Przykład mnożenia dwóch macierzy został zaprezentowany w poniższym przykładzie 5.5



Rysunek 3.2: mnożenia macierzy

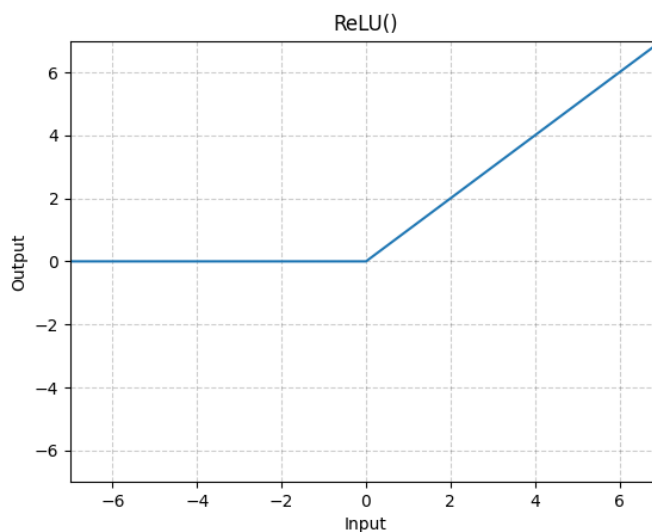
Mnożenie macierzy zostało zaimplementowane w języku C za pomocą stworzenia nowej funkcji "dotMatrix". Funkcja ta iteruje przez wszystkie wiersze macierzy A i kolumny macierzy B, obliczając każdy element macierzy wynikowej jako sumę iloczynów elementów odpowiadających sobie wiersza i kolumny.

```
1 void dotProduct(Matrix mat_a, Matrix mat_b, Matrix result) {  
2     result.rows = mat_a.rows;  
3     result.cols = mat_b.cols;  
4     for (int i = 0; i < result.rows; i++) {  
5         for (int j = 0; j < result.cols; j++) {  
6             for (int k = 0; k < mat_a.cols; k++) {  
7                 result.data[i * result.cols + j] += mat_a.data[i  
8                 * result.cols + k] * mat_b.data[k * result.cols + j];  
9             }  
10        }  
11    }
```

Listing 15: Mnożenie macierzy w C

3.5. Funkcja aktywacji ReLU

Funkcja aktywacji w sieciach neuronowych to element kluczowy, który wprowadza nieliniowość do modelu, co pozwala sieci lepiej uczyć się złożonych zależności w danych. ReLU jest jedną z najczęściej używanych funkcji aktywacji. Jej działanie jest proste: Jeśli wejście jest większe lub równe 0, pozostawia je bez zmian lub ustawia wartość na 0.



Rysunek 3.3: Funkcja aktywacji ReLu

Implementacja funkcji ReLU w C działa na wszystkich elementach macierzy wejściowej, modyfikując je. Kod przechodzi przez każdy element macierzy i sprawdza, czy jego wartość jest mniejsza od 0. Jeśli tak, ustawia ją na 0. W przeciwnym razie pozostawia ją bez zmian.

```
1 void ReLU(Matrix mat) {  
2     for (int i = 0; i < mat.rows; i++) {  
3         for (int j = 0; j < mat.cols; j++) {  
4             if (mat.data[i * mat.cols + j] < 0.0f) {  
5                 mat.data[i * mat.cols + j] = 0.0f;  
6             }  
7         }  
8     }  
9 }
```

Listing 16: Implementacja ReLU w C

3.6. Tworzenie struktury modelu

W tej sekcji przedstawiono sposób implementacji struktury modelu w języku C za pomocą struct. Struktura GoldModel definiuje trójwarstwową sieć neuronową, w której każda warstwa zawiera odpowiednie macierze reprezentujące wejścia, wagi oraz bias'y. Taka reprezentacja modelu pozwala na łatwe przechowywanie i przetwarzanie danych oraz wag w sieci neuronowej. Struktura ułatwia również dostęp do poszczególnych elementów modelu, dzięki czemu funkcje takie jak propagacja wsteczna, obliczanie aktywacji czy aktualizacja wag są prostsze w implementacji.

```
1 typedef struct {  
2     Matrix a0;  
3  
4     Matrix a1;  
5     Matrix w1;  
6     Matrix b1;  
7  
8     Matrix a2;  
9     Matrix w2;  
10    Matrix b2;  
11  
12    Matrix a3;  
13    Matrix w3;  
14    Matrix b3;  
15 } GoldModel;
```

Listing 17: Implementacja struktury modelu w C

3.7. Funckja Kosztu MSE

MSE(Mean Squared Error) - czyli błąd średniokwadratowy będzie idealną funkcją kosztu dla naszego problemu. Jest on jednym z najczęściej używanych funkcji kosztu dla problemów regresji, działa on w bardzo prosty sposób licząc średnią wartość kwadratów różnic między przewidywanymi wartościami a wartościami rzeczywistymi.

$$MSE = \frac{1}{n} \sum_{i=1}^n (y - \hat{y})^2$$

gdzie:

- n - liczba obserwacji
- y - rzeczywista wartość
- \hat{y} - przewidywana wartość

Poniżej przedstawiono implementację funkcji MSE w języku C. Kod zakłada, że dane wejściowe `actual` i `predicted` są macierzami o tej samej liczbie wierszy i kolumn.

```
1 float MSE(Matrix actual, Matrix predicted) {  
2     float mse = 0;  
3     for (int i = 0; i < actual.rows; i++) {  
4         for (int j = 0; j < actual.cols; j++) {  
5             float error = actual.data[i * actual.cols + j] -  
6                 predicted.data[i * predicted.cols + j];  
7             mse += error * error;  
8         }  
9     }  
10    return mse / actual.rows;  
}
```

Listing 18: Implementacja MSE w C

3.8. Goodness of fit - R^2

Wskaźnik R^2 , zwany również współczynnikiem determinacji, jest miarą, która ocenia, jak dobrze model regresji wyjaśnia zmienność danych rzeczywistych. Innymi słowy, R^2 pokazuje, jaka część zmienności wartości rzeczywistych y została przewidziana przez model. Czym wartość R^2 bliżej 1 tym model lepiej przewiduje wartości.

$$R^2 = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2}$$

Poniżej przedstawiono implementację funkcji R^2 w języku C. Kod zakłada, że dane wejściowe `actual` i `predicted` są macierzami o tej samej liczbie wierszy i kolumn.


```

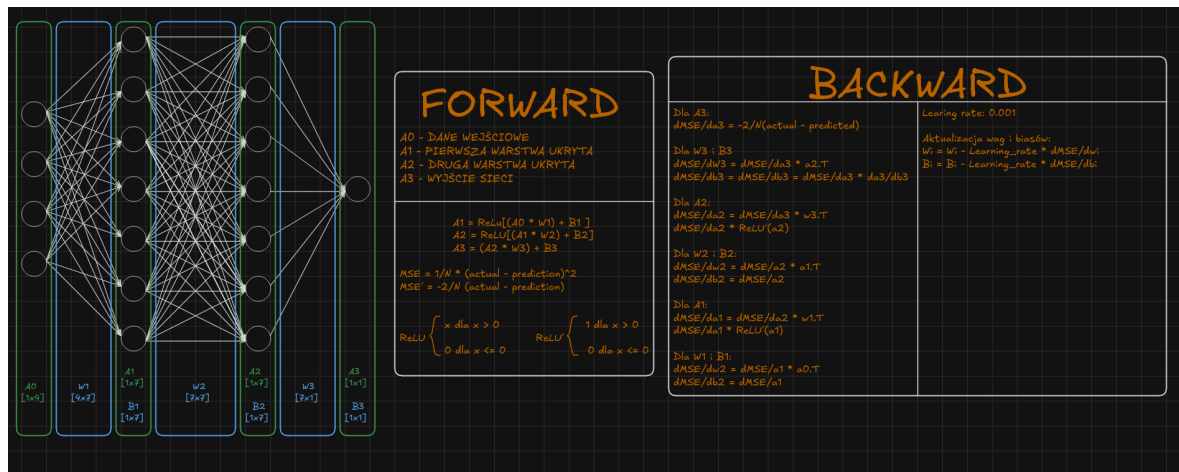
1 float R_squared(Matrix actual, Matrix predicted) {
2     float squared = 0;
3     float rss = 0;
4     float tss = 0;
5     float mean = MeanValue(predicted);
6     for (int i = 0; i < actual.rows; i++) {
7         for (int j = 0; j < actual.cols; j++) {
8             rss += pow((actual.data[i * actual.cols + j] -
9             predicted.data[i * actual.cols + j]), 2);
10            tss += pow((actual.data[i * actual.cols + j] - mean)
11            ,2);
12        }
13    }
14    return 1- (rss / tss);
15 }

```

Listing 19: Implementacja R^2 w C

4. Uczenie modelu

Uczenie modelu opiera się na propagacji w przód (ang. forward propagation) i wstecznej propagacji błędów (ang. backward propagation). W kodzie te dwie fazy odpowiadają za odpowiednio obliczenie predykcji modelu oraz aktualizację wag w celu zminimalizowania błędów predykcji na podstawie przewidzianej wartości.



Rysunek 4.4: Opis modelu oraz funkcji forward, backward

4.1. Pętla ucząca

Pętla ucząca to proces, w którym model przechodzi przez dane treningowe wiele razy, aby nauczyć się rozwiązywać postawiony problem. W każdej epoce model najpierw przechodzi przez fazę treningu, gdzie dane wejściowe są przekazywane przez sieć,

a następnie obliczane są predykcje. Na podstawie tych predykcji i prawdziwych wartości model aktualizuje swoje parametry, takie jak wagi i biasy, wykorzystując funkcję backward.

Po fazie treningu następuje faza walidacji, w której model sprawdza swoje predykcje na oddzielnym zbiorze walidacyjnym, nieaktualizując już parametrów. Obliczane są metryki, takie jak MSE (błąd średniokwadratowy) i R^2 (współczynnik determinacji), aby ocenić, jak dobrze model radzi sobie zarówno na danych treningowych, jak i walidacyjnych. W każdej epoce wyniki są wyświetlane, co pozwala monitorować postępy modelu i jego zdolność do poprawy w czasie.

```
1  const int epochs = x;
2  for (int epoch = 0; epoch < epochs; epoch++) {
3      // Training phase
4      for (int i = 0; i < Dataset.trainDatasetX.rows; i++) {
5          memset(model.a0.data, 0, sizeof(float) * model.a0.cols);
6          for (int j = 0; j < Dataset.trainDatasetX.cols; j++) {
7              model.a0.data[j] = Dataset.trainDatasetX.data[i *
Dataset.trainDatasetX.cols + j];
8          }
9          predictions.TrainPredictions.data[i] = forward(model);
10     }
11     backward(model, predictions.TrainPredictions, Dataset,
0.00000001);
12     // Validation phase
13     for (int i = 0; i < Dataset.validDatasetX.rows; i++) {
14         memset(model.a0.data, 0, sizeof(float) * model.a0.cols);
15         for (int j = 0; j < Dataset.validDatasetX.cols; j++) {
16             model.a0.data[j] = Dataset.validDatasetX.data[i *
Dataset.validDatasetX.cols + j];
17         }
18         predictions.ValidPredictions.data[i] = forward(model);
19     }
20     float TrainMse = MSE(Dataset.trainDatasetY, predictions.
TrainPredictions);
21     float ValidMse = MSE(Dataset.validDatasetY, predictions.
ValidPredictions);
22     float RSquaredTrain = R_squared(Dataset.trainDatasetY,
predictions.TrainPredictions);
23     float RSquaredValid = R_squared(Dataset.validDatasetY,
predictions.ValidPredictions);
24     printf("Epoch %d/%d %d/%d [=====] - ms
/step - MSE loss: %.03f, R^2: %f  val_loss : %.02f\n",
epoch + 1, epochs, Dataset.trainDatasetX.rows,
predictions.TrainPredictions.rows, TrainMse,
RSquaredTrain);
25
26
27
28
29 }
```

Listing 20: Implementacja pętli uczącej w C

4.2. Propagacja wsteczna

W tej implementacji propagacji wstecznej obliczamy gradienty dla każdej warstwy modelu, zaczynając od wyjścia sieci. Wykorzystujemy je do aktualizacji wag i biasów w celu minimalizacji funkcji kosztu. Gradienty są propagowane wstecz, warstwa po warstwie, aż do wejścia sieci. Obliczenia uwzględniają pochodne funkcji aktywacji oraz różnicę między przewidywanymi a rzeczywistymi wartościami celu.

```
1 void backward(GoldModel model, Matrix predictions, DatasetSplit
  dataset_split, float learningRate) {
2     float dMSEda3 = 0;
3     for (int i = 0; i < predictions.rows; i++) {
4         for (int j = 0; j < predictions.cols; j++) {
5             dMSEda3 += (-2.0 / (float) dataset_split.
trainDatasetY.rows) * (
6                 dataset_split.trainDatasetY.data[i] -
predictions.data[i]);
7         }
8     }
9
10    // dMSE/db3 = dMSE/da3 * da3/db3 = dMSE/da3 * 1
11    float dMSEdb3 = dMSEda3;
12
13    // dMSE/dw3 = dMSE/da3 * da3/da2 * da2/dw3 = dMSEda3 * a2.T
14    Matrix dMSEdw3 = createMatrix(model.w3.rows, model.w3.cols);
15    Matrix transposedA2 = transposeMatrix(model.a2);
16    for (int i = 0; i < dMSEdw3.rows; i++) {
17        for (int j = 0; j < dMSEdw3.cols; j++) {
18            dMSEdw3.data[i * dMSEdw3.cols + j] = transposedA2.
data[i * dMSEdw3.cols + j];
19        }
20    }
21    printf("dMSEdw3: %d x %d, transposedA2: %d x %d\n",
22        dMSEdw3.rows, dMSEdw3.cols, transposedA2.rows, transposedA2.
cols);
23    // dMSE/da2 = dMSE/a3 * w3.T
24    // Gradient propagowany do A2 -> dMSE/da2
25    Matrix dMSEda2 = copyMatrix(model.w3);
26    Matrix dMSEda2T = transposeMatrix(dMSEda2);
27    multipleMatrixByValue(dMSEda2T, dMSEda3);
28
29    Matrix da2 = copyMatrix(model.a2);
30    derivativeReLU(da2);
31    Matrix dReLUda2 = createMatrix(model.a2.rows, model.a2.cols)
;
32    dotProduct(dMSEda2T, da2, dReLUda2);
33
34 }
```

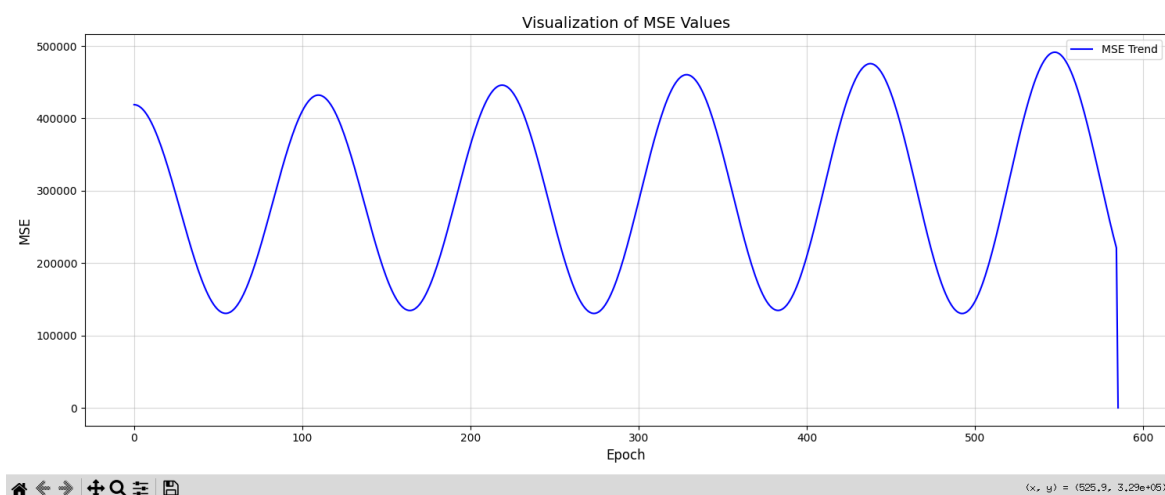
Listing 21: Implementacja propagacji wstecznej błędu w C

5. Podsumowanie

Celem tego projektu było stworzenie i zaimplementowanie podstawowej sieci neuronowej w języku C. Model jest trenowany przy użyciu algorytmu optymalizacji gradient descent, a jego wydajność monitorowana jest przez metryki takie jak błąd średniokwadratowy (MSE) i współczynnik determinacji (R^2).

5.1. Problem wybuchającego gradientu

Jednym z napotkanych problemów w implementacji sieci neuronowej jest tzw. eksplozja gradientów (exploding gradients). Problem ten występuje, gdy gradienty podczas propagacji wstecznej stają się zbyt duże, co prowadzi do niestabilności procesu uczenia, a w efekcie do niepoprawnych wartości wag. W przypadku dużych gradientów, zmiany wag mogą być zbyt gwałtowne, co powoduje, że model przestaje uczyć się w sposób efektywny, a wartości wag stają się nieprzewidywalne.



Rysunek 5.5: Gradient w czasie

5.2. Rozwiązania problemu wybuchających gradientów

Aby rozwiązać problem wybuchających gradientów, można zastosować kilka technik:

- **Batch Normalization:** Batch normalization to technika, która normalizuje dane wejściowe do każdej warstwy w trakcie treningu. Dzięki temu, wartości

wejściowe dla każdej warstwy są skalowane i przesunięte w sposób, który zapobiega nadmiernym wartościom gradientów.

- **Inicjalizacja wag:** Odpowiednia inicjalizacja wag może pomóc w uniknięciu eksplozji gradientów.
- **Użycie funkcji aktywacji z ograniczeniem gradientu:** Zamiast używać funkcji ReLU, można wypróbować inne funkcje aktywacji, takie jak Leaky ReLU lub ELU.
- **Techniki przycinania gradientów (Gradient Clipping):** Przycinanie gradientów polega na ograniczeniu wartości gradientu, jeżeli przekroczy ona określoną wartość progową.

6. Źródła

- [How To Read a CSV File in C](#)
- [Gradient Descent + MSE from Scratch](#)
- [Propagacja wsteczna cz.1](#)
- [Propagacja wsteczna cz.2](#)
- [Propagacja wsteczna cz.3](#)