Table of Contents

0. Wstęp

Niniejszy artykuł - **Sieci neuronowe (C#)** - ma na celu zilustrowanie podstawowych pojęć z zakresu sieci neuronowych za pomocą przykładów napisanych w C#. Równolegle, część z tu zaprezentowanych programów ma swoje odpowiedniki przeniesione do Delphi Object Pascala.

Dla kogo jest ten artykuł? Przede wszystkim dla programistów C#, którzy chcą zrozumieć, jak działają sieci neuronowe "od środka", bez korzystania z gotowych bibliotek i frameworków. Artykuł zakłada podstawową znajomość programowania w C# oraz podstaw matematyki (algebra liniowa, rachunek różniczkowy). I to w zasadzie tyle - reszta wiedzy jest przekazywana w trakcie lektury.

Pełne kody źródłowe dostępne są w repozytorium na GitHub.

Dostępna jest również wersja w formacie PDF.

Spis treści

Artykuł został podzielony na następujące rozdziały:

Przedstawiono strukturę i funkcjonalność biblioteki NeuralNetworks (C#), służącej do definiowania i trenowania modeli sieci neuronowych oraz do przeprowadzania procesu wnioskowania (inferencji) z użyciem tych modeli. Omówiono sposób definiowania architektury sieci, funkcji aktywacji, funkcji straty oraz metod optymalizacji dostępnych w bibliotece. Przedstawiono wzory matematyczne i implementacje podstawowych elementów, takich jak funkcje aktywacji, funkcje straty oraz algorytmy optymalizacji. Zaprezentowano również przykład użycia tej biblioteki do utworzenia modelu trenowanego na danych Boston Housing.

- 6. Dane MNIST

Zaprezentowano dwie sieci neuronowe (sieć z warstwami gęstymi oraz sieć konwolucyjną) trenowane na zbiorze danych MNIST do rozpoznawania ręcznie pisanych cyfr.

Reszta *in progress*...

Created: 2025-11-10

Last modified: 2025-12-19

Title: **0. Wstęp**

Tags: [C#] [Object Pascal] [Delphi] [Sieci neuronowe] [Regresja liniowa]

1. Prosta regresja liniowa

Omawianie sieci neuronowych w kontekście języka C# rozpoczniemy od *funkcji liniowej* i *regresji liniowej*.

> **NOTE**
> Dlaczego zaczynamy od regresji liniowej? Dlatego, że regresja liniowa może być traktowana jak sieć neuronowa o jednej warstwie z jednym neuronem o liniowej funkcji aktywacji.

1.1. Co to takiego?

Funkcja liniowa opisuje zależność między zmiennymi za pomocą wyrażenia liniowego $y = a_1x_1 + a_2x_2 + \cdots + a_nx_n + b$, a **regresja liniowa** wykorzystuje tę zależność do modelowania i przewidywania wartości jednej zmiennej na podstawie innej lub wielu innych zmiennych.

- Zmienne $x_1, x_2, \ldots, x_n$ nazywane są zmiennymi **niezależnymi**, a zmienna y - zmienną **zależną**.
- Współczynniki (albo **współczynniki kierunkowe**) $a_1, a_2, \ldots, a_n$ określają stopień wpływu poszczególnych zmiennych niezależnych na zmienną zależną y (im większa jest bezwzględna wartość współczynnika kierunkowego, tym większy wpływ; współczynnik równy zeru oznacza brak wpływu).
- Parametr b jest nazywany **wyrazem wolnym**.

Przyjmuje się, że **prosta regresja liniowa** (ang. *simple linear regression*) to model regresji liniowej z *jedną* zmienną niezależną (czyli z jednym współczynnikiem kierunkowym a i wyrazem wolnym b), a **wieloraka regresja liniowa** (lub *wielokrotna regresja liniowa*, ang. *multiple linear regression*) to model regresji liniowej z *wieloma* zmiennymi niezależnymi (czyli z wieloma współczynnikami kierunkowymi $a_1, a_2, \ldots, a_n$ i wyrazem wolnym b).

Więcej na ten temat można przeczytać w Wikipedii: Regresja liniowa i Funkcja liniowa.

1.2. Metoda najmniejszych kwadratów

Istnieje wiele metod służących do budowania modelu regresji liniowej. Najprostszą z nich jest metoda najmniejszych kwadratów, która ma zamknięte rozwiązanie analityczne.

Dla funkcji z jedną zmienną niezależną w postaci $y = ax + b$ możemy obliczyć współczynnik kierunkowy:

$$a = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{1.1}$$

oraz wyraz wolny:

$$b = \bar{y} - a\bar{x} \tag{1.2}$$

gdzie

- (x_i, y_i) – kolejne punkty danych (obserwacje),
- $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ – średnia z x,
- $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$ – średnia z y,
- n – liczba obserwacji.

1.2.1. Przykład liczbowy

Przykładowe dane dotyczące sprzedaży jakiegoś produktu w zależności od jego ceny (dane treningowe) przedstawia poniższa tabela.

Obserwacja (i)	Cena [zł] (x)	Sprzedaż [szt] (y)
1	10	100
2	20	80
3	30	60
4	40	40
5	50	20

Tabela 1.1. Dane treningowe: cena/sprzedaż

Najpierw obliczamy wartości średnie:

$$\bar{x} = \frac{10 + 20 + 30 + 40 + 50}{5} = 30 \tag{1.3}$$

$$\bar{y} = \frac{100 + 80 + 60 + 40 + 20}{5} = 60 \tag{1.4}$$

Następnie obliczamy współczynnik kierunkowy a. Zgodnie na wzorem (1.1) jego licznik wynosi:

$$(10-30)(100-60) + (20-30)(80-60) + (30-30)(60-60) \\ + (40-30)(40-60) + (50-30)(20-60) = -2000$$

a mianownik:

$$(10-30)^2 + (20-30)^2 + (30-30)^2 \\ + (40-30)^2 + (50-30)^2 = 1000$$

Współczynnik a przyjmuje więc wartość:

$$a = \frac{-2000}{1000} = -2 \tag{1.5}$$

Na koniec obliczamy wyraz wolny b:

$$b = \bar{y} - a\bar{x} = 60 - (-2) \cdot 30 = 60 + 60 = 120 \tag{1.6}$$

W ten sposób otrzymujemy następujący wzór na poszukiwaną funkcję:

$$y = -2x + 120 \tag{1.7}$$

Proste, prawda? 🤓

1.3. Metoda spadku gradientowego

W kontekście uczenia maszynowego bardziej interesującym algorytmem jest metoda **spadku gradientowego** (zwana również metodą *spadku gradientu*, ang. *gradient descent*, *GD*), która polega na iteracyjnej aktualizacji parametrów szukanej funkcji (modelu regresji).

> **NOTE**
> Angielski termin *gradient descent* jest często tłumaczony jako *spadek gradientu*, co wydaje się być mylące. W metodzie tej nie chodzi bowiem o to, że sam gradient "spada" - jego wartość, w zależności od geometrycznego kształtu funkcji straty, może nawet w kolejnych iteracjach rosnąć - ale mimo to taka nazwa ogólnie się przyjęła. Wg mnie lepszym terminem jest właśnie *spadek gradientowy* (a nawet *schodzenie gradientowe*).

Zaczynamy od wyboru wartości losowych dla $a_1, a_2, \ldots, a_n$ i b (choć możemy po prostu wstępnie ustawić je na 0), a następnie iteracyjnie zwiększamy lub zmniejszamy te wartości, tak aby zminimalizować tzw. **funkcję straty**.

Funkcja straty (ang. *loss function*, zwana również *funkcją kosztu*) mierzy to, jak dobrze model dopasowuje się do danych treningowych. Im *mniejsza* wartość tej funkcji, tym *lepsze* dopasowanie (ale bez przesady - zbyt dobre dopasowanie, czyli tzw. **overfitting**, też jest niepożądane). W przypadku regresji liniowej funkcją straty jest zazwyczaj **MSE** (czyli *błąd średniokwadratowy*, ang. *Mean Square Error*). Inne stosowane funkcje straty to **SSE** (*Sum of Squared Errors*) i **RMSE** (*Root Mean Square Error*).

Błąd średniokwadratowy jest zdefiniowany jako:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{1.8}$$

gdzie:

- n – liczba obserwacji,
- $\hat{y}_i = a_1x_{i1} + a_2x_{i2} + \cdots + a_nx_{in} + b$ – *przewidywana* wartość zmiennej zależnej dla obserwacji i (parametry $a_1, a_2, \ldots, a_n, b$ nie zależą od indeksu i),
- y_i – *rzeczywista* wartość zmiennej zależnej.

> **NOTE**
> Nie należy mylić $\hat{y}$ z $\bar{y}$. To pierwsze to wartość przewidywana, to drugie to wartość średnia.

Aby znaleźć optymalne wartości parametrów modelu, współczynniki $a_1, a_2, \ldots, a_n$ (oraz wyraz wolny b) są iteracyjnie aktualizowane w następujący sposób:

$$a_j := a_j - lr \cdot \frac{\partial}{\partial a_j} MSE \tag{1.9}$$

gdzie:

- a_j – współczynniki funkcji liniowej (dla $j = 1, 2, \ldots$),
- lr - współczynnik uczenia (o nim poniżej),
- $\frac{\partial}{\partial a_j} MSE$ – pochodna funkcji straty względem współczynnika a_j.

> **NOTE**
> Symbol $:=$ to operator przypisania (czyli "przypisz wartość z prawej strony do zmiennej po lewej stronie"). Będziemy go stosować dla odróżnienia od znaku równości $=$ używanego we wzorach matematycznych.

Wzór na pochodną $\frac{\partial}{\partial a_j} MSE$ względem a_j to:

$$\frac{\partial}{\partial a_j} MSE = \frac{2}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i) x_{ij} \tag{1.10}$$

a więc ostatecznie aktualizacja współczynnika a_j wygląda tak:

$$a_j := a_j - lr \cdot \frac{2}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i) x_{ij} \tag{1.11}$$

Wyraz wolny b aktualizowany jest w podobny sposób:

$$b := b - lr \cdot \frac{\partial}{\partial b} MSE \tag{1.12}$$

gdzie pochodna $\frac{\partial}{\partial b} MSE$ względem b jest równa:

$$\frac{\partial}{\partial b} MSE = \frac{2}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i) \tag{1.13}$$

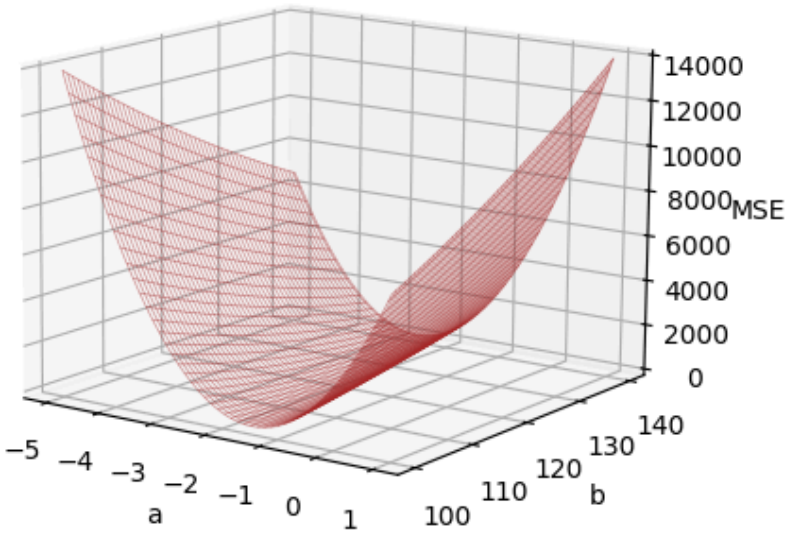
Jak widać, wzory na powyższe pochodne są podobne do siebie, z tym że w przypadku a_j uwzględniają dodatkowo zmienną niezależną x_{ij}, a w przypadku b - nie.

1.3.1. Po co w ogóle jest nam potrzebna pochodna?

Dlaczego zmiany parametrów regresji w kolejnych iteracjach są proporcjonalne (współczynnikiem proporcjonalności jest tu współczynnik uczenia lr) do **pochodnej** funkcji straty względem tych parametrów? Dlatego, że pochodna mówi nam o tym, w którą stronę mamy zmierzać ze zmianami parametrów a i b, tak aby funkcja straty malała.

Albo inaczej: **gradient** (czyli wektor pochodnych cząstkowych) wskazuje nam kierunek najszybszego *wzrostu* funkcji straty. A ponieważ szukamy drogi, która poprowadzi nas do *spadku* tej funkcji, więc poruszamy się w kierunku przeciwnym (stąd minus we wzorze $a_j := a_j - lr \cdot \mathbf{pochodna}$).

Dla danych z tabeli 1.1 wykres funkcji straty, po której się poruszamy wygląda tak:



Wykres 1.1. Funkcja straty MSE w zależności od współczynników regresji liniowej a i b

Jej minimum przypada na punkt $(a = -2, b = 120)$. Współczynniki te odpowiadają funkcji regresji liniowej $y = -2x + 120$ z równania (1.7).

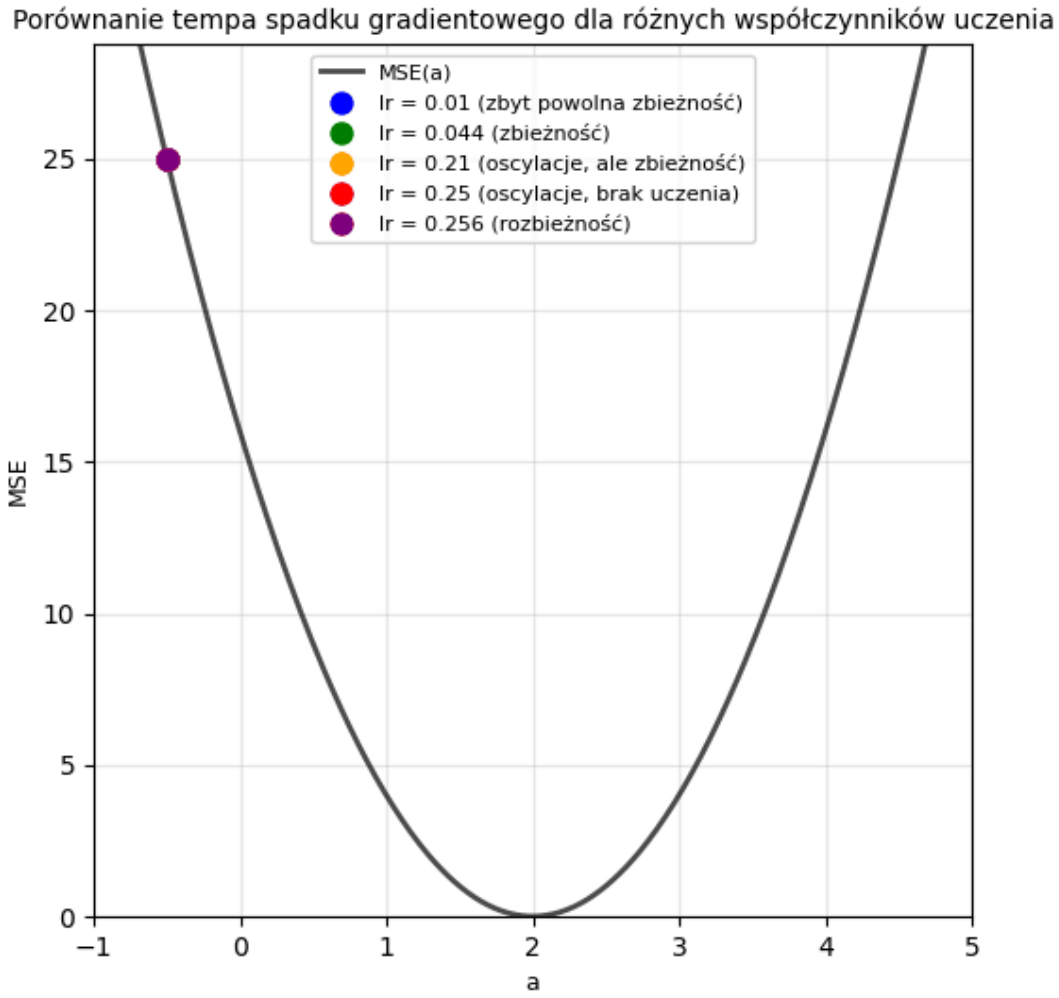
A więc: zaczynamy naszą zgadywankę na przykład od punktu $(a = 0, b = 0)$ (punkt startowy dobry jak każdy inny). Otrzymujemy dla niego jakąś - większą od zera - wartość MSE. Ponieważ interesuje nas najmniejsze możliwe do osiągnięcia MSE (najlepiej równe 0), to obliczamy pochodną (gradient) w tym właśnie punkcie $(a = 0, b = 0)$, która to pochodna wskazuje nam kierunek, w którym *rośnie* MSE. Gdy zmienimy znak wartości tej pochodnej (minus na początku wzoru: $-lr \cdot \frac{\partial}{\partial a_j} MSE$), będzie ona wskazywać kierunek, w którym MSE *maleje*.

W ten sposób możemy zaktualizować współczynniki regresji, aby zbliżyć się do minimum funkcji straty. Super 👍.

1.3.2. Współczynnik uczenia

Współczynnik uczenia (lr, ang. *learning rate*) to hiperparametr, który kontroluje wielkość kroku, jaki wykonujemy w kierunku minimum MSE w każdej iteracji. Jest wspólny dla wszystkich parametrów modelu i zawsze większy od 0. Zbyt *duża* wartość lr może prowadzić do niestabilności i oscylowania wokół minimum funkcji straty (model uczy się gorzej), podczas gdy zbyt *mała* wartość może spowodować zbyt wolne zbieganie do minimum (model uczy się wolniej).

Poniżej przedstawiono animację ilustrującą wpływ różnych wartości współczynnika uczenia (lr = 0,01; 0,044; 0,21; 0,25; 0,256) na tempo spadku wartości funkcji straty w trakcie uczenia modelu regresji liniowej metodą spadku gradientowego.



Wykres 1.2. Porównanie tempa spadku gradientowego dla różnych współczynników uczenia. Mamy tutaj daną $(x = 2, y = 4)$ *oraz funkcję* $y = a \cdot x + 0$ *, a więc szukanym parametrem jest w tym przypadku* $a = 2$*. Wartość początkowa, od której rozpoczynamy poszukiwanie, wynosi tu* $a = -0.5$

1.3.3. Przykład liczbowy

Zastosujemy teraz metodę spadku gradientowego dla danych z poprzedniego przykładu (cena/sprzedaż), aby znaleźć współczynniki regresji liniowej. Zaczniemy od wartości początkowych $a = 0$ i $b = 0$, a współczynnik uczenia ustawimy na $lr = 0.0005$.

Przeprowadźmy obliczenia dla pierwszych dwóch iteracji.

W każdej iteracji obliczamy:

- przewidywane wartości $\hat{y}_i$,
- wartości błędu ($\hat{y}_i - y_i$),
- pochodną względem współczynnika a, tj. $\frac{\partial}{\partial a} MSE = \frac{2}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i) x_i$ oraz
- pochodną względem współczynnika b, tj. $\frac{\partial}{\partial b} MSE = \frac{2}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)$.

1.3.3.1. Pierwsza iteracja

W poniższej tabeli umieszczono wyniki obliczeń dla pierwszej iteracji. Przewidywane wartości $\hat{y}_i$ są równe zeru, ponieważ zaczynamy od wartości początkowych $a = 0$ i $b = 0$.

Cena [zł] (x)	Sprzedaż [szt] (y)	Przewidywana sprzedaż ($\hat{y} = ax + b$)	Błąd ($\hat{y} - y$)
10	100	0 * 10 + 0 = 0	0 - 100 = -100
20	80	0 * 20 + 0 = 0	0 - 80 = -80
30	60	0 * 30 + 0 = 0	0 - 60 = -60
40	40	0 * 40 + 0 = 0	0 - 40 = -40
50	20	0 * 50 + 0 = 0	0 - 20 = -20

Tabela 1.2. Obliczenia dla pierwszej iteracji

Dla pierwszej iteracji:

- MSE (*funkcja straty*) wynosi:

$$
\begin{aligned}
MSE &= \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 \\
&= \frac{1}{5} \left((-100)^2 + (-80)^2 + (-60)^2 + (-40)^2 + (-20)^2 \right) \\
&= \frac{22000}{5} = 4400
\end{aligned}
\tag{1.14}
$$

- Pochodna względem a wynosi:

$$\begin{aligned}\frac{\partial}{\partial a}MSE &= \frac{2}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)x_i \\ &= \frac{2}{5}((-100)\cdot 10 + (-80)\cdot 20 + (-60)\cdot 30 + (-40)\cdot 40 + (-20)\cdot 50) \\ &= \frac{2}{5}(-7000) = -2800\end{aligned} \tag{1.15}$$

- Pochodna względem b wynosi:

$$\begin{aligned}\frac{\partial}{\partial b}MSE &= \frac{2}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i) \\ &= \frac{2}{5}((-100) + (-80) + (-60) + (-40) + (-20)) \\ &= \frac{2}{5}(-300) = -120\end{aligned} \tag{1.16}$$

- Parametry modelu po aktualizacji przyjmują wartości:

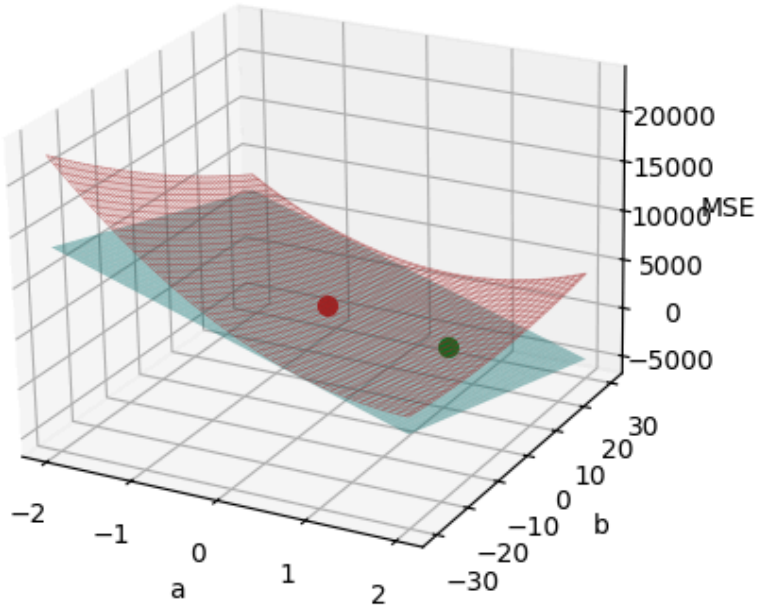
$$a := a - lr \cdot \frac{\partial}{\partial a}MSE = 0 - 0.0005 \cdot (-2800) = 1.4 \tag{1.17}$$

$$b := b - lr \cdot \frac{\partial}{\partial b}MSE = 0 - 0.0005 \cdot (-120) = 0.06 \tag{1.18}$$

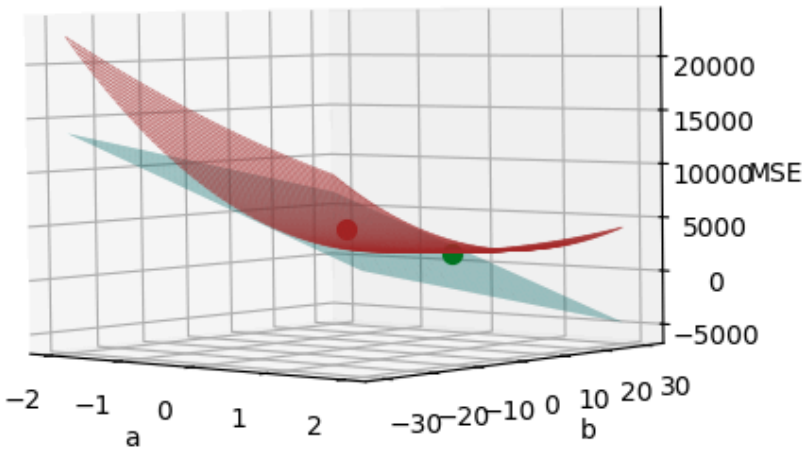
Płaszczyzna ilustrująca pochodną MSE w punkcie $(a = 0, b = 0)$ ma wzór $z = -2800a - 120b + 4400$. Poszczególne jej parametry wynikają z równań (1.14), (1.15) i (1.16).

Na poniższych wykresach (oba przedstawiają to samo, tylko z nieco innej perspektywy) oznaczono:

- punktem czerwonym - punkt styczności "płaszczyzny pochodnej" (kolor cyjanowy) z wykresem MSE (kolor brązowy),
- punktem zielonym - współrzędne $(a = 1.4, b = 0.06)$, które otrzymaliśmy po pierwszej iteracji (wzory (1.17) i (1.18)), poruszając się wzdłuż kierunku spadku wartości MSE, czyli jakby tocząc się w dół po "cyjanowej płaszczyźnie".



Wykres 1.3. Funkcja straty i płaszczyzna pochodnej w punkcie (a=0, b=0)



Wykres 1.4. Funkcja straty i płaszczyzna pochodnej w punkcie (a=0, b=0) - inna perspektywa

Na marginesie, zauważmy, że wykresy 1.1 i 1.2 (i oczywiście 1.3) przedstawiają tę samą funkcję straty MSE. Różnica w kształcie wynika z innych zakresów wartości współczynników a i b na osiach poziomych. Wykres 1.1 pokazuje funkcję straty w zakresie $a \in [-5, 1]$ i $b \in [100, 140]$ (tam znajduje się jej minimum), a wykresy 1.2 i 1.3 - w zakresie $a \in [-2, 2]$ i $b \in [-30, 30]$ (tam znajduje się startowy punkt iteracji $(a = 0, b = 0)$).

1.3.3.2. Druga iteracja

W drugiej iteracji powtarzamy powyższe kroki, przy czym teraz używamy już nowych wartości współczynników $a = 1.4$ i $b = 0.06$.

Tabela z obliczonymi błędami wygląda następująco:

Cena [zł] (x)	Sprzedaż [szt] (y)	Przewidywana sprzedaż ($\hat{y} = ax + b$)	Błąd ($\hat{y} - y$)
10	100	1.4 * 10 + 0.06 = 14.06	14.06 - 100 = -85.94
20	80	1.4 * 20 + 0.06 = 28.06	28.06 - 80 = -51.94
30	60	1.4 * 30 + 0.06 = 42.06	42.06 - 60 = -17.94
40	40	1.4 * 40 + 0.06 = 56.06	56.06 - 40 = 16.06
50	20	1.4 * 50 + 0.06 = 70.06	70.06 - 20 = 50.06

Tabela 1.3. Obliczenia dla drugiej iteracji

I podobnie jak wyżej, dla drugiej iteracji:

- MSE wynosi:

$$\begin{aligned} MSE &= \frac{1}{5}\left((-85.94)^2 + (-51.94)^2 + (-17.94)^2 + 16.06^2 + 50.06^2\right) \\ &= 2633.84375 \end{aligned} \tag{1.19}$$

- Pochodna względem a wynosi:

$$\begin{aligned} \frac{\partial}{\partial a} MSE &= \frac{2}{5}((-85.94) \cdot 10 + (-51.94) \cdot 20 + (-17.94) \cdot 30 + 16.06 \cdot 40 + 50.06 \cdot 50) \\ &= 283.6002 \end{aligned} \tag{1.20}$$

- Pochodna względem b wynosi:

$$\begin{aligned} \frac{\partial}{\partial b} MSE &= \frac{2}{5}(-85.94 - 51.94 - 17.94 + 16.06 + 50.06) \\ &= -35.8800 \end{aligned} \tag{1.21}$$

- Parametry modelu po aktualizacji przyjmują wartości:

$$a := a - lr \cdot \frac{\partial}{\partial a} MSE = 1.4 - 0.0005 \cdot 283.6002 = 1.2582 \tag{1.22}$$

$$b := b - lr \cdot \frac{\partial}{\partial b} MSE = 0.06 - 0.0005 \cdot (-35.88) = 0.0779 \tag{1.23}$$

1.3.3.3. Kolejne iteracje

Iteracje powtarzamy do momentu, aż wartości funkcji straty przestaną się znacząco zmieniać lub osiągniemy założone maksimum liczby iteracji.

Przykładowe wartości dla 4 pierwszych iteracji pokazane są na poniższym rysunku:

```
Iteration:     1 | MSE: 4400,00000 | ∂MSE/∂a: -2800,0000 | ∂MSE/∂b:  -120,0000 | a:    1,4000 | b:    0,0600
Iteration:     2 | MSE: 2633,84375 | ∂MSE/∂a:   283,6002 | ∂MSE/∂b:   -35,8800 | a:    1,2582 | b:    0,0779
Iteration:     3 | MSE: 2614,95093 | ∂MSE/∂a:   -27,2836 | ∂MSE/∂b:   -44,3521 | a:    1,2718 | b:    0,1001
Iteration:     4 | MSE: 2613,81860 | ∂MSE/∂a:     4,0589 | ∂MSE/∂b:   -43,4893 | a:    1,2698 | b:    0,1219

Learned parameters:  a =    1,2698 | b =    0,1219
Expected parameters: a =   -2,0000 | b =  120,0000
```

Rysunek 1.1. Wartości parametrów regresji liniowej i MSE w kolejnych iteracjach

1.3.4. Przykład implementacji

Poniżej znajduje się przykład implementacji w C# regresji liniowej przy użyciu metody spadku gradientowego. Kod źródłowy znajduje na GitHub.

Stałe `LearningRate`, `Iterations` i `PrintEvery` odpowiadają kolejno za współczynnik uczenia (5e-4), liczbę iteracji (35 tysięcy) oraz częstotliwość wypisywania informacji o postępach na konsolę (co 1 tysiąc). Liczbę iteracji można ustawić na 4 a `PrintEvery` na 1, aby wyświetlić dane takie jak na rysunku 1.1.

```
Console.OutputEncoding = System.Text.Encoding.UTF8;

// 1. Set the parameters for the model

const float LearningRate = 0.0005f;
const int Iterations = 35_000; // 4
const int PrintEvery = 1_000; // 1

// 2. Prepare training data

float[,] data = {
    { 10, 100 },
    { 20, 80 },
    { 30, 60 },
    { 40, 40 },
    { 50, 20 },
};

// 3. Initialize model
```

```
float a = 0, b = 0;

// Number of samples
int n = data.GetLength(0);

// 4. Training loop

for (int iteration = 1; iteration <= Iterations; iteration++)
{
    // Initialize accumulators for errors
    float sumErrorValue = 0, sumError = 0, squaredError = 0;

    // For each sample in data
    for (int row = 0; row < n; row++)
    {
        float x = data[row, 0];
        float y = data[row, 1];

        // Prediction and error calculation
        float prediction = a * x + b;
        float error = prediction - y;

        // Accumulate squared error and parts needed for gradient calculation
        squaredError += error * error;
        sumErrorValue += error * x;
        sumError += error;
    }

    // Calculate gradients (partial derivatives of MSE)
    float deltaA = 2.0f / n * sumErrorValue;
    float deltaB = 2.0f / n * sumError;

    // Update regression parameters
    a -= LearningRate * deltaA;
    b -= LearningRate * deltaB;

    if (iteration % PrintEvery == 0)
    {
        // MSE
        float meanSquaredError = squaredError / n;

        Console.WriteLine($"Iteration: {iteration,5} | MSE: {meanSquaredError,10:F5} | ∂MSE/∂a:
{deltaA,10:F4} | ∂MSE/∂b: {deltaB,10:F4} | a: {a,9:F4} | b: {b,9:F4}");
    }
}

// 5. Output learned parameters

Console.WriteLine();
```

```
Console.WriteLine($"{"Learned parameters:",-20} a = {a,9:F4} | b = {b,9:F4}");
Console.WriteLine($"{"Expected parameters:",-20} a = {-2,9:F4} | b = {120,9:F4}");
Console.ReadLine();
```

Listing 1.1. Implementacja regresji liniowej w C# przy użyciu metody spadku gradientowego

Na rysunku 1.2 możemy zobaczyć, że po 35 tysiącach iteracji wartości współczynników regresji liniowej ($a = -1.9943$ i $b = 119.7917$) zbliżyły się do oczekiwanych wartości ($a = -2$ i $b = 120$).

```
Iteration: 10000 | MSE:   69,13258 | ∂MSE/∂a:     0,1930 | ∂MSE/∂b:    -7,0855 | a:   -1,4682 | b:  100,5041
Iteration: 11000 | MSE:   48,06855 | ∂MSE/∂a:     0,1610 | ∂MSE/∂b:    -5,9082 | a:   -1,5566 | b:  103,7433
Iteration: 12000 | MSE:   33,42237 | ∂MSE/∂a:     0,1344 | ∂MSE/∂b:    -4,9266 | a:   -1,6302 | b:  106,4443
Iteration: 13000 | MSE:   23,23884 | ∂MSE/∂a:     0,1120 | ∂MSE/∂b:    -4,1080 | a:   -1,6917 | b:  108,6966
Iteration: 14000 | MSE:   16,15813 | ∂MSE/∂a:     0,0936 | ∂MSE/∂b:    -3,4255 | a:   -1,7429 | b:  110,5746
Iteration: 15000 | MSE:   11,23496 | ∂MSE/∂a:     0,0775 | ∂MSE/∂b:    -2,8564 | a:   -1,7856 | b:  112,1406
Iteration: 16000 | MSE:    7,81183 | ∂MSE/∂a:     0,0650 | ∂MSE/∂b:    -2,3818 | a:   -1,8212 | b:  113,4464
Iteration: 17000 | MSE:    5,43162 | ∂MSE/∂a:     0,0543 | ∂MSE/∂b:    -1,9860 | a:   -1,8509 | b:  114,5353
Iteration: 18000 | MSE:    3,77660 | ∂MSE/∂a:     0,0452 | ∂MSE/∂b:    -1,6561 | a:   -1,8757 | b:  115,4433
Iteration: 19000 | MSE:    2,62591 | ∂MSE/∂a:     0,0379 | ∂MSE/∂b:    -1,3809 | a:   -1,8964 | b:  116,2004
Iteration: 20000 | MSE:    1,82582 | ∂MSE/∂a:     0,0313 | ∂MSE/∂b:    -1,1515 | a:   -1,9136 | b:  116,8317
Iteration: 21000 | MSE:    1,26958 | ∂MSE/∂a:     0,0261 | ∂MSE/∂b:    -0,9602 | a:   -1,9279 | b:  117,3580
Iteration: 22000 | MSE:    0,88270 | ∂MSE/∂a:     0,0217 | ∂MSE/∂b:    -0,8006 | a:   -1,9399 | b:  117,7970
Iteration: 23000 | MSE:    0,61380 | ∂MSE/∂a:     0,0183 | ∂MSE/∂b:    -0,6676 | a:   -1,9499 | b:  118,1630
Iteration: 24000 | MSE:    0,42674 | ∂MSE/∂a:     0,0149 | ∂MSE/∂b:    -0,5567 | a:   -1,9582 | b:  118,4683
Iteration: 25000 | MSE:    0,29672 | ∂MSE/∂a:     0,0125 | ∂MSE/∂b:    -0,4642 | a:   -1,9652 | b:  118,7227
Iteration: 26000 | MSE:    0,20633 | ∂MSE/∂a:     0,0103 | ∂MSE/∂b:    -0,3871 | a:   -1,9709 | b:  118,9349
Iteration: 27000 | MSE:    0,14350 | ∂MSE/∂a:     0,0088 | ∂MSE/∂b:    -0,3228 | a:   -1,9758 | b:  119,1118
Iteration: 28000 | MSE:    0,09976 | ∂MSE/∂a:     0,0076 | ∂MSE/∂b:    -0,2691 | a:   -1,9798 | b:  119,2594
Iteration: 29000 | MSE:    0,06938 | ∂MSE/∂a:     0,0062 | ∂MSE/∂b:    -0,2245 | a:   -1,9832 | b:  119,3824
Iteration: 30000 | MSE:    0,04825 | ∂MSE/∂a:     0,0049 | ∂MSE/∂b:    -0,1872 | a:   -1,9860 | b:  119,4849
Iteration: 31000 | MSE:    0,03357 | ∂MSE/∂a:     0,0041 | ∂MSE/∂b:    -0,1561 | a:   -1,9883 | b:  119,5704
Iteration: 32000 | MSE:    0,02329 | ∂MSE/∂a:     0,0037 | ∂MSE/∂b:    -0,1301 | a:   -1,9902 | b:  119,6421
Iteration: 33000 | MSE:    0,01626 | ∂MSE/∂a:     0,0028 | ∂MSE/∂b:    -0,1087 | a:   -1,9918 | b:  119,7010
Iteration: 34000 | MSE:    0,01132 | ∂MSE/∂a:     0,0025 | ∂MSE/∂b:    -0,0907 | a:   -1,9932 | b:  119,7505
Iteration: 35000 | MSE:    0,00789 | ∂MSE/∂a:     0,0020 | ∂MSE/∂b:    -0,0757 | a:   -1,9943 | b:  119,7917

Learned parameters:  a =   -1,9943 | b =  119,7917
Expected parameters: a =   -2,0000 | b =  120,0000
```

Rysunek 1.2. Wartości parametrów regresji liniowej i MSE po 35 tysiącach iteracji

> **NOTE**
>
> Port powyższego kodu do Delphi Object Pascal można znaleźć tutaj.

1.4. Podsumowanie

W tym rozdziale omówiliśmy podstawy regresji liniowej, w tym metodę spadku gradientowego, którą wykorzystamy w kolejnych rozdziałach podczas uczenia sieci neuronowych.

1.5. Dodatek

1.5.1. Zupa z gwoździa (czyli wyprowadzanie wzorów)

Poniżej, dla kompletności opisu, przedstawiono wyprowadzenia wzorów na pochodne funkcji MSE względem współczynników a (wzór 1.10) i b (wzór 1.13):

$$
\begin{aligned}
\frac{\partial}{\partial a} MSE &= \frac{\partial}{\partial a} \left(\frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 \right) \\
&= \frac{1}{n} \sum_{i=1}^{n} \frac{\partial}{\partial a} (\hat{y}_i - y_i)^2 \\
&= \frac{1}{n} \sum_{i=1}^{n} \frac{\partial}{\partial a} (ax_i + b - y_i)^2 \\
&= \frac{1}{n} \sum_{i=1}^{n} 2(ax_i + b - y_i) \cdot \frac{\partial}{\partial a} (ax_i + b - y_i) \\
&= \frac{1}{n} \sum_{i=1}^{n} 2(\hat{y}_i - y_i) \cdot x_i \\
&= \frac{2}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i) x_i
\end{aligned}
\tag{1.24}
$$

$$
\begin{aligned}
\frac{\partial}{\partial b} MSE &= \frac{\partial}{\partial b} \left(\frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 \right) \\
&= \frac{1}{n} \sum_{i=1}^{n} \frac{\partial}{\partial b} (\hat{y}_i - y_i)^2 \\
&= \frac{1}{n} \sum_{i=1}^{n} \frac{\partial}{\partial b} (ax_i + b - y_i)^2 \\
&= \frac{1}{n} \sum_{i=1}^{n} 2(ax_i + b - y_i) \cdot \frac{\partial}{\partial b} (ax_i + b - y_i) \\
&= \frac{1}{n} \sum_{i=1}^{n} 2(\hat{y}_i - y_i) \cdot 1 \\
&= \frac{2}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)
\end{aligned}
\tag{1.25}
$$

1.5.2. Skrypty do odtworzenia wykresów

Cały niniejszy artykuł jest ilustrowany implementacją w C# (taki zresztą jest jeden z jego celów), ale akurat w przypadku wykresów najprostsze i najszybsze jest skorzystanie z Pythona. Poniżej znajduje się kod przydatny do odtworzenia wykresów z tego rozdziału.

Dla wykresu 1.1:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Training data: y = -2 * x + 120

x = np.array([10, 20, 30, 40, 50])
```

```
y_true = -2 * x + 120

# Grid

a_vals = np.linspace(-5, 1, 100)
b_vals = np.linspace(100, 140, 100)

A, B = np.meshgrid(a_vals, b_vals)

# MSE calculation

MSE = np.mean((A * x[:, None, None] + B - y_true[:, None, None]) ** 2, axis=0)

# Chart

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Wireframe (net) only:

ax.plot_wireframe(A, B, MSE, color='brown', linewidth=0.5, alpha=0.5)
ax.set_xlabel('a')
ax.set_ylabel('b')
ax.set_zlabel('MSE')

plt.show()
```

Listing 1.2

Dla wykresów 1.3 i 1.4:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Training data: y = -2 * x + 120

x = np.array([10, 20, 30, 40, 50])
y_true = -2 * x + 120

# Grid

a_vals = np.linspace(-2, 2, 100)
b_vals = np.linspace(-30, 30, 100)

A, B = np.meshgrid(a_vals, b_vals)

# MSE calculation
```

```
MSE = np.mean((A * x[:, None, None] + B - y_true[:, None, None]) ** 2, axis=0)

# Chart

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Wireframe

ax.plot_wireframe(A, B, MSE, color='brown', linewidth=0.5, alpha=0.5)
ax.set_xlabel('a')
ax.set_ylabel('b')
ax.set_zlabel('MSE')

# Adding a tangent plane at point (a0=0, b0=0)
# Plane equation: Z = grad_a * (A - a0) + grad_b * (B - b0) + MSE0
# MSE0 is the MSE at point (a0, b0)

grad_a = -2800
grad_b = -120
a0, b0 = 0, 0
MSE0 = np.mean((a0 * x + b0 - y_true) ** 2)
tangent_plane = grad_a * (A - a0) + grad_b * (B - b0) + MSE0

# Drawing the tangent plane

ax.plot_surface(A, B, tangent_plane, color='cyan', alpha=0.4)

# Add the red point (0, 0, MSE0)

ax.scatter(a0, b0, MSE0, color='red', s=50, label='Punkt (0, 0, MSE0)')

# Add the green point (1.4, 0.06, MSE1)

a1, b1 = 1.4, 0.06
MSE1 = np.mean((a1 * x + b1 - y_true) ** 2)
ax.scatter(a1, b1, MSE1, color='green', s=50, label='Punkt (1.4, 0.06, MSE1)')

# Printing the gradients and MSE values

print(f"Gradient a: {grad_a}, Gradient b: {grad_b}, MSE at (0, 0): {MSE0}")
print(f"MSE at (1.4, 0.06): {MSE1}")

plt.show()
```

Listing 1.3

See you next time! 👋

Created: 2025-06-19

Last modified: 2025-12-22

Title: **1. Prosta regresja liniowa**

Tags: [C#] [Python] [Sieci neuronowe] [Regresja liniowa] [Funkcja liniowa]

2. Po co nam macierze?

W poprzednim rozdziale zajmowaliśmy się prostą regresją liniową. Teraz rozbudujemy nieco przykład tam umieszczony i dodamy kilka dodatkowych zmiennych niezależnych.

2.1. Przykład z trzema zmiennymi niezależnymi (wieloraka regresja liniowa)

Załóżmy, że mamy następujący zbiór danych:

x_1	x_2	x_3	y
1	2	1	12
2	1	2	10
3	3	1	19
4	2	3	16
1	4	2	17

Tabela 2.1. Zbiór danych do regresji liniowej z trzema zmiennymi niezależnymi

Model regresji liniowej będzie wówczas wyglądał następująco:

$$y = a_1 \cdot x_1 + a_2 \cdot x_2 + a_3 \cdot x_3 + b$$

Wprowadźmy teraz zmiany w listingu 1.1 z poprzedniego rozdziału, tak aby uwzględnić powyższą modyfikację. W miejsce programistycznej zmiennej `a` wprowadzimy zmienne `a1`, `a2` i `a3` oraz zmodyfikujemy kod tak, aby można było na ich podstawie obliczyć gradient.

> **NOTE**
> Używam sformułowania "zmienna programistyczna", aby odróżnić je od zmiennych matematycznych (niezależnych i zależnych) używanych w opisie modelu.

Po wspomnianych wyżej zmianach kod, który implementuje teraz wieloraką regresję liniową, wygląda następująco:

```
// Set the hyperparameters for the model
```

```
const float LearningRate = 0.0005f;
const int Iterations = 35_000;
const int PrintEvery = 1_000;

// Prepare training data
// Each inner array represents a sample: [x1, x2, x3, y]
// We will try to find the relationship: y = 2*x1 + 3*x2 - 1*x3 + 5

float[,] data = new float[,] {
    {1, 2, 1, 12}, // y = 2*1 + 3*2 - 1*1 + 5 = 12
    {2, 1, 2, 10}, // etc.
    {3, 3, 1, 19},
    {4, 2, 3, 16},
    {1, 4, 2, 17}
};

// 1. Initialize model parameters

// Coefficients for the independent variables (x1, x2, x3) and the bias term
float a1 = 0, a2 = 0, a3 = 0;
float b = 0;

// Number of samples
int n = data.GetLength(0);

// 2. Training loop

for (int iteration = 1; iteration <= Iterations; iteration++)
{
    // Initialize accumulators for errors and gradients for this iteration
    float sumSquaredError = 0;
    float sumErrorForA1 = 0;
    float sumErrorForA2 = 0;
    float sumErrorForA3 = 0;
    float sumErrorForB = 0;

    // For each sample in the data
    for (int row = 0; row < n; row++)
    {
        // Get the independent variables (features) and the dependent variable (target)
        float x1 = data[row, 0];
        float x2 = data[row, 1];
        float x3 = data[row, 2];
        float y = data[row, 3];

        // Prediction and error calculation
```

```
        float prediction = a1 * x1 + a2 * x2 + a3 * x3 + b;
        float error = prediction - y;

        // Accumulate squared error for MSE calculation
        sumSquaredError += error * error;

        // Accumulate parts needed for gradient calculation
        sumErrorForA1 += error * x1;
        sumErrorForA2 += error * x2;
        sumErrorForA3 += error * x3;
        sumErrorForB += error;
    }

    // Calculate gradients (partial derivatives of MSE)
    // ∂MSE/∂a1 = 2/n * Σ(error * x1)
    float deltaA1 = 2.0f / n * sumErrorForA1;
    float deltaA2 = 2.0f / n * sumErrorForA2;
    float deltaA3 = 2.0f / n * sumErrorForA3;
    float deltaB = 2.0f / n * sumErrorForB;

    // Update regression parameters using gradient descent
    a1 -= LearningRate * deltaA1;
    a2 -= LearningRate * deltaA2;
    a3 -= LearningRate * deltaA3;
    b -= LearningRate * deltaB;

    if (iteration % PrintEvery == 0)
    {
        // MSE
        float meanSquaredError = sumSquaredError / n;

        Console.WriteLine($"Iteration: {iteration,6} | MSE: {meanSquaredError,8:F5} | a1:
{a1,8:F4} | a2: {a2,8:F4} | a3: {a3,8:F4} | b: {b,8:F4}");
    }
}

// 3. Output learned parameters

Console.WriteLine("\n--- Training Complete (Variables) ---");
Console.WriteLine($"{"Learned parameters:",-20} a1 = {a1,9:F4} | a2 = {a2,9:F4} | a3 =
{a3,9:F4} | b = {b,9:F4}");
Console.WriteLine($"{"Expected parameters:",-20} a1 = {2,9:F4} | a2 = {3,9:F4} | a3 =
{-1,9:F4} | b = {5,9:F4}");
```

Listing 2.1. Wieloraka regresja liniowa z trzema zmiennymi niezależnymi

> **NOTE**
>
> Kod źródłowy przykładów zawartych w niniejszym rozdziale znajduje się na GitHub.

Efekt działania programu z listingu 2.1 przedstawiony został na poniższej ilustracji:

```
Iteration:  13000 | MSE:  0,24071 | a1:   2,1404 | a2:   3,3621 | a3:  -0,6236 | b:   3,0312
Iteration:  14000 | MSE:  0,21444 | a1:   2,1343 | a2:   3,3424 | a3:  -0,6475 | b:   3,1413
Iteration:  15000 | MSE:  0,19105 | a1:   2,1278 | a2:   3,3236 | a3:  -0,6691 | b:   3,2454
Iteration:  16000 | MSE:  0,17021 | a1:   2,1214 | a2:   3,3057 | a3:  -0,6890 | b:   3,3437
Iteration:  17000 | MSE:  0,15165 | a1:   2,1152 | a2:   3,2888 | a3:  -0,7073 | b:   3,4365
Iteration:  18000 | MSE:  0,13511 | a1:   2,1091 | a2:   3,2727 | a3:  -0,7243 | b:   3,5241
Iteration:  19000 | MSE:  0,12038 | a1:   2,1032 | a2:   3,2575 | a3:  -0,7402 | b:   3,6069
Iteration:  20000 | MSE:  0,10726 | a1:   2,0976 | a2:   3,2431 | a3:  -0,7550 | b:   3,6850
Iteration:  21000 | MSE:  0,09556 | a1:   2,0922 | a2:   3,2295 | a3:  -0,7690 | b:   3,7587
Iteration:  22000 | MSE:  0,08515 | a1:   2,0871 | a2:   3,2167 | a3:  -0,7820 | b:   3,8283
Iteration:  23000 | MSE:  0,07586 | a1:   2,0823 | a2:   3,2045 | a3:  -0,7944 | b:   3,8940
Iteration:  24000 | MSE:  0,06759 | a1:   2,0777 | a2:   3,1931 | a3:  -0,8060 | b:   3,9560
Iteration:  25000 | MSE:  0,06022 | a1:   2,0734 | a2:   3,1823 | a3:  -0,8169 | b:   4,0146
Iteration:  26000 | MSE:  0,05366 | a1:   2,0693 | a2:   3,1720 | a3:  -0,8272 | b:   4,0699
Iteration:  27000 | MSE:  0,04781 | a1:   2,0654 | a2:   3,1624 | a3:  -0,8369 | b:   4,1220
Iteration:  28000 | MSE:  0,04260 | a1:   2,0618 | a2:   3,1533 | a3:  -0,8460 | b:   4,1712
Iteration:  29000 | MSE:  0,03795 | a1:   2,0583 | a2:   3,1447 | a3:  -0,8547 | b:   4,2177
Iteration:  30000 | MSE:  0,03382 | a1:   2,0550 | a2:   3,1366 | a3:  -0,8628 | b:   4,2616
Iteration:  31000 | MSE:  0,03013 | a1:   2,0519 | a2:   3,1290 | a3:  -0,8705 | b:   4,3030
Iteration:  32000 | MSE:  0,02685 | a1:   2,0491 | a2:   3,1217 | a3:  -0,8778 | b:   4,3421
Iteration:  33000 | MSE:  0,02392 | a1:   2,0463 | a2:   3,1149 | a3:  -0,8847 | b:   4,3790
Iteration:  34000 | MSE:  0,02131 | a1:   2,0437 | a2:   3,1085 | a3:  -0,8911 | b:   4,4138
Iteration:  35000 | MSE:  0,01899 | a1:   2,0413 | a2:   3,1024 | a3:  -0,8972 | b:   4,4466

--- Training Complete (Variables) ---
Learned parameters:  a1 =    2,0413 | a2 =    3,1024 | a3 =   -0,8972 | b =    4,4466
Expected parameters: a1 =    2,0000 | a2 =    3,0000 | a3 =   -1,0000 | b =    5,0000
```

Rysunek 2.1. Wyniki regresji liniowej z trzema zmiennymi niezależnymi

Widzimy, że po wykonaniu założonej liczby iteracji (`Iterations = 35_000`) wyliczone zostały parametry regresji liniowej a_1, a_2, a_3 oraz b. Są one zbliżone do oczekiwanych wartości $a_1 = 2$, $a_2 = 3$, $a_3 = -1$ oraz $b = 5$.

2.2. Tablice zamiast pojedynczych zmiennych

Na listingu 2.1 pojawiły się kolejne zmienne, które odpowiadają za obliczanie predykcji, błędów i gradientów. Zamiast `x`, `a`, `sumErrorValue` i `deltaA` mamy teraz `x1`, `x2`, `x3`, `a1`, `a2`, `a3`, `sumErrorForA1`, `sumErrorForA2`, `sumErrorForA3`, `deltaA1`, `deltaA2` i `deltaA3`. Jak łatwo można się domyśleć, przy każdym nowym współczynniku musielibyśmy dodawać kolejne odpowiadające mu zmienne do kodu.

Aby tego uniknąć, możemy użyć tablic. Wówczas zamiast operowania na każdej trójce zmiennych programistycznych, np. `a1`, `a2`, `a3`, będziemy operować na pojedynczych tablicach.

Propozycja takiego rozwiązania jest następująca (pominięto części wspólne - stałe i inne deklaracje - zamieszczone w poprzednim listingu):

```
// 1. Initialize model parameters

// Number of samples and coefficients
int n = data.GetLength(0);
int numCoefficients = data.GetLength(1) - 1;

// Coefficients for the independent variables and the bias term
float[] a = new float[numCoefficients];
float b = 0;

// 2. Training loop

for (int iteration = 1; iteration <= Iterations; iteration++)
{
    // Initialize accumulators for errors and gradients for this iteration
    float sumSquaredError = 0;
    float[] sumErrorForA = new float[numCoefficients]; // Accumulator for each coefficient's
gradient part
    float sumErrorForB = 0; // Accumulator for the bias's gradient part

    // For each sample in the data
    for (int row = 0; row < n; row++)
    {
        // Separate the independent variables (features) (x) from the dependent variable
(target) (y)
        float[] x = new float[numCoefficients];
        for (int i = 0; i < numCoefficients; i++)
        {
            x[i] = data[row, i];
        }
        float y = data[row, numCoefficients];

        // Prediction and error calculation
        // prediction = a1*x1 + a2*x2 + a3*x3 + b
        float prediction = b;
        for (int i = 0; i < numCoefficients; i++)
        {
            prediction += a[i] * x[i];
        }
        float error = prediction - y;

        // Accumulate squared error for MSE calculation
```

```
        sumSquaredError += error * error;

        // Accumulate parts needed for gradient calculation
        // For each ai, the gradient part is (error * xi)
        for (int i = 0; i < numCoefficients; i++)
        {
            sumErrorForA[i] += error * x[i];
        }
        // For the bias, the gradient part is just the error
        sumErrorForB += error;
    }

    // Calculate gradients (partial derivatives of MSE)
    // ∂MSE/∂ai = 2/n * Σ(error * xi)
    float[] deltaA = new float[numCoefficients];
    for (int i = 0; i < numCoefficients; i++)
    {
        deltaA[i] = 2.0f / n * sumErrorForA[i];
    }

    // ∂MSE/∂b = 2/n * Σ(error)
    float deltaB = 2.0f / n * sumErrorForB;

    // Update regression parameters using gradient descent
    for (int i = 0; i < numCoefficients; i++)
    {
        a[i] -= LearningRate * deltaA[i];
    }
    b -= LearningRate * deltaB;

    if (iteration % PrintEvery == 0)
    {
        // MSE
        float meanSquaredError = sumSquaredError / n;

        Console.WriteLine($"Iteration: {iteration,6} | MSE: {meanSquaredError,8:F5} | a1:
{a[0],8:F4} | a2: {a[1],8:F4} | a3: {a[2],8:F4} | b: {b,8:F4}");
    }
}

// 3. Output learned parameters

Console.WriteLine("\n--- Training Complete (Tables) ---");
Console.WriteLine($"{"Learned parameters:",-20} a1 = {a[0],9:F4} | a2 = {a[1],9:F4} | a3 =
{a[2],9:F4} | b = {b,9:F4}");
```

```
Console.WriteLine($"{"Expected parameters:",-20} a1 = {2,9:F4} | a2 = {3,9:F4} | a3 = {-1,9:F4} | b = {5,9:F4}");
```

Listing 2.2. Wieloraka regresja liniowa z trzema zmiennymi niezależnymi - wersja z wykorzystaniem tablic

Efekt działania tego programu (rysunek 2.2) jest taki sam jak poprzednio (rysunek 2.1), ale kod jest krótszy, no i oczywiście bardziej elastyczny. Wystarczy zmienić liczbę kolumn w tablicy `data`, aby - w miarę potrzeb - dodać lub usunąć zmienne niezależne.

```
Iteration:  13000 | MSE:  0,24071 | a1:   2,1404 | a2:   3,3621 | a3:  -0,6236 | b:   3,0312
Iteration:  14000 | MSE:  0,21444 | a1:   2,1343 | a2:   3,3424 | a3:  -0,6475 | b:   3,1413
Iteration:  15000 | MSE:  0,19105 | a1:   2,1278 | a2:   3,3236 | a3:  -0,6691 | b:   3,2454
Iteration:  16000 | MSE:  0,17021 | a1:   2,1214 | a2:   3,3057 | a3:  -0,6890 | b:   3,3437
Iteration:  17000 | MSE:  0,15165 | a1:   2,1152 | a2:   3,2888 | a3:  -0,7073 | b:   3,4365
Iteration:  18000 | MSE:  0,13511 | a1:   2,1091 | a2:   3,2727 | a3:  -0,7243 | b:   3,5241
Iteration:  19000 | MSE:  0,12038 | a1:   2,1032 | a2:   3,2575 | a3:  -0,7402 | b:   3,6069
Iteration:  20000 | MSE:  0,10726 | a1:   2,0976 | a2:   3,2431 | a3:  -0,7550 | b:   3,6850
Iteration:  21000 | MSE:  0,09556 | a1:   2,0922 | a2:   3,2295 | a3:  -0,7690 | b:   3,7587
Iteration:  22000 | MSE:  0,08515 | a1:   2,0871 | a2:   3,2167 | a3:  -0,7820 | b:   3,8283
Iteration:  23000 | MSE:  0,07586 | a1:   2,0823 | a2:   3,2045 | a3:  -0,7944 | b:   3,8940
Iteration:  24000 | MSE:  0,06759 | a1:   2,0777 | a2:   3,1931 | a3:  -0,8060 | b:   3,9560
Iteration:  25000 | MSE:  0,06022 | a1:   2,0734 | a2:   3,1823 | a3:  -0,8169 | b:   4,0146
Iteration:  26000 | MSE:  0,05366 | a1:   2,0693 | a2:   3,1720 | a3:  -0,8272 | b:   4,0699
Iteration:  27000 | MSE:  0,04781 | a1:   2,0654 | a2:   3,1624 | a3:  -0,8369 | b:   4,1220
Iteration:  28000 | MSE:  0,04260 | a1:   2,0618 | a2:   3,1533 | a3:  -0,8460 | b:   4,1712
Iteration:  29000 | MSE:  0,03795 | a1:   2,0583 | a2:   3,1447 | a3:  -0,8547 | b:   4,2177
Iteration:  30000 | MSE:  0,03382 | a1:   2,0550 | a2:   3,1366 | a3:  -0,8628 | b:   4,2616
Iteration:  31000 | MSE:  0,03013 | a1:   2,0519 | a2:   3,1290 | a3:  -0,8705 | b:   4,3030
Iteration:  32000 | MSE:  0,02685 | a1:   2,0491 | a2:   3,1217 | a3:  -0,8778 | b:   4,3421
Iteration:  33000 | MSE:  0,02392 | a1:   2,0463 | a2:   3,1149 | a3:  -0,8847 | b:   4,3790
Iteration:  34000 | MSE:  0,02131 | a1:   2,0437 | a2:   3,1085 | a3:  -0,8911 | b:   4,4138
Iteration:  35000 | MSE:  0,01899 | a1:   2,0413 | a2:   3,1024 | a3:  -0,8972 | b:   4,4466

--- Training Complete (Tables) ---
Learned parameters:  a1 =    2,0413 | a2 =    3,1024 | a3 =   -0,8972 | b =    4,4466
Expected parameters: a1 =    2,0000 | a2 =    3,0000 | a3 =   -1,0000 | b =    5,0000
```

Rysunek 2.2. Wyniki regresji liniowej z trzema zmiennymi niezależnymi - wersja z wykorzystaniem tablic

2.3. Zamiast tablic - macierze

Kolejnym krokiem w naszej wędrówce po regresji wielorakiej będzie wykorzystanie macierzy. Zamiast tablic jednowymiarowych użyjemy tablic dwuwymiarowych (macierzy), a co za tym idzie zamiast działań na skalarach (poszczególnych elementach tablicy) będziemy wykonywać działania na macierzach (operacje macierzowe).

Poniżej znajduje się opis operacji macierzowych, które na potrzeby niniejszego rozdziału zaimplementowano w klasie `ArrayExtensions`:

- `Add`: dodaje wartość skalarną do każdej komórki macierzy;
- `Mean`: oblicza średnią wszystkich komórek macierzy;
- `Multiply`: mnoży każdą komórkę macierzy przez wartość skalarną;

- `MultiplyDot`: mnoży macierz przez inną macierz przy użyciu iloczynu skalarnego (dot product);
- `Power`: podnosi każdą komórkę macierzy do potęgi;
- `Subtract`: odejmuje wartości z drugiej macierzy od pierwszej;
- `Sum`: oblicza sumę wszystkich komórek macierzy;
- `Transpose`: transponuje macierz, zamieniając wiersze na kolumny i odwrotnie.

> **NOTE**
>
> Kod źródłowy klasy `ArrayExtensions` znajduje się na GitHub a także w Dodatku na końcu rozdziału.

Tak wygląda kod implementujący budowę modelu wielorakiej regresji liniowej z wykorzystaniem macierzy:

```
// 1. Convert data to matrices

// Number of samples and coefficients
int n = data.GetLength(0);
int numCoefficients = data.GetLength(1) - 1;

float[,] X = new float[n, numCoefficients];
float[,] Y = new float[n, 1];

// Prepare the feature matrix X and the target vector Y
for (int row = 0; row < n; row++)
{
    for (int j = 0; j < numCoefficients; j++)
    {
        X[row, j] = data[row, j];
    }
    Y[row, 0] = data[row, numCoefficients];
}

// 2. Initialize model parameters

// Coefficients for the independent variables and the bias term
float[,] A = new float[numCoefficients, 1];
float b = 0;

// 3. Training loop

for (int iteration = 1; iteration <= Iterations; iteration++)
{
```

```
    // Prediction and error calculation

    // Make predictions for all samples at once: predictions = X * a + b
    float[,] predictions = X.MultiplyDot(A).Add(b);

    // Calculate errors for all samples: errors = predictions - Y
    float[,] errors = predictions.Subtract(Y);

    // Calculate the gradient for the coefficients 'a': ∂MSE/∂a = 2/n * X^T * errors
    // X.Transpose() aligns features with their corresponding errors for the dot product
    // We can pre-calculate X.Transpose() and (2.0f / n) for efficiency, but let's leave it
as is for clarity
    float[,] deltaA = X.Transpose().MultiplyDot(errors).Multiply(2.0f / n);

    // ∂MSE/∂b = 2/n * sum(errors)
    float deltaB = 2.0f / n * errors.Sum();

    // Update regression parameters using gradient descent
    A = A.Subtract(deltaA.Multiply(LearningRate));
    b -= LearningRate * deltaB;

    if (iteration % PrintEvery == 0)
    {
        // Calculate the Mean Squared Error loss: MSE = mean(errors^2)
        float meanSquaredError = errors.Power(2).Mean();

        Console.WriteLine($"Iteration: {iteration,6} | MSE: {meanSquaredError,8:F5} | a1:
{A[0, 0],8:F4} | a2: {A[1, 0],8:F4} | a3: {A[2, 0],8:F4} | b: {b,8:F4}");
    }
}

// 4. Output learned parameters

Console.WriteLine("\n--- Training Complete (Matrices) ---");
Console.WriteLine($"{"Learned parameters:",-20} a1 = {A[0, 0],9:F4} | a2 = {A[1, 0],9:F4} |
a3 = {A[2, 0],9:F4} | b = {b,9:F4}");
Console.WriteLine($"{"Expected parameters:",-20} a1 = {2,9:F4} | a2 = {3,9:F4} | a3 =
{-1,9:F4} | b = {5,9:F4}");
```

Listing 2.3. Wieloraka regresja liniowa z trzema zmiennymi niezależnymi - wersja z wykorzystaniem macierzy

Główne zmiany w stosunku do poprzednich wersji:

- Zamiast tablicy `data` mamy teraz dwie macierze: `X` (z niezależnymi zmiennymi) i `Y` (z zależną zmienną).

- Współczynniki regresji są teraz przechowywane w macierzy `A`, która ma rozmiar `numCoefficients x 1`.
- Obliczenia predykcji, błędów i gradientów są wykonywane na całych macierzach.
- Usunięcie pętli `foreach` (iteracji po poszczególnych obserwacjach) na rzecz operacji macierzowych, co sprawia, że kod jest bardziej zwięzły i czytelny.

I jeszcze jedna uwaga na temat powyższego programu. Wyrażenie `X.Transpose().MultiplyDot(errors).Multiply(2.0f / n)` to odpowiednik wzoru na gradient zapisanego w notacji macierzowej (porównaj wzór 1.10 z poprzedniego rozdziału):

$$\frac{\partial}{\partial A} MSE = \frac{2}{n} X^T (P - Y) \tag{2.1}$$

Skąd bierze się transpozycja macierzy `X` (X^T)? Jest ona niezbędna, aby wymiary (`wiersze x kolumny`) macierzy się zgadzały. Mnożąc macierz cech po transpozycji (`k x n`, gdzie `k` to liczba cech, a `n` to liczba próbek) przez wektor błędów (`n x 1`), otrzymujemy wektor gradientów (`k x 1`), w którym każdy element odpowiada gradientowi dla jednego współczynnika `a`.

2.4. Jeszcze prościej? Włączenie wyrazu wolnego do macierzy

Zauważmy, że wyraz wolny `b` wciąż jest oddzielną zmienną. Możemy go włączyć do macierzy współczynników `A`, stosując pewną sztuczkę: dodajemy do naszej macierzy danych `X` dodatkową kolumnę wypełnioną jedynkami.

Dzięki temu wyraz wolny `b` stanie się kolejnym współczynnikiem regresji, a my nie będziemy musieli go już osobno obsługiwać w kodzie. Poniżej znajduje się zmodyfikowany kod według tego podejścia:

```
// 1. Convert data to matrices with a bias term

// Number of samples and coefficients
int n = data.GetLength(0);
int numCoefficients = data.GetLength(1) - 1;

float[,] XAnd1 = new float[n, numCoefficients + 1]; // +1 column for bias term
float[,] Y = new float[n, 1];

// Prepare the feature matrix XAnd1 with the bias term and the target vector Y
for (int row = 0; row < n; row++)
{
    for (int j = 0; j < numCoefficients; j++)
    {
        XAnd1[row, j] = data[row, j];
    }
```

```
    XAnd1[row, numCoefficients] = 1; // Bias term
    Y[row, 0] = data[row, numCoefficients];
}

// 2. Initialize model parameters

// Coefficients for the independent variables and the bias term
float[,] AB = new float[numCoefficients + 1, 1];

// 3. Training loop

for (int iteration = 1; iteration <= Iterations; iteration++)
{
    // Prediction and error calculation

    // Make predictions for all samples at once: predictions = XAnd1 * AB
    float[,] predictions = XAnd1.MultiplyDot(AB);

    // Calculate errors for all samples: errors = predictions - Y
    float[,] errors = predictions.Subtract(Y);

    // Calculate the gradient for the coefficients 'AB': ∂MSE/∂AB = 2/n * XAnd1^T * errors
    // XAnd1.Transpose() aligns features and the additional column for the bias term with
their corresponding errors for the dot product
    // We can pre-calculate XAnd1.Transpose() and (2.0f / n) for efficiency, but let's leave
it as is for clarity
    float[,] deltaAB = XAnd1.Transpose().MultiplyDot(errors).Multiply(2.0f / n);

    // Update regression parameters using gradient descent
    AB = AB.Subtract(deltaAB.Multiply(LearningRate));

    if (iteration % PrintEvery == 0)
    {
        // Calculate the Mean Squared Error loss: MSE = mean(errors^2)
        float meanSquaredError = errors.Power(2).Mean();

        Console.WriteLine($"Iteration: {iteration,6} | MSE: {meanSquaredError,8:F5} | a1:
{AB[0, 0],8:F4} | a2: {AB[1, 0],8:F4} | a3: {AB[2, 0],8:F4} | b: {AB[3, 0],8:F4}");
    }
}

// 4. Output learned parameters

Console.WriteLine("\n--- Training Complete (Matrices with Bias) ---");
Console.WriteLine($"{"Learned parameters:",-20} a1 = {AB[0, 0],9:F4} | a2 = {AB[1, 0],9:F4}
| a3 = {AB[2, 0],9:F4} | b = {AB[3, 0],9:F4}");
```

```
Console.WriteLine($"{"Expected parameters:",-20} a1 = {2,9:F4} | a2 = {3,9:F4} | a3 = {-1,9:F4} | b = {5,9:F4}");
```

Listing 2.4. Wieloraka regresja liniowa z trzema zmiennymi niezależnymi - wersja z włączonym wyrazem wolnym do macierzy

> **NOTE**
>
> Odpowiednik powyższego kodu dla Delphi Object Pascal można znaleźć tutaj.

2.5. Podsumowanie

W tym rozdziale omówiliśmy, jak można wykorzystać macierze do implementacji wielorakiej regresji liniowej. Zaczęliśmy od prostego przykładu z trzema zmiennymi niezależnymi, a następnie przeszliśmy do bardziej zaawansowanych technik, takich jak użycie tablic i macierzy do przechowywania danych i współczynników regresji. Za wisienkę 🍒 na torcie 🍰 uznaliśmy włączenie wyrazu wolnego do macierzy 🧐.

2.6. Dodatek

Poniżej znajduje się kod źródłowy klasy `ArrayExtensions`, w zakresie, który obejmuje implementację operacji macierzowych użytych w powyższym przykładzie. Miłego kompilowania!

```
public static class ArrayExtensions
{
    /// <summary>
    /// Adds a scalar value to each element of the matrix.
    /// </summary>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public static float[,] Add(this float[,] source, float scalar)
    {
        int rows = source.GetLength(0);
        int columns = source.GetLength(1);
        float[,] res = new float[rows, columns];

        for (int row = 0; row < rows; row++)
        {
            for (int col = 0; col < columns; col++)
            {
                res[row, col] = source[row, col] + scalar;
            }
        }
```

```
        return res;
    }

    /// <summary>
    /// Calculates the mean of all elements in the matrix.
    /// </summary>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public static float Mean(this float[,] source)
        => source.Sum() / source.Length;

    /// <summary>
    /// Multiplies each element of the matrix by a scalar value.
    /// </summary>
    /// <remarks>
    /// Complexity: O(n * m), where n = rows of <paramref name="source"/>, m = columns of
<paramref name="source"/>
    /// </remarks>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public static float[,] Multiply(this float[,] source, float scalar)
    {
        int rows = source.GetLength(0);
        int columns = source.GetLength(1);
        float[,] res = new float[rows, columns];

        for (int row = 0; row < rows; row++)
        {
            for (int col = 0; col < columns; col++)
            {
                res[row, col] = source[row, col] * scalar;
            }
        }

        return res;
    }

    /// <summary>
    /// Multiplies the current matrix with another matrix using the dot product.
    /// </summary>
    /// <remarks>
    /// Complexity: O(n * m * p), where n = rows of <paramref name="source"/>, m = shared
dimension, p = columns of <paramref name="matrix"/>
    /// </remarks>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public static float[,] MultiplyDot(this float[,] source, float[,] matrix)
    {
        Debug.Assert(source.GetLength(1) == matrix.GetLength(0));
```

```
    int matrixColumns = matrix.GetLength(1);

    int rows = source.GetLength(0);
    int columns = source.GetLength(1);

    float[,] res = new float[rows, matrixColumns];

    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < matrixColumns; j++)
        {
            float sum = 0;
            for (int k = 0; k < columns; k++)
            {
                sum += source[i, k] * matrix[k, j];
            }
            res[i, j] = sum;
        }
    }

    return res;
}

/// <summary>
/// Raises each element of the matrix to the specified power.
/// </summary>
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static float[,] Power(this float[,] source, int scalar)
{
    int rows = source.GetLength(0);
    int columns = source.GetLength(1);
    float[,] res = new float[rows, columns];

    for (int row = 0; row < rows; row++)
    {
        for (int col = 0; col < columns; col++)
        {
            res[row, col] = MathF.Pow(source[row, col], scalar);
        }
    }

    return res;
}

/// <summary>
```

```
/// Subtracts the elements of the specified matrix from the current matrix.
/// </summary>
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static float[,] Subtract(this float[,] source, float[,] matrix)
{
    Debug.Assert(source.GetLength(0) == matrix.GetLength(0));
    Debug.Assert(source.GetLength(1) == matrix.GetLength(1));

    int rows = source.GetLength(0);
    int columns = source.GetLength(1);
    float[,] res = new float[rows, columns];

    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < columns; j++)
        {
            res[i, j] = source[i, j] - matrix[i, j];
        }
    }

    return res;
}

/// <summary>
/// Calculates the sum of all elements in the matrix.
/// </summary>
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static float Sum(this float[,] source)
{
    // Sum over all elements.
    float sum = 0;
    int rows = source.GetLength(0);
    int cols = source.GetLength(1);

    for (int row = 0; row < rows; row++)
    {
        for (int col = 0; col < cols; col++)
        {
            sum += source[row, col];
        }
    }

    return sum;
}

/// <summary>
```

```
    /// Transposes the matrix by swapping its rows and columns.
    /// </summary>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public static float[,] Transpose(this float[,] source)
    {
        int rows = source.GetLength(0);
        int columns = source.GetLength(1);

        float[,] array = new float[columns, rows];

        for (int i = 0; i < rows; i++)
        {
            for (int j = 0; j < columns; j++)
            {
                array[j, i] = source[i, j];
            }
        }

        return array;
    }
}
```

Listing 2.5. Klasa ArrayExtensions implementująca operacje macierzowe

---

Created: 2025-06-28

Last modified: 2025-12-22

Title: **2. Po co nam macierze?**

Tags: [C#] [Sieci neuronowe] [Regresja liniowa] [Macierze]

3. Dane Boston Housing

W niniejszym rozdziale zajmiemy się analizą danych rzeczywistych pochodzących ze zbioru Boston Housing (GitHub). Posłużymy w tym celu omówioną już wcześniej wieloraką regresję liniową.

Aby zbadać jakość utworzonego modelu regresji, podzielimy zbiór dostępnych danych na zbiór uczący i zbiór testowy (na którym ocenimy jakość predykcji).

W rozdziale kolejnym przeprowadzimy analogiczne badanie, korzystając z samodzielnie skonstruowanej prostej sieci neuronowej. Następnie porównamy wyniki obu metod – jakość predykcji generowanych przez sieć neuronową zestawimy z wynikami otrzymanymi z modelu regresji przy identycznym podziale danych (trening/test).

3.1. Zbiór danych Boston Housing

Zbiór Boston Housing zawiera informacje o cenach domów w różnych dzielnicach Bostonu oraz cechach tych dzielnic, które mogą wpływać na ceny nieruchomości. Zbiór ten jest często wykorzystywany do celów edukacyjnych i badawczych w dziedzinie uczenia maszynowego i statystyki. Składa się on z 506 rekordów (próbek), z których każdy zawiera 13 cech (zmiennych niezależnych) oraz jedną zmienną docelową (mediana cen domów [tys. USD]). Dokładniejszy opis znajduje się tu.

Poniżej przedstawiono pierwszych 20 rekordów z tego zbioru:

1	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	b	lstat	medv
2	0.00632	18	2.31	0	0.538	6.575	65.2	4.09	1	296	15.3	396.9	4.98	24
3	0.02731	0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.9	9.14	21.6
4	0.02729	0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
5	0.03237	0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
6	0.06905	0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.9	5.33	36.2
7	0.02985	0	2.18	0	0.458	6.43	58.7	6.0622	3	222	18.7	394.12	5.21	28.7
8	0.08829	12.5	7.87	0	0.524	6.012	66.6	5.5605	5	311	15.2	395.6	12.43	22.9
9	0.14455	12.5	7.87	0	0.524	6.172	96.1	5.9505	5	311	15.2	396.9	19.15	27.1
10	0.21124	12.5	7.87	0	0.524	5.631	100	6.0821	5	311	15.2	386.63	29.93	16.5
11	0.17004	12.5	7.87	0	0.524	6.004	85.9	6.5921	5	311	15.2	386.71	17.1	18.9
12	0.22489	12.5	7.87	0	0.524	6.377	94.3	6.3467	5	311	15.2	392.52	20.45	15
13	0.11747	12.5	7.87	0	0.524	6.009	82.9	6.2267	5	311	15.2	396.9	13.27	18.9
14	0.09378	12.5	7.87	0	0.524	5.889	39	5.4509	5	311	15.2	390.5	15.71	21.7
15	0.62976	0	8.14	0	0.538	5.949	61.8	4.7075	4	307	21	396.9	8.26	20.4
16	0.63796	0	8.14	0	0.538	6.096	84.5	4.4619	4	307	21	380.02	10.26	18.2
17	0.62739	0	8.14	0	0.538	5.834	56.5	4.4986	4	307	21	395.62	8.47	19.9
18	1.05393	0	8.14	0	0.538	5.935	29.3	4.4986	4	307	21	386.85	6.58	23.1
19	0.7842	0	8.14	0	0.538	5.99	81.7	4.2579	4	307	21	386.75	14.67	17.5
20	0.80271	0	8.14	0	0.538	5.456	36.6	3.7965	4	307	21	288.99	11.69	20.2

Rysunek 3.1. Pierwsze rekordy ze zbioru Boston Housing

3.2. Wieloraka regresja liniowa

Wykorzystajmy kod przedstawiony w poprzednim rozdziale do implementacji wielorakiej regresji liniowej z użyciem macierzy na omawianym zbiorze danych. Implementację tę przedstawia poniższy listing (pełna wersja znajduje się na GitHub):

```
// Define hyperparameters for both routines

const float LearningRate = 0.0005f;
```

```
const int Iterations = 48_000;
const int PrintEvery = 2_000;
const float TestSplitRatio = 0.7f;
const int RandomSeed = 251113;

// 1. Get data

(float[,] trainData, float[,] testData) = GetData();

// 2. Copy trainData and testData to matrices with bias term

int inputFeatureCount = trainData.GetLength(1) - 1;
int nTrain = trainData.GetLength(0);
int nTest = testData.GetLength(0);

float[,] XTrainAnd1 = new float[nTrain, inputFeatureCount + 1];
float[,] YTrain = new float[nTrain, 1];

float[,] XTestAnd1 = new float[nTest, inputFeatureCount + 1];
float[,] YTest = new float[nTest, 1];

// Prepare feature matrix XTrainAnd1 with bias term and target vector YTrain
for (int i = 0; i < nTrain; i++)
{
    for (int j = 0; j < inputFeatureCount; j++)
    {
        XTrainAnd1[i, j] = trainData[i, j];
    }

    // Add bias term
    XTrainAnd1[i, inputFeatureCount] = 1;

    // Target values
    YTrain[i, 0] = trainData[i, inputFeatureCount];
}

// Prepare feature matrix XTestAnd1 with bias term and target vector YTest
for (int i = 0; i < nTest; i++)
{
    for (int j = 0; j < inputFeatureCount; j++)
    {
        XTestAnd1[i, j] = testData[i, j];
    }

    // Add bias term
    XTestAnd1[i, inputFeatureCount] = 1;
```

```
    // Target values
    YTest[i, 0] = testData[i, inputFeatureCount];
}

// 3. Initialize model parameters

// Coefficients for our independent variables and the bias term initialized to zero
float[,] AB = new float[inputFeatureCount + 1, 1];

// 4. Training loop

float[,] XTrainAnd1T = XTrainAnd1.Transpose();
float twoOverN = 2.0f / nTrain;
for (int iteration = 1; iteration <= Iterations; iteration++)
{
    // Prediction and error calculation

    // Make predictions for all samples at once: predictions = XTrainAnd1 * AB
    float[,] predictions = XTrainAnd1.MultiplyDot(AB);

    // Calculate errors for all samples: errors = predictions - YTrain
    float[,] errors = predictions.Subtract(YTrain);

    // Calculate gradient for coefficients 'AB': ∂MSE/∂AB = 2/n * XTrainAnd1^T * errors
    float[,] deltaAB = XTrainAnd1T.MultiplyDot(errors).Multiply(twoOverN);

    // Update regression parameters using gradient descent
    AB = AB.Subtract(deltaAB.Multiply(LearningRate));

    if (iteration % PrintEvery == 0)
    {
        // Calculate the Mean Squared Error loss: MSE = mean(errors^2)
        float meanSquaredError = errors.Power(2).Mean();

        if (float.IsNaN(meanSquaredError))
        {
            Console.WriteLine($"NaN detected at iteration {iteration}");
            break;
        }

        Console.WriteLine($"Iteration: {iteration,6} | MSE: {meanSquaredError,8:F5} | a1:
{AB[0, 0],8:F4} | a2: {AB[1, 0],8:F4} | a3: {AB[2, 0],8:F4} | ... | b:
{AB[inputFeatureCount, 0],8:F4}");
    }
}
```

```
// 5. Output learned parameters

Console.WriteLine("\n--- Training Complete (Matrices with Bias on Boston Data) ---");
Console.WriteLine("Learned parameters:");

for (int i = 0; i < inputFeatureCount; i++)
{
    Console.WriteLine($" a{i + 1,-2} = {AB[i, 0],8:F4}");
}
Console.WriteLine($" b   = {AB[inputFeatureCount, 0],8:F4}");

Console.WriteLine();
Console.WriteLine("Sample predictions vs actual values:");
Console.WriteLine();
Console.WriteLine($"{"Sample No",14}{"Predicted",14}{"Actual",14}");
Console.WriteLine();

// Show predictions for the test set

int[] showTestSamples = { 0, 1, 2, nTest - 3, nTest - 2, nTest - 1 };
float[,] testPredictions = XTestAnd1.MultiplyDot(AB);

for (int i = 0; i < showTestSamples.Length; i++)
{
    int testSampleIndex = showTestSamples[i];
    float predicted = testPredictions[testSampleIndex, 0];
    float actual = YTest[testSampleIndex, 0];
    Console.WriteLine(
        $"{testSampleIndex + 1,14}" +
        $"{predicted,14:F4}" +
        $"{actual,14:F4}"
    );
}

// Show MSE for test data

float[,] testErrors = YTest.Subtract(testPredictions);
float testMeanSquaredError = testErrors.Power(2).Mean();
Console.ForegroundColor = ConsoleColor.Yellow;
Console.WriteLine($"\nMSE on test data: {testMeanSquaredError:F5}");
Console.ResetColor();
```

Listing 3.1. Wieloraka regresja liniowa z użyciem macierzy na zbiorze danych Boston Housing

3.2.1. Pobieranie, standaryzowanie, i permutacja danych oraz ich podział na zbiór uczący i zbiór testowy

W powyższym listingu znalazła się linia `(float[,] trainData, float[,] testData) = GetData()`, która odpowiada za pobranie i przygotowanie danych do analizy. Wywołuje ona funkcję pokazaną na listingu 3.2:

```
static (float[,] TrainData, float[,] TestData) GetData()
{
    float[,] bostonData = LoadCsv("..\\..\\..\\..\\..\\data\\Boston\\BostonHousing.csv");

    // Number of independent variables
    int inputFeatureCount = bostonData.GetLength(1) - 1;

    // Standardize each feature column (mean = 0, stddev = 1) except the target variable
(last column)
    // Note: the upper bound in Range is exclusive, so we use inputFeatureCount to exclude
the last column
    bostonData.Standardize(0..inputFeatureCount);

    // Permute the data randomly
    bostonData.PermuteInPlace(RandomSeed);

    // Return train and test data split by ratio
    return bostonData.SplitRowsByRatio(TestSplitRatio);
}
```

Listing 3.2. Funkcja pobierająca, standaryzująca, permutująca i dzieląca dane Boston Housing na zbiór uczący i testowy

Jak widzimy, powyższy kod:

- ładuje dane z pliku CSV,
- standaryzuje cechy wejściowe (zmienne niezależne),
- losowo permutuje dane,
- dzieli dane na zbiór uczący i testowy według określonego stosunku.

3.2.2. Standaryzacja cech wejściowych

Zwróćmy uwagę na wywołanie w powyższej procedurze metody bostonData.Standardize(0..numCoefficients), która standaryzuje cechy wejściowe (zmienne niezależne) do średniej 0 i odchylenia standardowego 1. Standaryzacja danych jest istotnym etapem przygotowania danych do trenowania modeli uczenia maszynowego, zwłaszcza algorytmów opartych na optymalizacji gradientowej, ponieważ sprzyja szybszej i bardziej stabilnej konwergencji.

Konwergencja algorytmu optymalizacyjnego to proces, w którym algorytm poprzez iteracyjne aktualizacje parametrów stopniowo zbliża się do optymalnego rozwiązania, czyli minimum funkcji straty. Standaryzacja danych ułatwia ten proces, ponieważ gdy cechy mają porównywalne skale, funkcja straty ma lepsze własności geometryczne, co pozwala algorytmowi unikać zbyt dużych lub zbyt małych kroków aktualizacji parametrów.

W naszym wypadku, kod metody `Standardize` wygląda następująco:

```
public static void Standardize(this float[,] source, Range? columnRange = null)
{
    int rows = source.GetLength(0);
    int columns = source.GetLength(1);

    int beginColumn, endColumn;

    if (columnRange is not null)
    {
        var (offset, length) = columnRange.Value.GetOffsetAndLength(columns);
        beginColumn = offset;
        endColumn = beginColumn + length;
    }
    else
    {
        beginColumn = 0;
        endColumn = columns;
    }

    for (int col = beginColumn; col < endColumn; col++)
    {
        // Calculate mean
        float sum = 0;
        for (int row = 0; row < rows; row++)
        {
            sum += source[row, col];
        }
        float mean = sum / rows;

        // Calculate standard deviationFromMean
        float sumOfSquares = 0;
        for (int row = 0; row < rows; row++)
        {
            float value = source[row, col] - mean;
            sumOfSquares += value * value;
        }
        float stdDev = MathF.Sqrt(sumOfSquares / rows);
```

```
            if (stdDev == 0)
            {
                stdDev = 1; // To avoid division by zero
            }

            // Standardize values
            for (int row = 0; row < rows; row++)
            {
                source[row, col] = (source[row, col] - mean) / stdDev;
            }
        }
    }
```

Listing 3.3. Metoda standaryzująca kolumny macierzy do średniej 0 i odchylenia standardowego 1

Metoda ta oblicza średnią i odchylenie standardowe dla każdej kolumny w całym lub w określonym zakresie (`Range? columnRange`), a następnie standaryzuje wartości w tej kolumnie.

3.2.2.1. Szybsza metoda obliczania odchylenia standardowego

Korzystając z własności wariancji, możemy obliczyć odchylenie standardowe nieco szybciej, bez konieczności dwukrotnego przechodzenia przez dane. Wariancja jest zdefiniowana jako średnia kwadratów różnic wartości od średniej:

$$\frac{\sum(x_i - \bar{x})^2}{N} \tag{3.1}$$

Powyższy wzór można przekształcić do postaci:

$$\begin{aligned}
\frac{\sum(x_i - \bar{x})^2}{N} &= \frac{\sum(x_i^2 - 2x_i\bar{x} + \bar{x}^2)}{N} \\
&= \frac{\sum x_i^2}{N} - \frac{\sum 2x_i\bar{x}}{N} + \frac{\sum \bar{x}^2}{N} \\
&= \frac{\sum x_i^2}{N} - 2\bar{x}\frac{\sum x_i}{N} + \bar{x}^2 \\
&= \frac{\sum x_i^2}{N} - 2\bar{x} \cdot \bar{x} + \bar{x}^2 \\
&= \frac{\sum x_i^2}{N} - 2\bar{x}^2 + \bar{x}^2 \\
&= \frac{\sum x_i^2}{N} - \bar{x}^2
\end{aligned} \tag{3.2}$$

umożliwiając wyliczenie wariancji w pojedynczym przebiegu przez dane (*single pass*), w którym obliczamy sumę wartości (do obliczenia średniej $\bar{x}$) i sumę kwadratów wartości (do obliczenia średniej z kwadratów

wartości $\frac{\sum x_i^2}{N}$). Odchylenie standardowe jest pierwiastkiem kwadratowym z wariancji.

Oto przykładowa implementacja wersji *single pass* (pełen listing można znaleźć na GitHub):

```
// Calculate standard deviation in a single pass
float sum = 0, sumOfSquares = 0;
for (int row = 0; row < rows; row++)
{
    float value = source[row, col];
    sum += value;
    sumOfSquares += value * value;
}
float mean = sum / rows;
float variance = (sumOfSquares / rows) - (mean * mean);
float stdDev = MathF.Sqrt(variance);
```

Listing 3.4. Szybsza metoda obliczania odchylenia standardowego z wykorzystaniem własności wariancji ze wzoru (3.2)

> **NOTE**
>
> Podnoszenie do kwadratu realizujemy poprzez mnożenie `value * value`, zamiast użycia funkcji `MathF.Pow(value, 2)`, ponieważ mnożenie jest szybsze niż wywołanie funkcji potęgowania.

3.2.3. Efekt działania regresji liniowej na zbiorze Boston Housing

Poniżej przedstawiono efekt działania programu. Widzimy, że w trakcie treningu modelu wartość funkcji straty (MSE) stopniowo maleje, co wskazuje na to, że model uczy się dopasowywać do danych treningowych.

```
Iteration:  22000 | MSE: 19,48249 | a1:  -0,9110 | a2:   1,3007 | a3:  -0,4898 | ... | b:  22,4733
Iteration:  24000 | MSE: 19,47082 | a1:  -0,9218 | a2:   1,3184 | a3:  -0,4707 | ... | b:  22,4721
Iteration:  26000 | MSE: 19,46206 | a1:  -0,9306 | a2:   1,3328 | a3:  -0,4534 | ... | b:  22,4711
Iteration:  28000 | MSE: 19,45545 | a1:  -0,9379 | a2:   1,3447 | a3:  -0,4380 | ... | b:  22,4702
Iteration:  30000 | MSE: 19,45045 | a1:  -0,9440 | a2:   1,3545 | a3:  -0,4242 | ... | b:  22,4695
Iteration:  32000 | MSE: 19,44664 | a1:  -0,9491 | a2:   1,3628 | a3:  -0,4120 | ... | b:  22,4688
Iteration:  34000 | MSE: 19,44374 | a1:  -0,9534 | a2:   1,3698 | a3:  -0,4013 | ... | b:  22,4682
Iteration:  36000 | MSE: 19,44154 | a1:  -0,9570 | a2:   1,3757 | a3:  -0,3917 | ... | b:  22,4677
Iteration:  38000 | MSE: 19,43987 | a1:  -0,9601 | a2:   1,3807 | a3:  -0,3834 | ... | b:  22,4673
Iteration:  40000 | MSE: 19,43856 | a1:  -0,9628 | a2:   1,3851 | a3:  -0,3760 | ... | b:  22,4669
Iteration:  42000 | MSE: 19,43758 | a1:  -0,9651 | a2:   1,3888 | a3:  -0,3695 | ... | b:  22,4666
Iteration:  44000 | MSE: 19,43682 | a1:  -0,9671 | a2:   1,3920 | a3:  -0,3639 | ... | b:  22,4663
Iteration:  46000 | MSE: 19,43625 | a1:  -0,9688 | a2:   1,3948 | a3:  -0,3589 | ... | b:  22,4661
Iteration:  48000 | MSE: 19,43581 | a1:  -0,9702 | a2:   1,3972 | a3:  -0,3545 | ... | b:  22,4658

--- Training Complete (Matrices with Bias on Boston Data) ---
Learned parameters:
 a1  =  -0,9702
 a2  =   1,3972
 a3  =  -0,3545
 a4  =   0,5873
 a5  =  -2,0943
 a6  =   1,9746
 a7  =   0,1827
 a8  =  -3,4413
 a9  =   2,7323
 a10 =  -2,3118
 a11 =  -1,8305
 a12 =   0,4663
 a13 =  -3,8503
 b   =  22,4658

Sample predictions vs actual values:

     Sample No     Predicted        Actual

             1       17,5248       20,2000
             2       17,6483       20,6000
             3       21,4478       18,6000
           150       25,8629       50,0000
           151       14,5508       14,5000
           152       34,3006       33,8000

MSE on test data: 29,49182
```

Rysunek 3.2. Wyniki wielorakiej regresji liniowej na zbiorze Boston Housing

Dla zbioru *uczącego* wartość funkcji straty po 48 000 iteracjach wynosi `MSE = 19.43581`, co oznacza, że taki jest średni błąd kwadratowy między przewidywanymi a rzeczywistymi cenami domów. Wyświetlane są również wyuczone parametry regresji (współczynniki `a1`, `a2`, ..., `a13` oraz wyraz wolny `b`). Największy (co do wartości bezwzględnej) wpływ na cenę posiada cecha `a13 = -3.8503`, co dla danych Boston Housing odpowiada zmiennej `LSTAT` (procent populacji o niskich dochodach).

Dla zbioru *testowego* MSE wynosi `MSE = 29.49182`, co jest wyższą wartością niż dla zbioru uczącego.

W kolejnym rozdziale zobaczymy jak z tym samym problemem predykcji cen domów na naszym zbiorze danych poradzi sobie sieć neuronowa.

Created: 2025-11-09

Last modified: 2025-12-22

Title: **3. Dane Boston Housing**

Tags: [C#] [Sieci neuronowe] [Regresja liniowa] [Macierze] [Boston Housing] [Standaryzacja danych]

4. Pierwsza sieć neuronowa

W poprzednim rozdziale utworzyliśmy model wielorakiej regresji liniowej i zbadaliśmy jakość predycji przez ten model generowanej (MSE) na danych Boston Housing. W tym rozdziale natomiast utworzymy prostą sieć neuronową i pokażemy sposób, w jaki generuje ona swoje predykcje. Porównamy również sposoby uczenia się (uaktualniania parametrów) obu modeli. Na koniec porównamy jakość generowania predykcji dla danych, które nie były wykorzystywane podczas treningu.

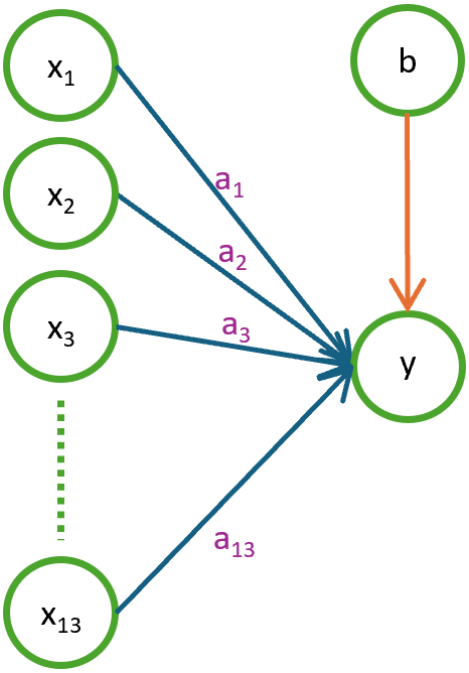
4.1. Charakterystyka modeli i obliczanie funkcji straty

Na początek przeprowadźmy porównanie obu modeli. Zobaczmy jak oba modele są zbudowane, jak generują swoje predykcje oraz jak obliczana jest funkcja straty MSE w obu przypadkach.

4.1.1. Wieloraka regresja liniowa

4.1.1.1. Model

Na rysunku 4.1. przedstawiono schemat modelu wielorakiej regresji liniowej, który zbudowaliśmy w poprzednim rozdziale.



Rysunek 4.1. Schemat modelu wielorakiej regresji liniowej

Powyższy diagram przedstawia pojedyncze równanie liniowe. Widzimy na nim:

- zmienne wejściowe: $x_1, x_2, x_3, \ldots, x_{13}$
- parametry modelu (wagi): $a_1, a_2, a_3, \ldots, a_{13}$
- wyraz wolny (bias): b
- wyjście: y.

Proces generowania predykcji wygląda następująco:

$$\hat{y} := a_1 x_1 + a_2 x_2 + \cdots + a_{13} x_{13} + b$$

Podkreślmy, że w odróżnieniu od tego, co zobaczymy za chwilę, model regresji liniowej charakteryzuje się:

- brakiem warstw ukrytych
- brakiem funkcji aktywacji (a właściwie funkcją aktywacji liniową o wzorze $f(x) = x$)
- wyjściem będącym jedynie kombinacją liniową wejść
- możliwością wyuczenia jedynie relacji liniowych

4.1.1.2. Obliczanie funkcji straty

Dowiedzieliśmy się już (z listingu 2.3), że w przypadku wielorakiej regresji liniowej obliczanie MSE wygląda następująco:

```
float[,] predictions = X.MultiplyDot(A).Add(b);
float[,] errors = predictions.Subtract(Y);
float meanSquaredError = errors.Power(2).Mean();
```

Listing 4.1. Obliczanie MSE dla wielorakiej regresji liniowej

gdzie `X` to macierz cech wejściowych, `A` to wektor współczynników regresji, `b` to wyraz wolny, a `Y` to wektor wartości docelowych.

Matematycznie zapisalibyśmy to jako:

$$\begin{aligned} P &= X \cdot A + b \\ E &= P - Y \\ MSE &= \frac{1}{n} \sum_{i=1}^{n} E_i^2 \end{aligned} \tag{4.1}$$

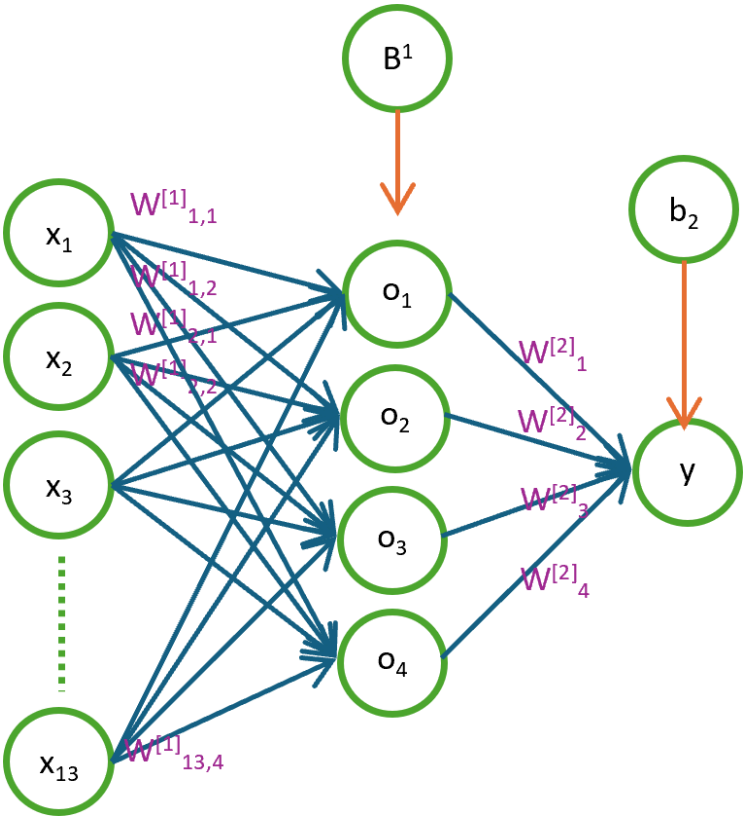
gdzie P to macierz predykcji, E to macierz błędów (różnic) między rzeczywistymi wartościami docelowymi Y a przewidywaniami regresji P, a n to liczba próbek.

4.1.2. Sieć neuronowa

Porównajmy to teraz z siecią neuronową.

4.1.2.1. Model

Na rysunku 4.2. przedstawiono schemat prostej sieci neuronowej z jedną warstwą ukrytą.



Rysunek 4.2. Schemat prostej sieci neuronowej z jedną warstwą ukrytą

Powyższy model składa się z:

- warstwy wejściowej (takiej samej jak w regresji: $x_1, x_2, x_3, \ldots, x_{13}$)
- warstwy ukrytej (pierwszej, z czterema neuronami o wyjściach o_1, o_2, o_3, o_4)
- warstwy wyjściowej (drugiej, z jednym neuronem o wyjściu y - analogicznie do modelu regresji liniowej)

Na rysunku zaznaczono również parametry modelu:

- wagi $W^{[1]}$ i $W^{[2]}$ łączące warstwę wejściową z warstwą ukrytą oraz warstwę ukrytą z warstwą wyjściową
- biasy (wyrazy wolne) $B^{[1]}$ i $b^{[2]}$ dla warstwy ukrytej i wyjściowej

4.1.2.2. Obliczanie funkcji straty

Zobaczmy teraz jak wygląda obliczanie MSE w przypadku naszej sieci neuronowej. Oto odpowiedni fragment kodu:

```
/*
   W1 - weights for the first layer [ inputSize (no of columns/attributes of X) x hiddenSize ]
   W2 - weights for the second layer [ hiddenSize x 1 ]
   B1 - bias for the first layer (for every neuron in the first layer)
   b2 - bias for the second layer (there is only one neuron in the second layer)
   M1 - input multiplied by W1
   N1 - input multiplied by W1 plus B1
   O1 - result of the activation function applied to (input multiplied by W1 plus B1)
   M2 - result of O1 (result of the activation function from the first layer) multiplied by W2
   predictions - (M2 + b2)
```

```
   errors - subtract predictions from Y
   meanSquaredError - MSE, mean of errors squared
*/

// The first layer (hidden)
float[,] M1 = X.MultiplyDot(W1);
float[,] N1 = M1.AddRow(B1);
// Apply sigmoid activation function, so we can get O1 - outputs of the first layer
float[,] O1 = N1.Sigmoid();

// The second layer (output)
float[,] M2 = O1.MultiplyDot(W2);
float[,] predictions = M2.Add(b2);

// Calculate errors for all samples: errors = predictions - Y
float[,] errors = predictions.Subtract(Y);

float meanSquaredError = errors.Power(2).Mean();
```

Listing 4.2. Obliczanie MSE dla prostej sieci neuronowej

gdzie:

- `X` to macierz cech wejściowych,
- `W1` to macierz wag pierwszej warstwy (ukrytej),
- `M1` to macierz wyników mnożenia wejść `X` przez wagi pierwszej warstwy `W1`,
- `B1` to wektor biasów pierwszej warstwy,
- `N1` to macierz wyników dodania biasów `B1` do `M1`,
- `O1` to macierz wyjść pierwszej warstwy po zastosowaniu funkcji aktywacji sigmoid,
- `W2` to macierz wag drugiej warstwy (wyjściowej),
- `M2` to macierz wyników mnożenia wyjść pierwszej warstwy `O1` przez wagi drugiej warstwy,
- `b2` to wyraz wolny drugiej warstwy,
- `Y` to macierz wartości docelowych,
- `predictions` to macierz przewidywanych wartości wyjściowych,
- `errors` to macierz błędów (różnic) między rzeczywistymi wartościami docelowymi `Y` a przewidywaniami sieci `predictions`.

> **NOTE**
>
> Pełen kod omawiany w tym rozdziale znajduje się na GitHub.
>
> Fragmenty kody, które zostały przeniesione do Delphi Object Pascala znajdują się tu.

W matematycznym zapisie wygląda to następująco:

$$
\begin{aligned}
M^{[1]} &= X \cdot W^{[1]} \\
N^{[1]} &= M^{[1]} + B^{[1]} \\
O^{[1]} &= \sigma(N^{[1]}) \\
M^{[2]} &= O_1 \cdot W^{[2]} \\
P &= M^{[2]} + b^{[2]} \\
E &= P - Y \\
MSE &= \frac{1}{n} \sum_{i=1}^{n} E_i^2
\end{aligned}
\tag{4.2}
$$

gdzie:

- σ to funkcja aktywacji sigmoid,
- P to macierz predykcji sieci,
- E to macierz błędów między przewidywaniami sieci P a rzeczywistymi wartościami docelowymi Y,
- pozostałe symbole są analogiczne do tych używanych w listingu 4.2, np. $W^{[1]}$ oznacza macierz wag pierwszej warstwy, a $O^{[1]}$ to macierz wyjść pierwszej warstwy.

4.1.2.2.1. Funkcja aktywacji

Funkcja aktywacji jest to matematyczna funkcja stosowana w neuronach sieci neuronowej, która wprowadza nieliniowość do modelu. W naszym przypadku używamy funkcji sigmoid, która jest zdefiniowana wzorem:

$$
\sigma(x) = \frac{1}{1 + e^{-x}}
$$

Cały proces przebiega następująco:

- Najpierw obliczamy ważoną sumę wejść dla każdego neuronu w warstwie ukrytej (mnożenie macierzy `X` przez `W1` i dodanie biasów `B1`).
- Następnie stosujemy funkcję aktywacji sigmoid do tych sum, aby uzyskać wyjścia warstwy ukrytej `O1`.
- W następnej kolejności obliczamy ważoną sumę wyjść z warstwy ukrytej dla neuronu wyjściowego (mnożenie `O1` przez `W2` i dodanie biasu `b2`).
- Na końcu obliczamy błędy i MSE tak samo jak w przypadku regresji liniowej.

Tak wygląda obliczanie predykcji i funkcji straty MSE w naszej prostej sieci neuronowej.

4.2. Spadek gradientowy

Zajmijmy się teraz "drogą powrotną", czyli aktualizacją wag i wyrazów wolnych (biasów) w procesie uczenia się obu modeli.

4.2.1. Wieloraka regresja liniowa

Jak wiemy z poprzednich rozdziałów, obliczanie gradientów dla wielorakiej regresji liniowej wygląda następująco:

$$
\begin{aligned}
\frac{\partial}{\partial A} MSE &= \frac{2}{n} X^T (P - Y) \\
\frac{\partial}{\partial B} MSE &= \frac{2}{n} \sum (P - Y)
\end{aligned}
\tag{4.3}
$$

gdzie różnica $P - Y$ to macierz błędów.

Programistycznie przekłada się to na poniższy kod:

```
float[,] deltaA = X.Transpose().MultiplyDot(errors).Multiply(2.0f / n);
float deltaB = 2.0f / n * errors.Sum();
A = A.Subtract(deltaA.Multiply(LearningRate));
b -= LearningRate * deltaB;
```

Listing 4.3. Aktualizacja współczynników regresji liniowej za pomocą metody spadku gradientowego

gdzie `deltaA` i `deltaB` to gradienty funkcji straty względem współczynników `A` i wyrazu wolnego `b`.

4.2.2. Sieć neuronowa

A jak to wygląda w sieci neuronowej?

Aktualizacja wag i biasów w sieci neuronowej podczas treningu odbywa się - podobnie jak w przypadku regresji liniowej - z wykorzystaniem metody spadku gradientowego.

4.2.2.1. Reguła łańcuchowa

W przypadku sieci wielowarstwowych obliczanie gradientów związane jest z obliczeniami pochodnych funkcji złożonych. Jeżeli jakaś wartość wejściowa po przejściu przez dwie warstwy sieci (f i g) przyjmuje wartość:

$$P = f(g(X)) \tag{4.4}$$

to pochodna P', obliczana za pomocą reguły łańcuchowej, wyniesie:

$$P' = f'(g(X)) \cdot g'(X) \tag{4.5}$$

Regułę tę można również zapisać w inny sposób:

$$\frac{\partial P}{\partial X} = \frac{\partial P}{\partial g} \cdot \frac{\partial g}{\partial X} \tag{4.6}$$

4.2.2.2. Obliczanie gradientów

Spróbujmy teraz przeprowadzić wyliczenia wszystkich gradientów wykorzystywanych w procesie aktualizacji wag i biasów naszej sieci neuronowej. Aktualizacje te będą odbywać się wg poniższych wzorów (analog do wzorów (1.9) i (1.12) z rozdziału o regresji liniowej):

$$\begin{aligned} W^{[1]} &:= W^{[1]} - lr \cdot \frac{\partial}{\partial W^{[1]}} MSE \\ W^{[2]} &:= W^{[2]} - lr \cdot \frac{\partial}{\partial W^{[2]}} MSE \\ B^{[1]} &:= B^{[1]} - lr \cdot \frac{\partial}{\partial B^{[1]}} MSE \\ b^{[2]} &:= b^{[2]} - lr \cdot \frac{\partial}{\partial b^{[2]}} MSE \end{aligned} \tag{4.7}$$

Jak widać, musimy obliczyć cztery pochodne. Trzy z nich są macierzami: $\frac{\partial}{\partial W^{[1]}} MSE$, $\frac{\partial}{\partial W^{[2]}} MSE$, $\frac{\partial}{\partial B^{[1]}} MSE$, a czwarta jest skalarem: $\frac{\partial}{\partial b^{[2]}} MSE$.

W celu obliczenia tych pochodnych skorzystamy z poniższych wzorów.

4.2.2.2.1. Warstwa wyjściowa

Pochodna funkcji straty MSE względem predykcji P jest identyczna, jak w przypadku regresji liniowej (wzory (4.1) i (4.3)):

$$\frac{\partial}{\partial P} MSE = \frac{2}{n}(P - Y) \tag{4.8}$$

Ponieważ $P = M^{[2]} + b^{[2]}$ (wzory (4.2)), to

$$\frac{\partial}{\partial M^{[2]}} P = J \tag{4.9}$$

gdzie J to macierz jedynek o wymiarach zgodnych z wymiarami macierzy $M^{[2]}$.

Kolejny wzór, z którego skorzystamy to wzór na pochodną funkcji straty MSE względem $M^{[2]}$:

$$\frac{\partial}{\partial M^{[2]}} MSE = \frac{\partial}{\partial P} MSE \cdot \frac{\partial}{\partial M^{[2]}} P \tag{4.10}$$

Na tej samej podstawie, co wzór (4.8), mamy również:

$$\frac{\partial}{\partial b^{[2]}} P = 1 \tag{4.11}$$

Ponieważ $b^{[2]}$ jest skalarem, a $\frac{\partial}{\partial P} MSE$ jest macierzą o wymiarach odpowiadających liczbie próbek treningowych, to w kolejnym wzorze musimy zastosować agregację, aby uzyskać gradient również będący skalarem. Agregacją tą jest w naszym przypadku sumowanie, ponieważ gradient $\frac{\partial}{\partial P} MSE = \frac{2}{n}(P - Y)$ (wzór (4.8)) jest już uśredniony względem liczby próbek. Zatem:

$$\begin{aligned}
\frac{\partial}{\partial b^{[2]}} MSE &= \mathbf{sum}\left(\frac{\partial}{\partial P} MSE \cdot \frac{\partial}{\partial b^{[2]}} P\right) \\
&= \sum_{i=1}^{n} \left(\frac{\partial}{\partial P} MSE \cdot \frac{\partial}{\partial b^{[2]}} P\right)_i \\
&= \sum_{i=1}^{n} \left(\frac{2}{n}(P_i - Y_i) \cdot 1\right) \\
&= \frac{2}{n} \sum_{i=1}^{n} (P_i - Y_i)
\end{aligned} \tag{4.12}$$

Otrzymaliśmy wzór bardzo podobny do wzoru (4.3) dla regresji liniowej.

Kolejna pochodna, którą musimy wyliczyć, to $\frac{\partial}{\partial W^{[2]}} MSE$. Ponieważ $M^{[2]} = O^{[1]} \cdot W^{[2]}$ (wzory (4.2)), to:

$$\frac{\partial}{\partial W^{[2]}} M^{[2]} = O^{[1]T} \tag{4.13}$$

Zatem, korzystając z reguły łańcuchowej, mamy:

$$\frac{\partial}{\partial W^{[2]}} MSE = \frac{\partial}{\partial M^{[2]}} MSE \cdot \frac{\partial}{\partial W^{[2]}} M^{[2]} \tag{4.14}$$

4.2.2.2.2. Warstwa ukryta

W przypadku warstwy ukrytej będziemy korzystać z poniższych wzorów:

$$\begin{aligned}
\frac{\partial}{\partial O^{[1]}} M^{[2]} &= W^{[2]T} \\
\frac{\partial}{\partial O^{[1]}} MSE &= \frac{\partial}{\partial M^{[2]}} MSE \cdot \frac{\partial}{\partial O^{[1]}} M^{[2]} \\
\frac{\partial}{\partial N^{[1]}} O^{[1]} &= \sigma'(N^{[1]}) \\
\frac{\partial}{\partial N^{[1]}} MSE &= \frac{\partial}{\partial O^{[1]}} MSE \cdot \frac{\partial}{\partial N^{[1]}} O^{[1]} \\
\frac{\partial}{\partial B^{[1]}} N^{[1]} &= J \\
\frac{\partial}{\partial M^{[1]}} N^{[1]} &= J \\
\frac{\partial}{\partial M^{[1]}} MSE &= \frac{\partial}{\partial N^{[1]}} MSE \cdot \frac{\partial}{\partial M^{[1]}} N^{[1]} \\
\frac{\partial}{\partial W^{[1]}} M^{[1]} &= X^T \\
\frac{\partial}{\partial W^{[1]}} MSE &= \frac{\partial}{\partial M^{[1]}} MSE \cdot \frac{\partial}{\partial W^{[1]}} M^{[1]}
\end{aligned} \tag{4.15}$$

gdzie J to macierz jedynek o odpowiednich wymiarach.

W przypadku pochodnej MSE względem biasów warstwy ukrytej ($B^{[1]}$) ponownie musimy zastosować agregację, tym razem w postaci sumowania wartości z każdej z kolumn z osobna, tak aby uzyskać wektor gradientów o wymiarach zgodnych z liczbą neuronów w warstwie ukrytej.

Liczba kolumn w macierzy będącej iloczynem

$\frac{\partial}{\partial N^{[1]}} MSE \cdot \frac{\partial}{\partial B^{[1]}} N^{[1]}$

jest równa liczbie neuronów w warstwie ukrytej, a liczba wierszy - liczbie próbek treningowych.

Zatem:

$$\begin{aligned}
\frac{\partial}{\partial B^{[1]}} MSE &= \mathbf{sumByColumn}\left(\frac{\partial}{\partial N^{[1]}} MSE \cdot \frac{\partial}{\partial B^{[1]}} N^{[1]}\right) \\
&= \mathbf{sumByColumn}\left(\frac{\partial}{\partial N^{[1]}} MSE\right) \\
&= \left[\sum_{i=1}^{n}\left(\frac{\partial}{\partial N^{[1]}} MSE\right)_{i1}, \sum_{i=1}^{n}\left(\frac{\partial}{\partial N^{[1]}} MSE\right)_{i2}, \ldots, \sum_{i=1}^{n}\left(\frac{\partial}{\partial N^{[1]}} MSE\right)_{im}\right]
\end{aligned} \tag{4.16}$$

gdzie m to liczba neuronów w warstwie ukrytej.

4.2.2.3. Implementacja w C#

Programistycznie przekłada się to na poniższy kod (w komentarzach podano wymiary poszczególnych macierzy):

```
// == The second layer (output) ==

// [nTrain, 1]
float[,] dLdP = errors.Multiply(2.0f / nTrain);

// [nTrain, 1]
float[,] dPdM2 = M2.AsOnes();

// [nTrain, 1]
float[,] dLdM2 = dLdP.MultiplyElementwise(dPdM2);

float dPdBias2 = 1;

// mean([nTrain, 1]) -> scalar
float dLdBias2 = dLdP.Multiply(dPdBias2).Sum();

// [HiddenLayerSize, nTrain]
float[,] dM2dW2 = O1.Transpose();

// [HiddenLayerSize, 1]
float[,] dLdW2 = dM2dW2.MultiplyDot(dLdP);

// == The first layer (hidden) ==

// [1, HiddenLayerSize]
float[,] dM2dO1 = W2.Transpose();

// [nTrain, HiddenLayerSize]
float[,] dLdO1 = dLdM2.MultiplyDot(dM2dO1);

// [nTrain, HiddenLayerSize]
float[,] dO1dN1 = N1.SigmoidDerivative();

// [nTrain, HiddenLayerSize]
float[,] dLdN1 = dLdO1.MultiplyElementwise(dO1dN1);

// [HiddenLayerSize]
float[] dN1dBias1 = B1.AsOnes();

// [nTrain, HiddenLayerSize]
float[,] dN1dM1 = M1.AsOnes();

// [HiddenLayerSize]
float[] dLdBias1 = dN1dBias1.MultiplyElementwise(dLdN1).SumByColumn();
```

```
// [nTrain, HiddenLayerSize]
float[,] dLdM1 = dLdN1.MultiplyElementwise(dN1dM1);

// [inputFeatureCount, nTrain]
float[,] dM1dW1 = XTrainT;

// [inputFeatureCount, HiddenLayerSize]
float[,] dLdW1 = dM1dW1.MultiplyDot(dLdM1);

// Update parameters
W1 = W1.Subtract(dLdW1.Multiply(LearningRate));
W2 = W2.Subtract(dLdW2.Multiply(LearningRate));
B1 = B1.Subtract(dLdBias1.Multiply(LearningRate));
b2 -= dLdBias2 * LearningRate;
```

Listing 4.4. Aktualizacja wag i biasów sieci neuronowej za pomocą metody spadku gradientowego

gdzie:

- `nTrain` to liczba próbek treningowych,
- `HiddenLayerSize` to liczba neuronów w warstwie ukrytej,
- `inputFeatureCount` to liczba cech wejściowych.

4.3. Jakość predykcji

Na koniec porównajmy jakość predykcji obu modeli na danych testowych czyli tych, które nie były wykorzystywane podczas treningu. W tym celu uruchomiliśmy oba modele na tych samych danych i obliczyliśmy MSE.

4.3.1. Wieloraka regresja liniowa

Poniżej, jako przypomnienie z poprzedniego rozdziału, przedstawiono efekt działania modelu wielorakiej regresji liniowej na danych testowych:

```
Iteration:  26000 | MSE: 19,46206 | a1:  -0,9306 | a2:   1,3328 | a3:  -0,4534 | ... | b:  22,4711
Iteration:  28000 | MSE: 19,45545 | a1:  -0,9379 | a2:   1,3447 | a3:  -0,4380 | ... | b:  22,4702
Iteration:  30000 | MSE: 19,45045 | a1:  -0,9440 | a2:   1,3545 | a3:  -0,4242 | ... | b:  22,4695
Iteration:  32000 | MSE: 19,44664 | a1:  -0,9491 | a2:   1,3628 | a3:  -0,4120 | ... | b:  22,4688
Iteration:  34000 | MSE: 19,44374 | a1:  -0,9534 | a2:   1,3698 | a3:  -0,4013 | ... | b:  22,4682
Iteration:  36000 | MSE: 19,44154 | a1:  -0,9570 | a2:   1,3757 | a3:  -0,3917 | ... | b:  22,4677
Iteration:  38000 | MSE: 19,43987 | a1:  -0,9601 | a2:   1,3807 | a3:  -0,3834 | ... | b:  22,4673
Iteration:  40000 | MSE: 19,43856 | a1:  -0,9628 | a2:   1,3851 | a3:  -0,3760 | ... | b:  22,4669
Iteration:  42000 | MSE: 19,43758 | a1:  -0,9651 | a2:   1,3888 | a3:  -0,3695 | ... | b:  22,4666
Iteration:  44000 | MSE: 19,43682 | a1:  -0,9671 | a2:   1,3920 | a3:  -0,3639 | ... | b:  22,4663
Iteration:  46000 | MSE: 19,43625 | a1:  -0,9688 | a2:   1,3948 | a3:  -0,3589 | ... | b:  22,4661
Iteration:  48000 | MSE: 19,43581 | a1:  -0,9702 | a2:   1,3972 | a3:  -0,3545 | ... | b:  22,4658

--- Training Complete (Matrices with Bias on Boston Data) ---
Learned parameters:
 a1  =  -0,9702
 a2  =   1,3972
 a3  =  -0,3545
 a4  =   0,5873
 a5  =  -2,0943
 a6  =   1,9746
 a7  =   0,1827
 a8  =  -3,4413
 a9  =   2,7323
 a10 =  -2,3118
 a11 =  -1,8305
 a12 =   0,4663
 a13 =  -3,8503
 b   =  22,4658

Sample predictions vs actual values:

     Sample No     Predicted        Actual

             1       17,5248       20,2000
             2       17,6483       20,6000
             3       21,4478       18,6000
           150       25,8629       50,0000
           151       14,5508       14,5000
           152       34,3006       33,8000

MSE on test data: 29,49182
Elapsed time: ~0,65 seconds.
```

Rysunek 4.3. Wynik działania modelu wielorakiej regresji liniowej na danych testowych

Jak widzimy, model osiągnął MSE równe 29,49182, przy obliczeniach trwających 0,65 sekundy.

W ostatniej iteracji uczenia (48000) MSE dla danych treningowych wynosiło 19,43581.

4.3.2. Sieć neuronowa

W przypadku sieci neuronowej efekt działania na tych samych danych treningowych i testowych wygląda jak poniżej:

```
Iteration:  48000 | MSE:  7,71858

--- Training Complete (Neural Network on Boston Data) ---
Learned parameters:
Weights for the first layer (W1):
 -0,6213  -0,8456  -0,8064   0,0426
  0,0684   1,2166  -0,1887   0,8967
  0,6973   0,2085   0,0345  -0,7023
  0,6386   0,2052  -0,4259   0,1168
  0,8662  -0,1327  -1,0424  -1,2038
  2,9202  -1,2509   1,1216  -0,4074
 -0,2565  -1,0757   0,1375   0,7446
  1,1436  -0,8656  -1,0794  -2,1168
 -0,2182   0,2973   2,2307   2,1693
 -0,0283  -1,1168  -1,0311  -0,2428
  0,2698  -0,5135  -0,1872  -1,3896
  0,0644   0,8340  -0,1082  -0,0994
 -1,3172  -1,6076  -1,2015  -2,7930
Biases for the first layer (B1):
 B1[0] =  -4,6616
 B1[1] =   2,4061
 B1[2] =  -1,2612
 B1[3] =  -5,2701
Weights for the second layer (W2):
 11,0283
 10,0692
  9,9197
 11,2849
Bias for the second layer (b2):   9,3614

Sample predictions vs actual values:

     Sample No     Predicted        Actual

             1       19,7906       20,2000
             2       21,1270       20,6000
             3       16,6064       18,6000
           150       27,4527       50,0000
           151       14,7714       14,5000
           152       34,3744       33,8000

MSE on test data: 17,43655
Elapsed time: ~3,75 seconds.
```

Rysunek 4.4. Wynik działania modelu sieci neuronowej na danych testowych

Możemy odczytać następujące informacje:

- MSE na danych testowych wyniósł 17,43655
- MSE dla ostatniej iteracji na danych treningowych wyniósł 7,71858
- czas obliczeń to 3,75 sekundy.

Oprócz tego mamy również wyświetlone wagi obliczone dla wartwy ukrytej (W1) i wyjściowej (W2) oraz biasy dla obu warstw (B1 i b2).

Podsumowując, sieć neuronowa osiągnęła lepszą jakość predykcji (niższe MSE) na danych testowych w porównaniu do modelu wielorakiej regresji liniowej. Jednakże czas obliczeń był dłuższy ze względu na bardziej złożoną strukturę modelu i dodatkowe operacje związane z funkcją aktywacji i propagacją wsteczną (oraz potwornie niezoptymalizowany kod 🤪).

Zauważmy też, że w obu modelach występuje spora różnica pomiędzy MSE obliczanym na danych treningowych względem danych testowych.

4.4. Uproszczenie obliczeń

Na listingu 4.4 staraliśmy się przedstawić kod, który jest jak najbardziej zbliżony do wzorów matematycznych, na których został oparty. Wiele obliczeń można jednak uprościć, otrzymując następujący fragment kodu:

```
// The second layer (output)
float[,] dLdP = errors.Multiply(twoOverN);
float dLdBias2 = dLdP.Sum();
float[,] dLdW2 = O1.Transpose().MultiplyDot(dLdP);

// The first layer (hidden)
float[,] dLdO1 = dLdP.MultiplyDot(W2.Transpose());
float[,] dLdN1 = dLdO1.MultiplyElementwise(N1.SigmoidDerivative());
float[] dLdBias1 = dLdN1.SumByColumn();
float[,] dLdW1 = XTrainT.MultiplyDot(dLdN1);
```

z wyprowadzoną poza pętlę uczącą zmienną:

```
float twoOverN = 2.0f / nTrain;
```

Listing 4.5. Uproszczony kod obliczania gradientów sieci neuronowej

Efekt działania powyższej procedury przedstawiono na poniższym rysunku.

```
Iteration:  46000 | MSE:  7,77237
Iteration:  48000 | MSE:  7,71858

--- Training Complete (Simplified Neural Network on Boston Data) ---
Learned parameters:
Weights for the first layer (W1):
 -0,6213  -0,8456  -0,8064   0,0426
  0,0684   1,2166  -0,1887   0,8967
  0,6973   0,2085   0,0345  -0,7023
  0,6386   0,2052  -0,4259   0,1168
  0,8662  -0,1327  -1,0424  -1,2038
  2,9202  -1,2509   1,1216  -0,4074
 -0,2565  -1,0757   0,1375   0,7446
  1,1436  -0,8656  -1,0794  -2,1168
 -0,2182   0,2973   2,2307   2,1693
 -0,0283  -1,1168  -1,0311  -0,2428
  0,2698  -0,5135  -0,1872  -1,3896
  0,0644   0,8340  -0,1082  -0,0994
 -1,3172  -1,6076  -1,2015  -2,7930
Biases for the first layer (B1):
 B1[0] =  -4,6616
 B1[1] =   2,4061
 B1[2] =  -1,2612
 B1[3] =  -5,2701
Weights for the second layer (W2):
 11,0283
 10,0692
  9,9197
 11,2849
Bias for the second layer (b2):   9,3614

Sample predictions vs actual values:

     Sample No     Predicted        Actual

             1       19,7906       20,2000
             2       21,1270       20,6000
             3       16,6064       18,6000
           150       27,4527       50,0000
           151       14,7714       14,5000
           152       34,3744       33,8000

MSE on test data: 17,43655
Elapsed time: ~2,80 seconds.
```

Rysunek 4.5. Wynik działania uproszczonego modelu sieci neuronowej na danych testowych

Wszystkie wyliczone parametry pozostały bez zmian, natomiast czas obliczeń skrócił się z 3,75 do 2,8 sekundy.

4.5. Podsumowanie

W tym rozdziale utworzyliśmy prostą sieć neuronową z jedną warstwą ukrytą i porównaliśmy ją z modelem wielorakiej regresji liniowej. Omówiliśmy sposób generowania predykcji przez oba modele oraz obliczania funkcji straty MSE. Przeanalizowaliśmy również proces aktualizacji wag i biasów w obu modelach za pomocą metody spadku gradientowego, wykorzystując regułę łańcuchową do obliczenia niezbędnych pochodnych w przypadku sieci neuronowej. Na koniec porównaliśmy jakość predykcji obu modeli na danych testowych, zauważając, że sieć neuronowa osiągnęła lepsze wyniki kosztem dłuższego czasu obliczeń. Przedstawiliśmy także uproszczony kod obliczania gradientów dla sieci neuronowej, który poprawił wydajność bez zmiany wyników.

W następnym rozdziale spróbujemy opakować poznane elementy w bibliotekę *C# NeuralNetworks* i sprawdzić w jaki sposób za jej pomocą można zbudować i wytrenować sieć w oparciu o dane Boston Housing.

Created: 2025-11-15

Last modified: 2025-12-22

Title: **4. Pierwsza sieć neuronowa**

Tags: [C#] [Sieci neuronowe] [Regresja liniowa] [Macierze]

5. Biblioteka NeuralNetworks

Każdorazowe implementowanie sieci neuronowej od podstaw, tak jak to zostało przedstawione na listingach 4.2 i 4.5 w poprzednim rozdziale, byłoby niepraktyczne. Dlatego też w kolejnych rozdziałach będziemy posługiwać się specjalizowaną biblioteką (o nazwie *NeuralNetworks*), służącą do definiowania i trenowania modeli sieci neuronowych oraz do przeprowadzania procesu wnioskowania (inferencji) z użyciem tych modeli.

> **NOTE**
>
> Kod źródłowy omawianej biblioteki znajduje się na GitHub. Dostępna jest również jej dokumentacja.
>
> Ponadto, projekt NeuralNetworksExamples zawiera przykładowe procedury wykorzystujące tę bibliotekę.

A pod koniec tego rozdziału ponownie spróbujemy rozwiązać problem przewidywania cen domów w zbiorze Boston Housing, tym razem korzystając z omawianej biblioteki.

5.1. Struktura biblioteki

Biblioteka *NeuralNetworks* składa się z trzech części. Są to:

1. część ogólna, zawierająca elementy wspólne dla całej biblioteki (NeuralNetworks.Core),
2. część związana z definicją architektury modelu sieci neuronowej (NeuralNetworks.Models, NeuralNetworks.Layers, NeuralNetworks.Operations, NeuralNetworks.Losses),
3. część związana z trenowaniem modelu (NeuralNetworks.Trainers, NeuralNetworks.Optimizers, NeuralNetworks.DataSources, NeuralNetworks.LearningRates, NeuralNetworks.ParamInitializers).

Każdą z tych części omówimy w kolejnych podpunktach.

5.2. Część ogólna

5.2.1. ArrayExtensions

Jednym z podstawowych elementów biblioteki jest klasa statyczna ArrayExtensions, implementująca przydatne - w kontekście naszej biblioteki - operacje na macierzach. Operacje te można podzielić na następujące grupy:

1. tworzenie i inicjalizacja macierzy (np. AsZeros, AsZeroOnes, RandomInPlace),
2. operacje arytmetyczne na macierzach (np. Add, AddInPlace, Multiply, MultiplyDot, MultiplyElementwise),
3. agregacje i statystyki (np. Sum, Mean, ArgMax, StdDev),

4. selekcja danych i manipulacja macierzami (np. GetRow, SetRow, Transpose),
5. normalizacja danych (np. Standardize)
6. funkcje mogące pełnić role funkcji aktywacji (np. Sigmoid, Tanh, Softmax (nie jest to "typowa" funkcja aktywacji, ponieważ działa na całym wektorze jednocześnie)),
7. permutacja i operacje pomocnicze (np. ClipInPlace, PermuteInPlace).

5.2.2. OperationBackend

OperationBackend jest to klasa statyczna, która pełni rolę rozdzielnika pomiędzy konkretnymi implementacjami interfejsu IOperations. W ten sposób możemy wybierać konkretną implementację, która będzie wykorzystywana przez bibliotekę *NeuralNetworks* do wykonywania operacji numerycznych podczas trenowania lub uruchamiania danego modelu sieci neuronowej.

Interfejs `IOperations` definiuje zestaw operacji macierzowych niezbędnych do działania sieci, takich jak mnożenie macierzy, dodawanie biasów, funkcje aktywacji, operacje konwolucyjne, itp. Różne implementacje tego interfejsu mogą być zoptymalizowane pod kątem różnych scenariuszy użycia, takich jak wydajność na CPU, wykorzystanie GPU, czy minimalizacja zużycia pamięci.

Obecnie zaimplentowane są (częściwo lub w całości) następujące zestawy operacji:

- OperationsArray - podstawowa, "naiwna" implementacja, służąca do eksperymentów/debugowania, oparta głównie o działania na tablicach `float[]`, `float[,]` oraz `float[,,,]`,
- OperationsSpan - implementacja wykorzystująca struktury Span i ReadOnlySpan do wykonywania operacji macierzowych, oferująca lepszą wydajność niż `OperationsArray`,
- OperationsSpanParallel - implementacja oparta o `Span<T>` i `ReadOnlySpan<T>`, wykorzystująca przetwarzanie równoległe (metoda Parallel.For) do dalszego przyspieszenia obliczeń,
- OperationsGpu - implementacja wykorzystująca bibliotekę ILGPU (kod źródłowy) do wykonywania operacji macierzowych na karcie graficznej (GPU).

Każda z powyższych klas dziedziczy po poprzedniej (z wyjątkiem `OperationsArray`), umożliwiając wykonywanie operacji domyślnych w razie braku implementacji danej operacji w konkretnej klasie.

Z naszych eksperymentów wynika, że o ile klasa `OperationsSpanParallel` oferuje zazwyczaj całkiem przyzwoitą wydajność w porównaniu z klasami "wolniejszymi", o tyle klasa `OperationsGpu` może przyspieszyć, ale może też spowolnić obliczenia w stosunku do `OperationsSpanParallel` w zależności od konkretnego scenariusza użycia i rozmiaru danych, niezaleznie od posiadanej karty graficznej. Dlatego zalecamy przeprowadzanie własnych testów wydajnościowych w kontekście konkretnego zastosowania.

Ustawienie odpowiedniego typu backendu odbywa się poprzez wywołanie metody statycznej Use(OperationBackendType), gdzie OperationBackendType to typ klasy implementującej interfejs `IOperations`, np.:

```
OperationBackend.Use(OperationBackendType.CpuSpansParallel);
```

> **NOTE**
>
> W bibliotece NeuralNetworks zrezygnowaliśmy z wykorzystania zewnętrznych pakietów do obsługi macierzy (jak np. Math.NET Numerics), ponieważ chcieliśmy zachować pełną kontrolę nad jej implementacją oraz utrzymać wartość edukacyjną całego projektu. Nie znaczy to, że w przyszłości nie zostaną dodane implementacje oparte na takich bibliotekach, np `OperationsMathNet`.

5.3. Definicja architektury modelu sieci neuronowej

5.3.1. Klasy opisujące model

Podstawowym elementem biblioteki jest abstrakcyjna klasa Model<TInputData, TPrediction>, która reprezentuje sieć neuronową. Klasa ta posiada metody służące do definiowania warstw i ustawiania funkcji straty, do dokonywania predykcji oraz metody wywoływane przez trenera Trainer<TInputData, TPrediction> w trakcie procesu uczenia.

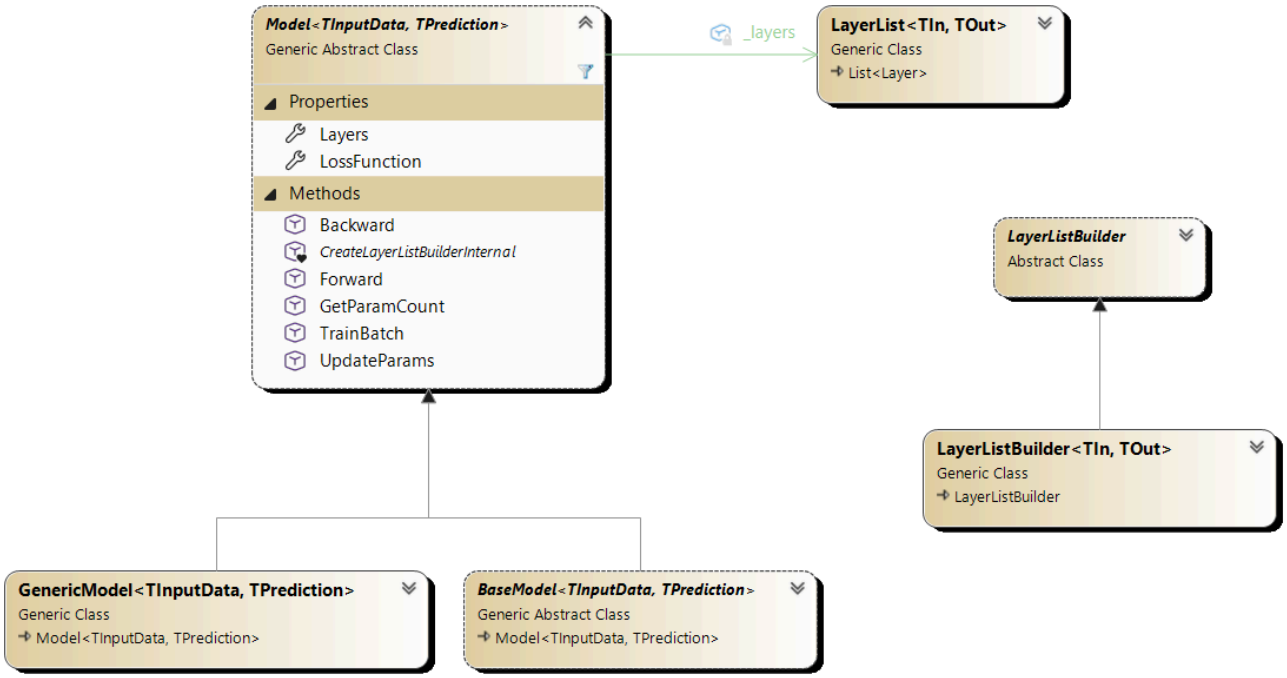
```
public abstract class Model<TInputData, TPrediction>
    where TInputData : notnull
    where TPrediction : notnull
{
    private LayerList<TInputData, TPrediction> _layers;
    private float _lastLoss;

    protected Model(LayerListBuilder<TInputData, TPrediction>? layerListBuilder,
Loss<TPrediction> lossFunction)
    {
        LossFunction = lossFunction;
        _layers = layerListBuilder.Build();
    }

    public IReadOnlyList<Layer> Layers => _layers;
    public Loss<TPrediction> LossFunction { get; }

    public TPrediction Forward(TInputData input, bool inference)
        => _layers.Forward(input, inference);

    public void Backward(TPrediction lossGrad)
        => _layers.Backward(lossGrad);

    public float TrainBatch(TInputData xBatch, TPrediction yBatch)
```

```
    {
        TPrediction predictions = Forward(xBatch, false);
        _lastLoss = LossFunction.Forward(predictions, yBatch);
        Backward(LossFunction.Backward());
        return _lastLoss;
    }

    public void UpdateParams(Optimizer optimizer)
        => _layers.UpdateParams(optimizer);

    public int GetParamCount()
        => _layers.Sum(l => (int?)l.GetParamCount()) ?? 0;

}
```

Listing 5.1. Szkic klasy `Model<TInputData, TPrediction>`

Klasami pochodnymi są BaseModel<TInputData, TPrediction>, przeznaczona do pokrycia przez konkretną implementację, oraz klasa GenericModel<TInputData, TPrediction>, przeznaczona do bezpośredniego użycia.

Klasy **modelu** umieszczone zostały w przestrzeni nazw NeuralNetworks.Models. Ich hierarchia została przedstawiona na poniższym diagramie klas.



Rysunek 5.1. Diagram klas modelu sieci neuronowej

Przykład użycia tych klas przedstawiono w rozdziale 5.5.1.

5.3.2. Warstwy

Biblioteka zawiera szereg gotowych **warstw** sieci neuronowych, zdefiniowanych jako klasy dziedziczące po abstrakcyjnej klasie Layer<TIn, TOut>.

```
public abstract class Layer<TIn, TOut> : Layer
    where TIn : notnull
    where TOut : notnull
{
    private TOut? _output;
    private TIn? _input;

    private OperationList<TIn, TOut>? _operations;

    protected TIn? Input => _input;

    public TOut Forward(TIn input, bool inference)
    {
        bool firstPass = _input is null;

        _input = input;
        if (firstPass)
        {
            // First pass, set up the layer.
            SetupLayer();
        }

        _output = _operations.Forward(input, inference);
        return _output;
    }

    public TIn Backward(TOut outputGradient)
    {
        TIn inputGradient = _operations.Backward(outputGradient);
        return inputGradient;
    }

    public override void UpdateParams(Optimizer optimizer)
        => _operations.UpdateParams(this, optimizer);

    protected virtual void SetupLayer()
    {
        // Build the operation list
        _operations = CreateOperationListBuilder().Build();
    }
```

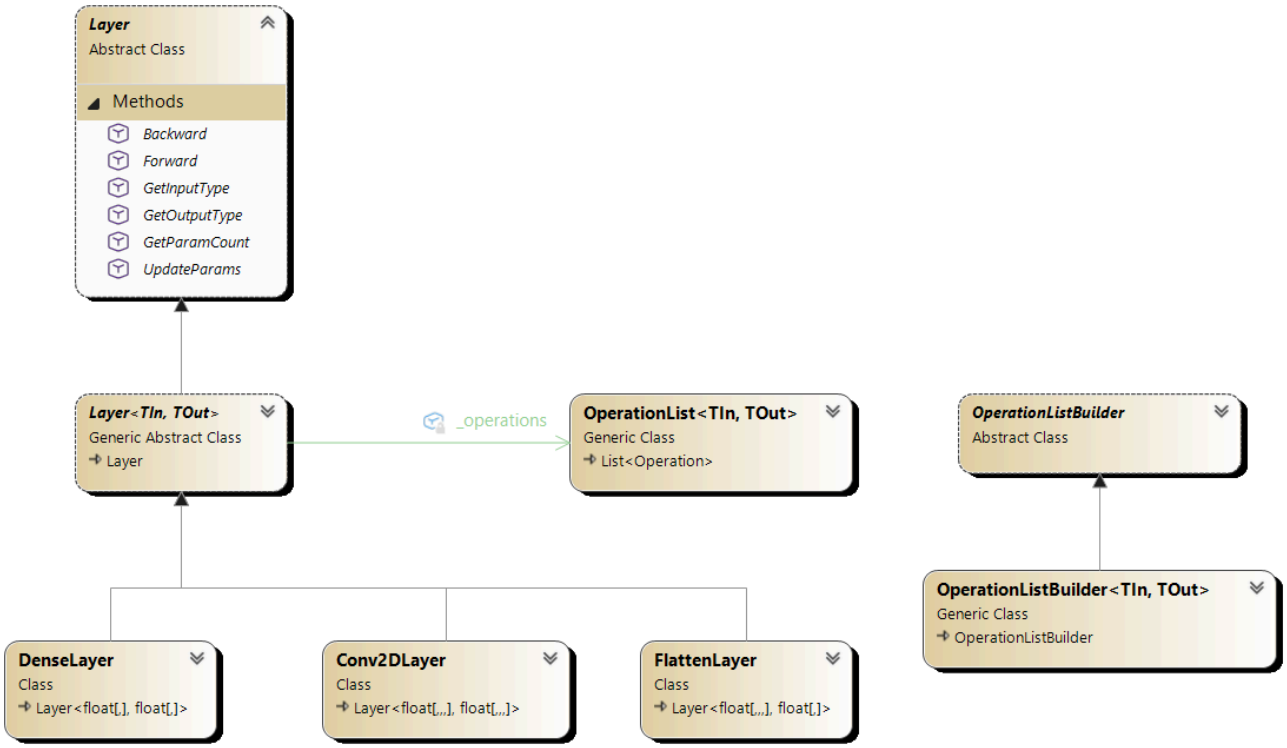
```
    public override int GetParamCount()
        => _operations.GetParamCount();

}
```

Listing 5.2. Szkic abstrakcyjnej klasy `Layer<TIn, TOut>`

Klasy opisujące warstwy znajdują się w przestrzeni nazw NeuralNetworks.Layers i obejmują między innymi:

- warstwy gęste (ang. *dense*, *fully connected*) - DenseLayer,
- warstwy konwolucyjne - Conv2DLayer,
- warstwy spłaszczające - FlattenLayer.

Poniżej przedstawiono diagram klas związanych z definiowaniem warstw sieci neuronowej.



Rysunek 5.2. Diagram klas warstw

5.3.3. Operacje

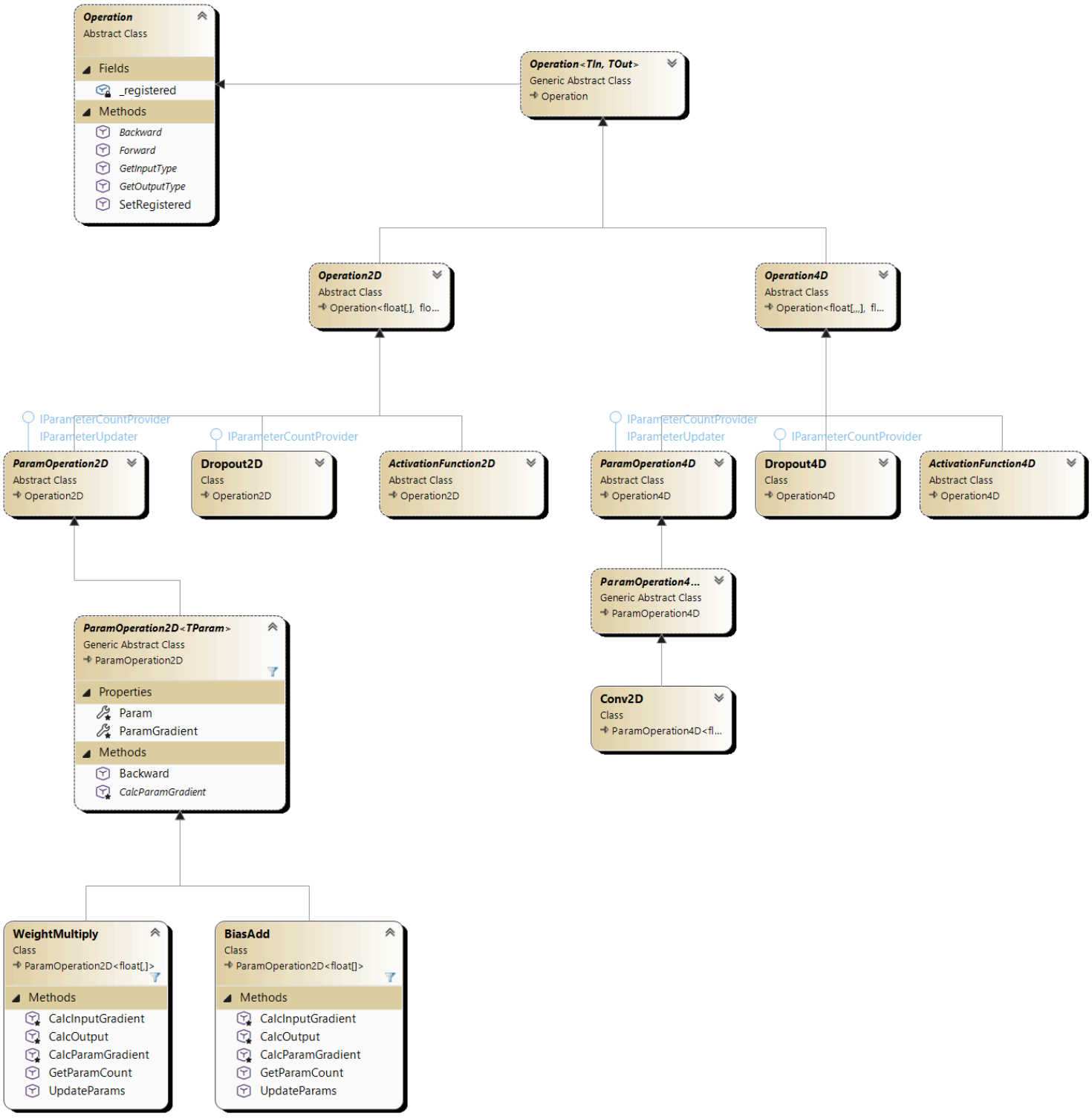
Definicja każdej warstwy obejmuje listę **operacji** (w tym funkcję aktywacji oraz dropout). Klasy implementujące operacje umieszczone są w przestrzeni nazw NeuralNetworks.Operations.

Warstwa stanowi logiczny etap przetwarzania danych, natomiast operacje są elementarnymi krokami wykonywanymi wewnątrz warstwy.

Przykładowe operacje to:

- zastosowanie wag i biasów - WeightMultiply, BiasAdd,
- funkcje aktywacji - Sigmoid, ReLU2D,
- dropout - Dropout2D,
- konwolucja - Conv2D,
- operacje spłaszczające - Flatten.

Poniżej przedstawiony jest diagram klas związanych z definiowaniem operacji.



Rysunek 5.3. Diagram klas operacji

Poniżej przedstawiono kod przykładowej operacji - `WeightMultiply` - w wersji poglądowej, bez użycia backendu.

```
public class WeightMultiply(float[,] weights) : ParamOperation2D<float[,]>(weights)
{
    protected override float[,] CalcOutput(bool inference)
        => Input.MultiplyDot(Param);

    protected override float[,] CalcInputGradient(float[,] outputGradient)
        => outputGradient.MultiplyDot(Param.Transpose());

    protected override float[,] CalcParamGradient(float[,] outputGradient)
        => Input.Transpose().MultiplyDot(outputGradient);

    public override void UpdateParams(Layer? layer, Optimizer optimizer)
        => optimizer.Update(layer, Param, ParamGradient);

    public override int GetParamCount()
        => Param.Length;
}
```

Listing 5.3a. Klasa `WeightMultiply` - wersja bez użycia backendu

Rzeczywista implementacja tej klasy jest mniej interesująca i wygląda następująco:

```
using static NeuralNetworks.Core.Operations.OperationBackend;

public class WeightMultiply(float[,] weights) : ParamOperation2D<float[,]>(weights)
{
    protected override float[,] CalcOutput(bool inference)
        => WeightMultiplyOutput(Input, Param);

    protected override float[,] CalcInputGradient(float[,] outputGradient)
        => WeightMultiplyInputGradient(outputGradient, Param);

    protected override float[,] CalcParamGradient(float[,] outputGradient)
        => WeightMultiplyParamGradient(Input, outputGradient);

    public override void UpdateParams(Layer? layer, Optimizer optimizer)
        => optimizer.Update(layer, Param, ParamGradient);

    protected override void EnsureSameShapeForParam(float[,]? param, float[,] paramGradient)
        => EnsureSameShape(param, paramGradient);

    public override int GetParamCount()
```

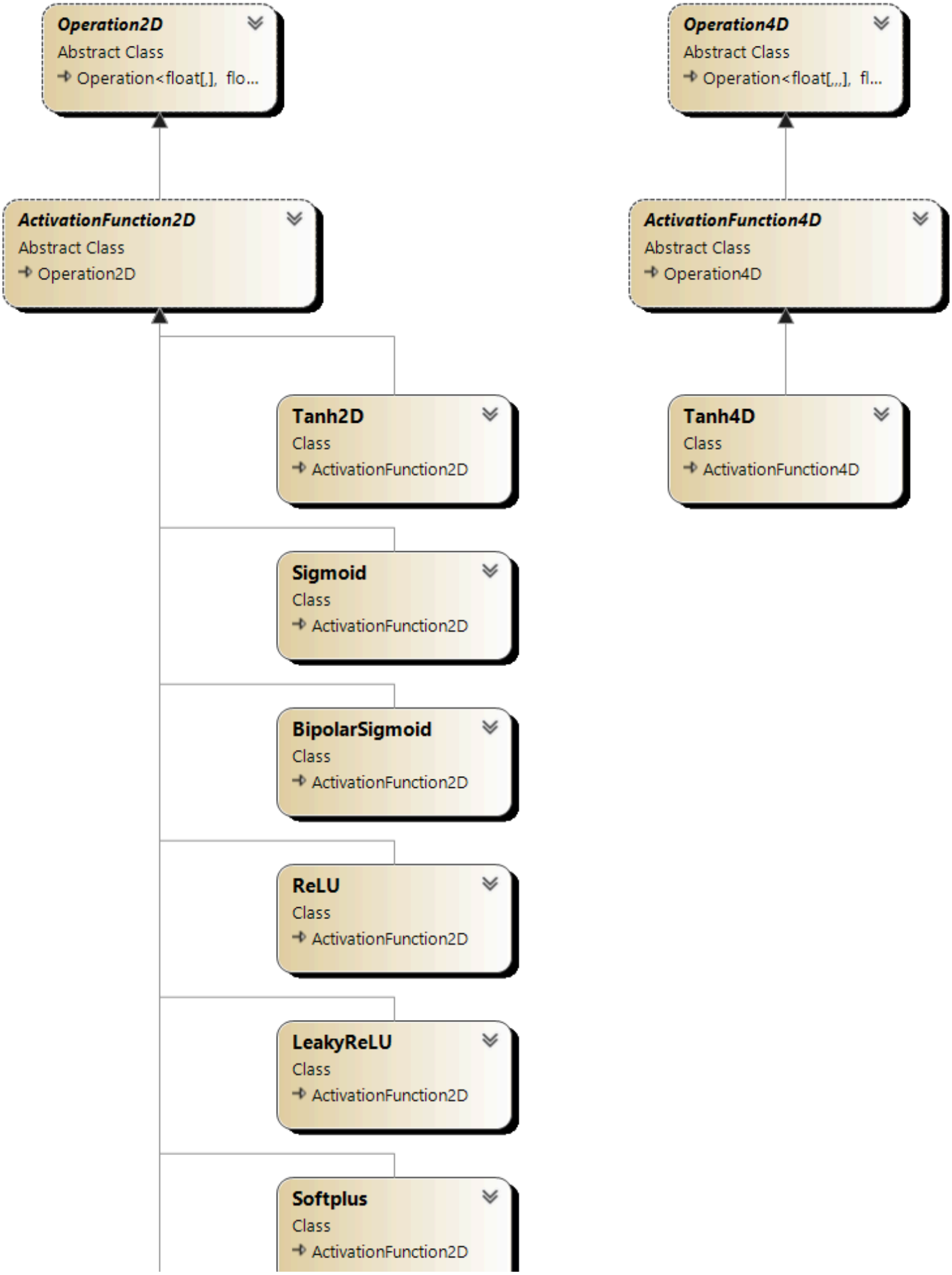
```
        => Param.Length;
}
```

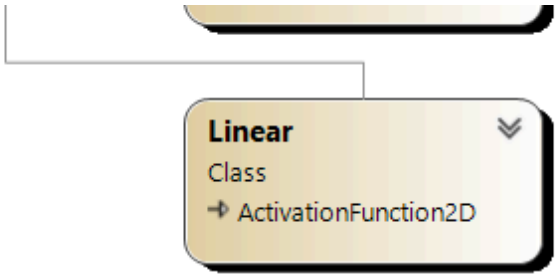
Listing 5.3b. Klasa `WeightMultiply` - wersja z użyciem backendu

5.3.3.1. Funkcje aktywacji

Część z wyżej omówionych operacji to funkcje aktywacji. W bibliotece zdefiniowano m.in. następujące, podstawowe funkcje aktywacji.

Poniżej przedstawiony został diagram klas związanych z definicjami funkcji aktywacji.


Operation2D
Abstract Class
Operation<float[,], flo...
Operation4D
Abstract Class
Operation<float[,,,], fl...
ActivationFunction2D
Abstract Class
Operation2D
ActivationFunction4D
Abstract Class
Operation4D
Tanh2D
Class
ActivationFunction2D
Tanh4D
Class
ActivationFunction4D
Sigmoid
Class
ActivationFunction2D
BipolarSigmoid
Class
ActivationFunction2D
ReLU
Class
ActivationFunction2D
LeakyReLU
Class
ActivationFunction2D
Softplus
Class
ActivationFunction2D



Rysunek 5.4. Diagram klas funkcji aktywacji

5.3.3.1.1. ReLU

Definicja matematyczna w klasycznym wydaniu to:

$$f(x) = \begin{cases} x & \text{jeśli } x \geq 0 \\ 0 & \text{jeśli } x < 0 \end{cases} \tag{5.1}$$

albo inaczej:

$$f(x) = \max(0, x) \tag{5.2}$$

My jednak zastosowaliśmy nieco zmodyfikowaną wersję, w której wynik jest skalowany przez współczynnik beta:

$$f(x) = \begin{cases} x \cdot \beta & \text{jeśli } x \geq 0 \\ 0 & \text{jeśli } x < 0 \end{cases} \tag{5.3}$$

Kod w C# w bibliotece NeuralNetworks:

```
public static float[,] ReLU(this float[,] source, float beta = 1f)
{
    int rows = source.GetLength(0);
    int columns = source.GetLength(1);
    float[,] res = new float[rows, columns];
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < columns; j++)
        {
            float value = source[i, j];
            res[i, j] = value >= 0 ? value * beta : 0;
        }
    }
    return res;
}
```

Listing 5.4. Implementacja funkcji ReLU w bibliotece NeuralNetworks

5.3.3.1.2. Leaky ReLU

Tu również nieco oddaliliśmy się od standardu. Definicja matematyczna naszej "autorskiej" wersji Leaky ReLU to:

$$f(x) = \begin{cases} x \cdot \beta & \text{jeśli } x \geq 0 \\ x \cdot \alpha & \text{jeśli } x < 0 \end{cases} \tag{5.4}$$

Umożliwi to nam szalone eksperymenty z różnymi skalami tej funkcji aktywacji.

Kod w C# w bibliotece NeuralNetworks:

```
public static float[,] LeakyReLU(this float[,] source, float alpha = 0.01f, float beta = 1f)
{
    int rows = source.GetLength(0);
    int columns = source.GetLength(1);
    float[,] res = new float[rows, columns];
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < columns; j++)
        {
            float value = source[i, j];
            res[i, j] = value >= 0 ? value * beta : value * alpha;
        }
    }
    return res;
}
```

Listing 5.5. Implementacja funkcji Leaky ReLU w bibliotece NeuralNetworks

5.3.3.1.3. Sigmoid

Definicja matematyczna:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{5.5}$$

Kod w C# w bibliotece NeuralNetworks:

```
public static float[,] Sigmoid(this float[,] source)
{
    int rows = source.GetLength(0);
    int columns = source.GetLength(1);
    float[,] res = new float[rows, columns];

    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < columns; j++)
```

```
        {
            res[i, j] = 1 / (1 + MathF.Exp(-source[i, j]));
        }
    }

    return res;
}
```

Listing 5.6. Implementacja funkcji Sigmoid w bibliotece NeuralNetworks

5.3.3.1.4. Tanh

Definicja matematyczna:

$$f(x) = \tanh(x) \tag{5.6}$$

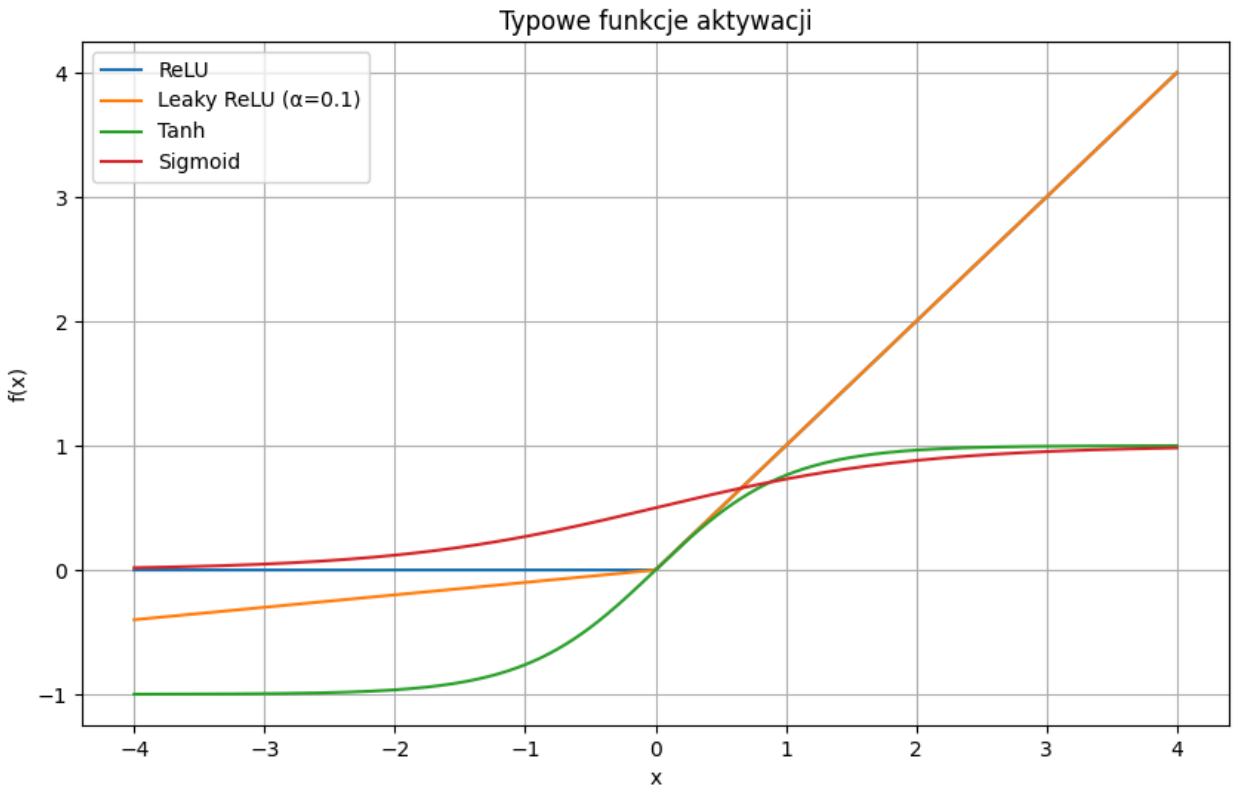
Kod w C# w bibliotece NeuralNetworks:

```
public static float[,] Tanh(this float[,] source)
{
    int rows = source.GetLength(0);
    int columns = source.GetLength(1);
    float[,] res = new float[rows, columns];

    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < columns; j++)
        {
            res[i, j] = MathF.Tanh(source[i, j]);
        }
    }

    return res;
}
```

Listing 5.7. Implementacja funkcji Tanh w bibliotece NeuralNetworks

5.3.3.1.5. Wykresy i diagram klas funkcji aktywacji

Przebiegi zaimplementowanych, typowych funkcji aktywacji przedstawiono na poniższym wykresie.



Rysunek 5.5. Wykresy funkcji aktywacji: ReLU, Leaky ReLU, Sigmoid, Tanh

5.3.4. Funkcje straty

Kompletna definicja modelu obejmuje również (oprócz listy warstw) podanie wykorzystywanej **funkcji straty** (*loss function*). Klasy implementujące funkcje straty znajdują się w przestrzeni nazw Neural Networks.Losses.

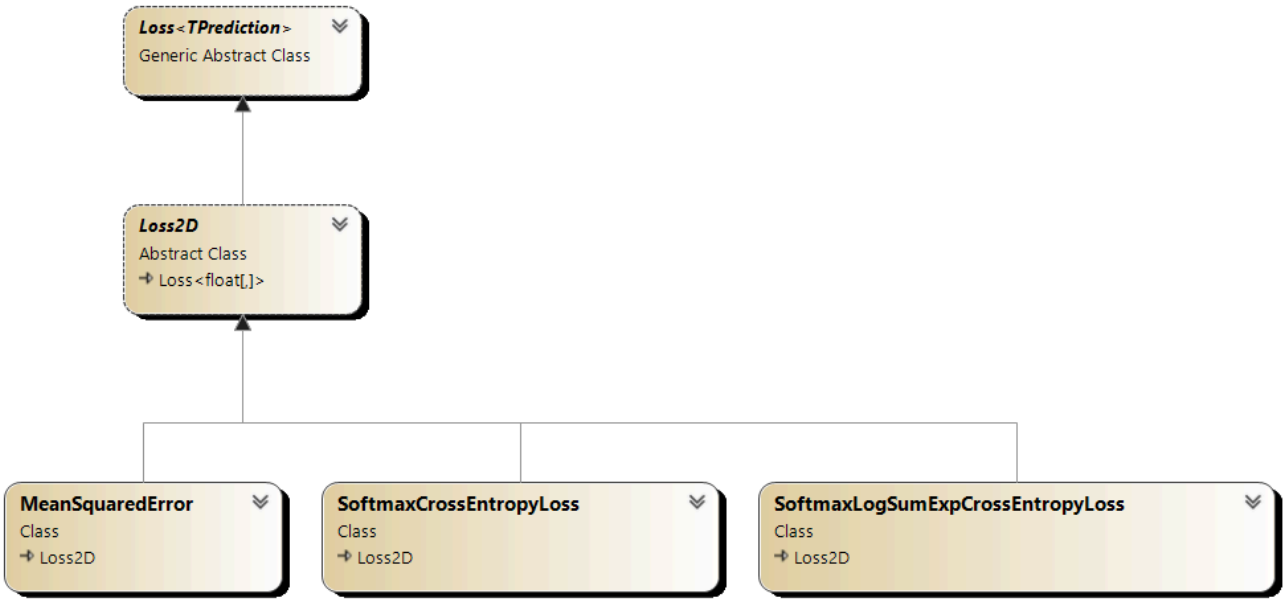
```
public abstract class Loss<TPrediction>
{
    private TPrediction? _prediction;
    private TPrediction? _target;

    public TPrediction Prediction => _prediction;
    protected internal TPrediction Target => _target;

    public float Forward(TPrediction prediction, TPrediction target)
    {
        _prediction = prediction;
        _target = target;
        return CalculateLoss();
    }

    public TPrediction Backward()
```

```
    {
        TPrediction lossGradient = CalculateLossGradient();
        return lossGradient;
    }

    protected abstract float CalculateLoss();

    protected abstract TPrediction CalculateLossGradient();
}
```

Listing 5.8. Szkic abstrakcyjnej klasy `Loss<TPrediction>`

Przykładowe funkcje straty zaimplementowane w naszej bibliotece i dziedziczące po `Loss<TPrediction>` to:

- błąd średniokwadratowy (*Mean Squared Error*) - MeanSquaredError,
- entropia krzyżowa (*Softmax Cross Entropy Loss*) - SoftmaxCrossEntropyLoss,
- entropia krzyżowa z "log-sum-exp trick" (*Softmax Log-Sum-Exp Cross Entropy Loss*) - SoftmaxLog SumExpCrossEntropyLoss.



Rysunek 5.6. Diagram klas funkcji strat

Poniżej przedstawiono dwie popularne funkcje straty.

5.3.4.1. MSE (Mean Squared Error)

O tej funkcji wspominaliśmy już w poprzednich rozdziałach (wzory 1.8 i 4.2). Jest to jedna z najprostszych i najczęściej stosowanych funkcji straty, zwłaszcza w regresji. Tytułem przypomnienia ponownie przedstawiamy jej definicję.

Definicja matematyczna:

$$
\begin{aligned}
\boldsymbol{E} &= \boldsymbol{P} - \boldsymbol{Y} \\
\boldsymbol{MSE} &= \frac{1}{\boldsymbol{n}} \sum_{\boldsymbol{i}=1}^{\boldsymbol{n}} \boldsymbol{E}_{\boldsymbol{i}}^{2}
\end{aligned}
\tag{5.7}
$$

Kod w C# w bibliotece NeuralNetworks:

```
protected override float CalculateLoss()
{
    int batchSize = Prediction.GetLength(0);
    _errors = Prediction.Subtract(Target);
    return _errors.Power(2).Sum() / batchSize;
}
```

Listing 5.9. Implementacja funkcji MSE w bibliotece NeuralNetworks

Wartości zapisane w zmiennej `_errors` będą później wykorzystane do obliczenia gradientów w metodzie `CalculateLossGradient`.

5.3.4.2. Softmax Cross-Entropy

Ta funkcja straty jest szczególnie odpowiednia dla zadań klasyfikacji wieloklasowej z pojedynczą etykietą. Łączy w sobie funkcję softmax, która przekształca wyjścia sieci w prawdopodobieństwa klas, oraz entropię krzyżową, która mierzy różnicę między przewidywanymi a rzeczywistymi etykietami klas.

Definicja matematyczna:

$$
\begin{aligned}
S_{i,j} &= \operatorname{Softmax}(P_i)_j = \frac{e^{P_{i,j}}}{\sum_{k=1}^{c} e^{P_{i,k}}} \\
SCE &= \frac{1}{n} \sum_{i=1}^{n} Y_i \cdot (-\log(S_i)) = -\frac{1}{n} \sum_{i=1}^{n} Y_i \log(S_i)
\end{aligned}
\tag{5.8}
$$

gdzie:

- $\boldsymbol{Y}$ to rzeczywiste etykiety klas (w formie one-hot, czyli same zera za wyjątkiem jednego elementu równego 1 dla prawidłowej klasy),
- $\boldsymbol{P}$ to wyjścia sieci przed zastosowaniem funkcji softmax (tzw. *logits*),
- $\boldsymbol{S}$ to przewidywane prawdopodobieństwa klas po zastosowaniu funkcji softmax (zsumowane wartości wynoszą 1 dla każdej próbki),
- $\boldsymbol{n}$ to liczba próbek w batchu,
- $\boldsymbol{c}$ to liczba klas (w przypadku MNIST jest to 10 - jedna klasa dla każdej cyfry),
- $\boldsymbol{i}$ to indeks próbki w batchu (od 1 do $\boldsymbol{n}$),

- j to indeks klasy (od 1 do c).

W skrócie, podnosimy e do potęgi predykcji sieci dla danej próbki i kategorii, następnie dzielimy przez sumę tych wartości dla wszystkich kategorii, aby uzyskać prawdopodobieństwa. Następnie logarytmujemy te prawdopodobieństwa i mnożymy przez rzeczywiste etykiety klas (wartości etykiet są równe tylko 0 lub 1), uśredniając wyniki dla wszystkich próbek.

Odpowiedni kod w bibliotece NeuralNetworks wygląda następująco:

```
protected override float CalculateLoss()
{
    _softmaxPrediction = Prediction.Softmax();
    float[,] clippedSoftmax = _softmaxPrediction.Clip(eps, 1 - eps);
    return -clippedSoftmax.Log().MultiplyElementwise(Target).Mean();
}
```

Listing 5.10. Implementacja funkcji Softmax Cross-Entropy w bibliotece NeuralNetworks

Dwie uwagi. Po pierwsze, w implementacji funkcji straty stosujemy klipowanie wartości softmax do przedziału otwartego (0, 1), aby uniknąć problemów z logarytmem zera. Po drugie, zapisujemy wartości softmax w polu `_softmaxPrediction`, ponieważ będą one potrzebne podczas obliczania gradientów w metodzie `CalculateLossGradient`.

5.3.5. Inicjalizacja wag

Wagi i biasy w warstwach sieci neuronowej muszą być odpowiednio zainicjalizowane przed rozpoczęciem procesu trenowania. W bibliotece NeuralNetworks dostępne są różne strategie inicjalizacji parametrów, zaimplementowane jako klasy dziedziczące po abstrakcyjnej klasie ParamInitializer.

Przykładowy inicjalizator wag to GlorotInitializer o następującej implementacji:

```
public class GlorotInitializer(SeededRandom? random = null) : RandomInitializer(random)
{
    internal override float[,] InitWeights(int inputColumns, int neurons)
    {
        float stdDev = MathF.Sqrt(2.0f / (inputColumns + neurons));
        return CreateRandomNormal(inputColumns, neurons, Random, 0, stdDev);
    }

    internal override float[] InitBiases(int neurons)
        => new float[neurons];
}

public static float[,] CreateRandomNormal(int rows, int columns, Random random, float mean =
```

```
0, float stdDev = 1)
{
    float[,] res = new float[rows, columns];
    for (int row = 0; row < rows; row++)
    {
        for (int col = 0; col < columns; col++)
        {
            res[row, col] = BoxMuller() * stdDev + mean;
        }
    }
    return res;

    float BoxMuller()
    {
        // uniform(0,1] random doubles
        // NextDouble returns [0,1), so to convert to (0,1], we use 1 - NextDouble()
        // Zero must be excluded because log(0) is undefined.
        double u1 = 1 - random.NextDouble();
        double u2 = 1 - random.NextDouble();

        //random normal(0,1)
        float randStdNormal = Convert.ToSingle(Math.Sqrt(-2.0 * Math.Log(u1)) * Math.Sin(2.0
* Math.PI * u2));
        return randStdNormal;
    }
}
```

Listing 5.11. Implementacja inicjalizatora Glorot w bibliotece NeuralNetworks

Matematycznie zapisalibyśmy to za pomocą następujących wzorów:

$$\begin{aligned} \sigma &= \sqrt{\frac{2}{n_{in} + n_{out}}} \\ W_{i,j} &\sim \mathcal{N}(0, \sigma^2) \\ b_j &= 0 \end{aligned} \tag{5.9}$$

gdzie

$W_{i,j}$ to waga łącząca neuron i z warstwy poprzedniej z neuronem j w bieżącej warstwie, b_j to bias neuronu j, n_{in} to liczba neuronów w warstwie poprzedniej, n_{out} to liczba neuronów w bieżącej warstwie. Symbol $\mathcal{N}(0, \sigma^2)$ oznacza rozkład normalny o średniej 0 i wariancji σ^2 (σ to odchylenie standardowe).

5.3.6. Dropout

Dropout to technika stosowana w sieciach neuronowych w celu zapobiegania przeuczeniu (*overfitting*). Polega ona na losowym "wyłączaniu" (ustawianiu na zero) pewnego odsetka neuronów podczas treningu, co zmusza sieć do nauki bardziej ogólnych cech danych. W bibliotece NeuralNetworks dropout został zaimplementowany jako operacja dziedzicząca Dropout2D.

Implementacja wygląda następująco:

```
public class Dropout2D(float keepProb = 0.8f, SeededRandom? random = null) :
Operation2D, IParameterCountProvider
{
    private float[,]? _mask;

    protected override float[,] CalcOutput(bool inference)
    {
        if (inference)
        {
            return Input.Multiply(keepProb);
        }
        else
        {
            _mask = Input.AsZeroOnes(keepProb, random ?? new());
            return Input.MultiplyElementwise(_mask);
        }
    }

    protected override float[,] CalcInputGradient(float[,] outputGradient)
    {
        return outputGradient.MultiplyElementwise(_mask);
    }

    public int GetParamCount()
        => _mask?.Length ?? 0;
}
```

Listing 5.12. Implementacja operacji Dropout2D w bibliotece NeuralNetworks

Zauważmy, że podczas inferencji (czyli predykcji) dropout nie jest stosowany - zamiast tego wyjścia są skalowane przez prawdopodobieństwo zachowania neuronu (`keepProb`), aby uwzględnić fakt, że podczas treningu część neuronów była wyłączana. Gdybyśmy nie skalowali wyjść podczas inferencji, wartości wyjściowe byłyby zawyżone w porównaniu do tych uzyskiwanych podczas treningu.

5.4. Trenowanie modelu

5.4.1. Trener

Do trenowania modelu służy klasa Trainer<TInputData, TPrediction>. Klasa ta przyjmuje jako parametry typ danych wejściowych i wyjściowych oraz posiada metody służące do trenowania modelu na podstawie dostarczonych danych treningowych. Przykładowe użycie trenera zaprezentowano w rozdziale 5.5.3.

Podstawową metodą tej klasy jest metoda `Fit`, która realizuje proces trenowania modelu. Metoda ta przyjmuje jako argumenty dostawcę danych treningowych i testowych, liczbę epok, rozmiar batcha oraz optymalizator.

Poniżej przedstawiono zasadniczą część kodu trenera (metoda `Fit`):

```
(TInputData xTrain, TPrediction yTrain, TInputData? xTest, TPrediction? yTest)
= dataSource.GetData();

for (int epoch = 1; epoch <= epochs; epoch++)
{
    PermuteData(xTrain, yTrain, random);
    optimizer.UpdateLearningRate(epoch, epochs);

    foreach ((TInputData xBatch, TPrediction yBatch) in GenerateBatches(xTrain,
yTrain, batchSize))
    {
        trainLoss = model.TrainBatch(xBatch, yBatch);
        model.UpdateParams(optimizer);
    }
}
```

Listing 5.13. Fragment kodu trenera

Implementacja metody `TrainBatch`, wywoływanej w powyższym kodzie została przedstawiona na listingu 5.1.

5.4.2. Dostarczanie danych treningowych i testowych

Do zaopatrywania trenera w dane treningowe i testowe służy klasa DataSource<TInputData, TPrediction>. Dla danych Boston Housing wykorzystaliśmy klasę dziedziczącą po tej klasie.

```
SimpleDataSource<float[,], float[,]> dataSource = new(XTrain, YTrain, XTest, YTest);
```

Listing 5.9. Definicja dostawcy danych treningowych i testowych dla danych Boston Housing

Zmienna `dataSource` jest następnie przekazywana do metody `Fit` trenera, jak pokazano na listingu 5.13.

Pozostałe klasy dostawców danych znajdują się w przestrzeni nazw NeuralNetworks.DataSources.

5.4.3. Optymalizatory i współczynniki uczenia

Do aktualizacji wag i biasów modelu podczas procesu trenowania służą **optymalizatory** (*optimizers*). Klasy implementujące optymalizatory znajdują się w przestrzeni nazw NeuralNetworks.Optimizers.

Przykładowe optymalizatory to:

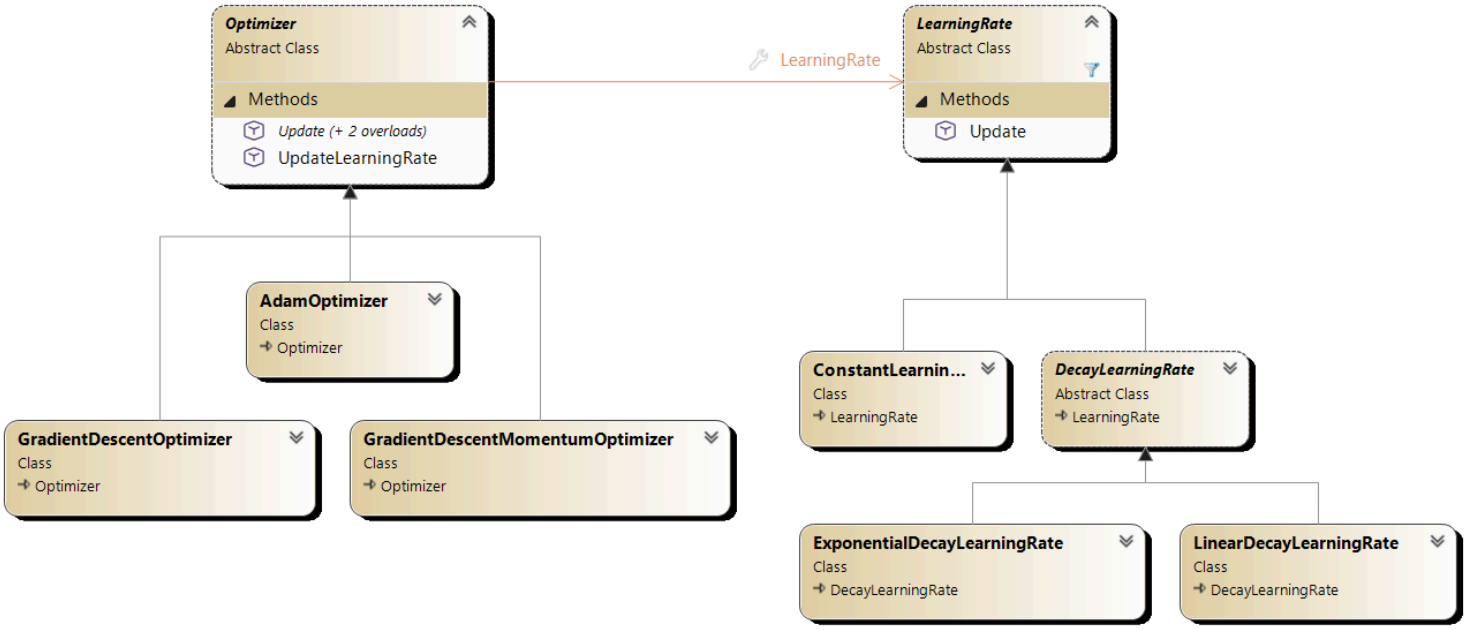
- optymalizator spadku gradientowego (*Stochastic Gradient Descent*, *SGD*) - GradientDescentOptimizer,
- optymalizator spadku gradientowego z momentem - GradientDescentMomentumOptimizer,
- optymalizator Adam - AdamOptimizer.

> **NOTE**
>
> Optymalizatory SGD zawierają w nazwie słowo "Stochastic", ale w rzeczywistości ich implementacja nie wprowadza żadnego losowego, "stochastycznego" aspektu. W założeniach losowość ta polegała na losowym wyborze próbki treningowej do obliczania gradientu w każdej iteracji. W naszej implementacji gradient jest obliczany na podstawie całego batcha (lub nawet wszystkich) próbek treningowych, co jest zgodne z podejściem zwanym *mini-batch gradient descent*. Nazwa ta jednak jest powszechnie używana w literaturze i w implementacjach.

Optymalizatory korzystają ze **współczynników uczenia** (*learning rates*), które określają, jak duże kroki mają być wykonywane podczas aktualizacji wag i biasów w kolejnych epokach. Klasy implementujące współczynniki uczenia znajdują się w przestrzeni nazw NeuralNetworks.LearningRates. Zaimplementowane zostały między innymi:

- stały współczynnik uczenia - ConstantLearningRate,
- wykładniczy spadek współczynnika uczenia - ExponentialDecayLearningRate,
- liniowy spadek współczynnika uczenia - LinearDecayLearningRate.

Diagramy klas odpowiedzialnych za optymalizację przedstawiono poniżej.



Rysunek 5.7. Diagram klas optymalizatorów i współczynników uczenia

5.4.3.1. Spadek gradientowy z momentem

Optymalizator spadku gradientowego z momentem (SGD with Momentum) został zaimplementowany w następujący sposób:

```
public class GradientDescentMomentumOptimizer(LearningRate learningRate, float momentum)
: Optimizer(learningRate)
{
    private readonly Dictionary<float[,], float[,]> _velocities2D = [];

    public override void Update(Layer? layer, float[,] param, float[,] paramGradient)
    {
        float learningRate = LearningRate.GetLearningRate();

        float[,] velocities = GetOrCreateVelocities(param);

        int dim1 = param.GetLength(0);
        int dim2 = param.GetLength(1);
        for (int i = 0; i < dim1; i++)
        {
            for (int j = 0; j < dim2; j++)
            {
                velocities[i, j] = velocities[i, j] * momentum + learningRate *
paramGradient[i, j];
                param[i, j] -= velocities[i, j];
            }
        }
    }
```

```
    private float[,] GetOrCreateVelocities(float[,] param)
    {
        if (_velocities2D.TryGetValue(param, out float[,]? velocities))
        {
            return velocities;
        }
        else
        {
            velocities = new float[param.GetLength(0), param.GetLength(1)];
            _velocities2D.Add(param, velocities);
            return velocities;
        }
    }

}
```

Listing 5.14. Implementacja optymalizatora spadku gradientowego z momentem w bibliotece NeuralNetworks

Zasadę działania tego optymalizatora możemy przedstawić za pomocą poniższych wzorów:

$$\begin{aligned} v_t &= \mu \cdot v_{t-1} + lr \cdot g_t \\ w_t &:= w_t - v_t \end{aligned} \tag{5.10}$$

gdzie

- t to numer kroku (kolejnego batcha),
- w_t to waga w kroku t (przed i po aktualizacji),
- g_t to gradient straty względem wagi w kroku t ($g_t = \nabla_w L(w_t)$),
- v_t to prędkość (moment) w kroku t (początkowo v_0 jest inicjalizowane jako 0),
- μ to współczynnik momentu (zazwyczaj ustawiany na 0.9),
- lr to współczynnik uczenia dla danej epoki.

Jeżeli współczynnik momentu μ jest ustawiony na 0, optymalizator ten sprowadza się do klasycznego spadku gradientowego.

5.4.3.2. Adam

Nieco bardziej złożony jest optymalizator Adam (Adaptive Moment Estimation). Jego implementacja w C# w bibliotece NeuralNetworks wygląda następująco (pokazano jedynie wybrane metody dla tablic 2D):

```
public class AdamOptimizer : Optimizer
{
    private readonly float _beta1;
```

```
    private readonly float _beta2;
    private readonly float _eps;

    private readonly Dictionary<float[,], State2D> _states2D = [];

    public AdamOptimizer(LearningRate learningRate, float beta1 = 0.9f, float beta2 =
0.999f, float eps = 1e-8f)
        : base(learningRate)
    {
        _beta1 = beta1;
        _beta2 = beta2;
        _eps = eps;
    }

    public override void Update(Layer? layer, float[,] param, float[,] paramGradient)
    {
        (int t, float[,] m, float[,] v) = GetOrCreateState(param);

        float beta1t = MathF.Pow(_beta1, t);
        float beta2t = MathF.Pow(_beta2, t);

        float lr = LearningRate.GetLearningRate();

        int dim1 = param.GetLength(0);
        int dim2 = param.GetLength(1);

        for (int i = 0; i < dim1; i++)
        {
            for (int j = 0; j < dim2; j++)
            {
                m[i, j] = _beta1 * m[i, j] + (1 - _beta1) * paramGradient[i, j];
                v[i, j] = _beta2 * v[i, j] + (1 - _beta2) * paramGradient[i, j] *
paramGradient[i, j];

                float mHat = m[i, j] / (1 - beta1t);
                float vHat = v[i, j] / (1 - beta2t);

                param[i, j] -= lr * mHat / (MathF.Sqrt(vHat) + _eps);
            }
        }
    }

    private State2D GetOrCreateState(float[,] param)
    {
        if (_states2D.TryGetValue(param, out State2D? state))
        {
```

```
            state.T++;
            return state;
        }
        var newState = new State2D(param);
        _states2D[param] = newState;
        return newState;
    }

    private sealed class State2D
    {
        public int T { get; set; } = 1;
        public float[,] M { get; }
        public float[,] V { get; }

        public State2D(float[,] param)
        {
            int rows = param.GetLength(0);
            int cols = param.GetLength(1);
            M = new float[rows, cols];
            V = new float[rows, cols];
        }

        public void Deconstruct(out int t, out float[,] m, out float[,] v)
        {
            t = T;
            m = M;
            v = V;
        }
    }
}
```

Listing 5.15. Implementacja optymalizatora Adam w bibliotece NeuralNetworks

Zasadę działania optymalizatora Adam możemy przedstawić za pomocą poniższych wzorów:

$$
\begin{aligned}
m_t &= \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \\
v_t &= \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \\
\hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1 - \beta_2^t} \\
w_t &:= w_t - lr \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \\
t &:= t + 1
\end{aligned}
\tag{5.11}
$$

gdzie

- t to numer kroku (kolejnego batcha) (początkowo $t = 1$),
- w_t to waga w kroku t (przed i po aktualizacji),
- g_t to gradient straty względem wagi w kroku t ($g_t = \nabla_w L(w_t)$),
- m_t to pierwszy moment (średnia krocząca gradientów) w kroku t (m_0 jest inicjalizowane jako 0),
- v_t to drugi moment (średnia krocząca kwadratów gradientów) w kroku t (v_0 jest inicjalizowane jako 0),
- $\hat{m}_t$ to skorygowany pierwszy moment w kroku t,
- $\hat{v}_t$ to skorygowany drugi moment w kroku t,
- β_1 i β_2 to współczynniki wygładzania (zazwyczaj ustawiane na 0.9 i 0.999),
- lr to współczynnik uczenia dla danej epoki,
- ϵ to mała stała dodawana do mianownika w celu uniknięcia dzielenia przez zero (zazwyczaj ustawiana na 1e-8).

Częściowe objaśnienie zasady działania tego optymalizatora można znaleźć w na tej stronie. Oryginalny artykuł znajduje się na ArXiv.

5.5. Zastosowanie biblioteki do analizy danych Boston Housing

W poprzednim rozdziale utworzyliśmy prostą sieć neuronową przeznaczoną do przewidywania cen domów na podstawie danych ze zbioru Boston Housing. Spróbujmy więc ponownie przeanalizować dane z tego zbioru, tym razem korzystając z biblioteki *NeuralNetworks*.

> **NOTE**
>
> Poniższy kod w pełnej wersji znajduje się na GitHub

5.5.1. Definicja modelu

Model sieci neuronowej możemy zdefiniować poprzez utworzenie klasy dziedziczącej po BaseModel<TInputData, TPrediction> i nadpisanie metody `CreateLayerListBuilder`, w której określamy strukturę sieci (liczbę warstw, liczbę neuronów w każdej warstwie oraz funkcje aktywacji). W naszym przypadku model będzie miał jedną warstwę ukrytą z czterema neuronami i funkcją aktywacji tanh oraz warstwę wyjściową z jednym neuronem i funkcją liniową. W naszej bibliotece nie musimy jawnie deklarować warstwy wejściowej, ponieważ jest ona implikowana przez kształt danych wejściowych.

```
class BostonHousingModel(SeededRandom? random)
    : BaseModel<float[,], float[,]>(new MeanSquaredError(), random)
{
```

```
    protected override LayerListBuilder<float[,], float[,]> CreateLayerListBuilder()
    {
        GlorotInitializer initializer = new(Random);

        return AddLayer(new DenseLayer(4, new Tanh2D(), initializer))
            .AddLayer(new DenseLayer(1, new Linear(), initializer));
    }
}
```

Listing 5.16. Definicja modelu sieci neuronowej do przewidywania cen w zbiorze Boston Housing

5.5.2. Dane źródłowe

```
SimpleDataSource<float[,], float[,]> dataSource = new(XTrain, YTrain, XTest, YTest);
```

Tablice `XTrain`, `YTrain`, `XTest` i `YTest` zostały przygotowane w sposób analogiczny do przedstawionego na listingach 3.1 i 3.2. Między innymi zostały one poddane normalizacji za pomocą procedury Standardize.

5.5.3. Trenowanie modelu

Trenowanie modelu odbywa się poprzez utworzenie instancji klasy Trainer<TInputData, TPrediction>, przekazanie do niej modelu i optymalizatora, a następnie wywołanie metody `Fit` wraz z dostawcą danych i parametrami uczenia/logowania, jak pokazano poniżej.

```
BostonHousingModel model = new(commonRandom);

ExponentialDecayLearningRate learningRate = new(
    initialLearningRate: 0.0009f,
    finalLearningRate: 0.0005f
);

Trainer2D trainer = new(
    model,
    new GradientDescentMomentumOptimizer(learningRate, 0.9f),
    random: commonRandom,
    logger: logger
);

trainer.Fit(
    dataSource,
    epochs: 48_000,
    evalEveryEpochs: 2_000,
    logEveryEpochs: 2_000,
```

```
    batchSize: 400
);
```

Listing 5.17. Trenowanie modelu sieci neuronowej do przewidywania cen w zbiorze Boston Housing

5.5.4. Rezultat trenowania

Po zakończeniu trenowania modelu uzyskaliśmy wyniki przedstawione na poniższej ilustracji.

```
Train loss (average): 5,783022
Test loss: 15,191841

Epoch 40000/48000...
Current learning rate: 0,00055146293.
Train loss (average): 5,7712727
Test loss: 15,190115

Epoch 42000/48000...
Current learning rate: 0,00053812074.
Train loss (average): 5,761054
Test loss: 15,195444

Epoch 44000/48000...
Current learning rate: 0,0005251014.
Train loss (average): 5,7518935
Test loss: 15,208461

Epoch 46000/48000...
Current learning rate: 0,00051239703.
Train loss (average): 5,7434587
Test loss: 15,231816

Epoch 48000/48000...
Current learning rate: 0,0005.
Train loss (average): 5,735411
Test loss: 15,269817

Fit finished in 8,68 s.
61 parameters trained.

Loss on test data: 15,26982


Sample predictions vs actual values:

     Sample No     Predicted        Actual

             1       16,3194       20,2000
             2       18,9333       20,6000
             3       16,6895       18,6000
           150       38,4352       50,0000
```

Rysunek 5.8. Rezultat działania modelu sieci neuronowej do przewidywania cen w zbiorze Boston Housing

Ponieważ jest to już nasze ostatnie spotkanie z danymi Boston Housing, pozwolimy sobie na krótkie podsumowanie wyników.

Metoda	Funkcja aktywacji	Optymalizator	MSE na zbiorze treningowym	MSE na zbiorze testowym	Czas treningu [s]
Regresja liniowa (rozdział 3)	Linear	SGD	19,44	29,49	0,87
Pierwsza sieć neuronowa (rozdział 4)	Sigmoid	SGD	7,72	17,44	4,37
Biblioteka *NeuralNetworks* (rozdział 5)	Tanh	SGD z momentum	5,74	15,27	8,23

Tabela 5.1. Porównanie wyników różnych metod na danych Boston Housing (48 tys. epok)

5.6. Dodatek

Wybrane pojęcia związane z sieciami neuronowymi i uczeniem maszynowym, które pojawiły się w tym rozdziale, zostały wyjaśnione poniżej.

5.6.1. Logits

Logits stanowią wyjście sieci przed aktywacją. Nie są prawdopodobieństwami (mogą być ujemne, nie sumują się do 1). Są podstawą do obliczania strat i decyzji modelu. Softmax / sigmoid zamieniają logits na prawdopodobieństwa.

5.6.2. Normalizacja danych

Normalizacja danych to proces skalowania cech wejściowych do określonego zakresu lub rozkładu. Pomaga to w stabilizacji i przyspieszeniu procesu trenowania sieci neuronowej. Popularne metody normalizacji to:

- Min-Max Scaling: Skalowanie cech do zakresu [0, 1] lub [-1, 1].
- Standaryzacja (Z-score Normalization): Przekształcanie cech do rozkładu o średniej 0 i odchyleniu standardowym 1.

5.6.3. Epoka i krok

- Epoka (*epoch*): Pełne przejście przez cały zbiór treningowy podczas trenowania modelu.
- Krok (*step*): Pojedyncza aktualizacja wag modelu na podstawie jednej partii danych (batcha).

5.6.4. Batch i rozmiar batcha

- Batch: Podzbiór danych treningowych używany do jednej aktualizacji wag modelu.
- Rozmiar batcha (*batch size*): Liczba próbek w jednym batchu. Wpływa na stabilność i szybkość trenowania.

5.6.5. Odchylenie standardowe i wariancja

- Odchylenie standardowe (*standard deviation*): Miara rozproszenia danych wokół średniej. Oblicza się je jako pierwiastek kwadratowy z wariancji.
- Wariancja (*variance*): Średnia z kwadratów odchyleń poszczególnych wartości od średniej. Mierzy, jak bardzo dane są rozproszone.

5.7. Podsumowanie

W tym rozdziale przedstawiliśmy bibliotekę *NeuralNetworks*, która umożliwia definiowanie, trenowanie i wykorzystywanie modeli sieci neuronowych w języku C#. Omówiliśmy kluczowe komponenty biblioteki, takie jak warstwy sieci, funkcje aktywacji, funkcje straty, inicjalizatory wag, optymalizatory oraz mechanizmy dostarczania danych treningowych. Na zakończenie zastosowaliśmy bibliotekę do rozwiązania problemu przewidywania cen domów na podstawie danych ze zbioru Boston Housing, demonstrując jej praktyczne zastosowanie.

Created: 2025-12-03

Last modified: 2026-01-10

Title: **5. Biblioteka NeuralNetworks**

Tags: [C#] [Sieci neuronowe] [Biblioteka] [NeuralNetworks]

6. Dane MNIST

W poprzednim rozdziale przedstawiona została biblioteka NeuralNetworks - jej struktura, kluczowe komponenty oraz sposób definiowania, trenowania i wykorzystywania modeli sieci neuronowych. Omówione zostały elementy niskopoziomowe, takie jak operacje macierzowe i funkcje straty, oraz wyższego poziomu abstrakcje obejmujące modele, warstwy, operacje, optymalizatory oraz proces trenowania. Celem tamtego rozdziału było przedstawienie względnie elastycznego, ogólnego narzędzia, które ułatwi dalszą, praktyczną pracę z sieciami neuronowymi bez konieczności każdorazowego implementowania ich od podstaw.

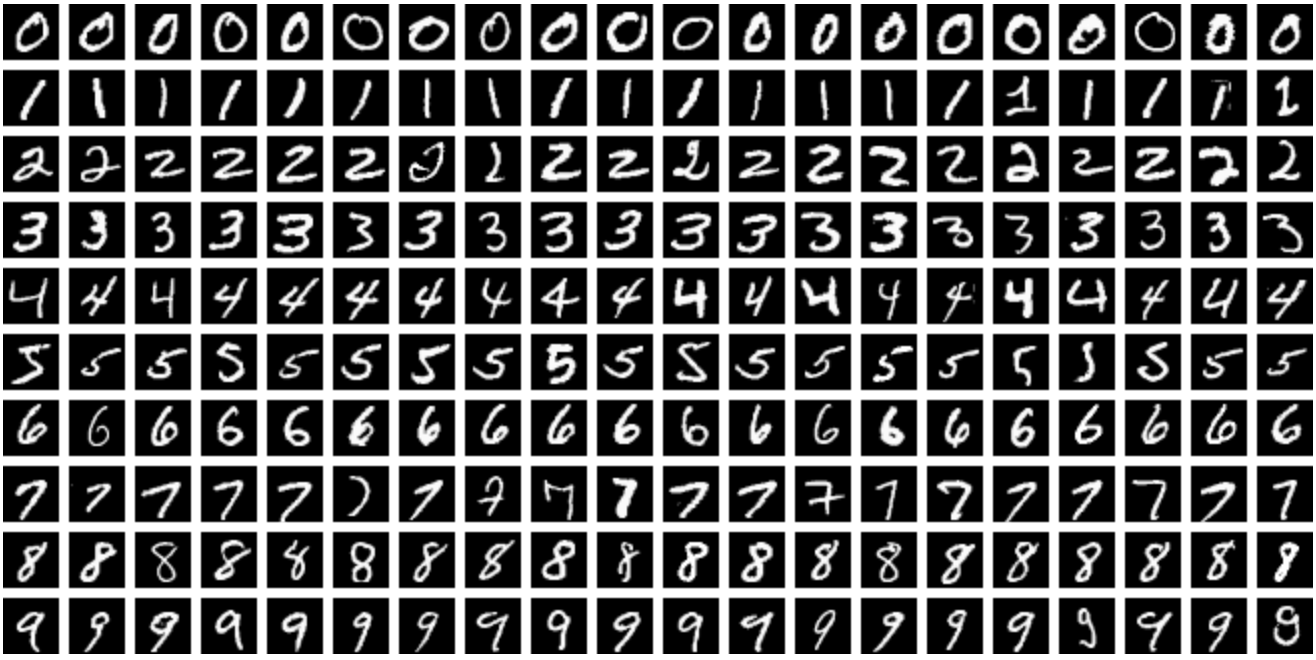
W niniejszym rozdziale przejdziemy od opisu biblioteki do jej praktycznego zastosowania. Jako przykład wykorzystamy klasyczny zbiór danych MNIST, zawierający obrazy odręcznie pisanych cyfr, który tradycyjnie stanowi punkt odniesienia w zadaniach klasyfikacji obrazów.

Analiza danych MNIST zostanie przeprowadzona z wykorzystaniem sieci opartych o warstwy gęste (ang. *dense layers*, *fully connected layers*) oraz z wykorzystaniem sieci konwolucyjnych.

6.1. Zbiór danych MNIST

Podobnie jak zbiór Boston Housing, dane MNIST są powszechnie dostępne i stosowane w literaturze. Zbiór ten zawiera 70 000 obrazów odręcznie pisanych cyfr (0-9), podzielonych na 60 000 obrazów treningowych i 10 000 obrazów testowych. Każdy obraz ma rozmiar 28x28 pikseli i jest reprezentowany jako macierz wartości szarości (od 0 do 255). Celem zadania jest sklasyfikowanie każdego obrazu do jednej z dziesięciu klas odpowiadających cyfrom od 0 do 9 (problem klasyfikacji wieloklasowej z pojedynczą etykietą, ang. *multi-class, single-label classification problem*).

W naszych eksperymentach będziemy pracować na znacznie mniejszym zbiorze uczącym, zawierającym jedynie 20 000 obrazów (plus 10 000 obrazów testowych). Zbiór ten znajduje się na GitHub.



Rysunek 6.1. Przykładowe obrazy z zestawu danych MNIST (źródło: Wikipedia)

6.2. Warstwy gęste

6.2.1. Architektura modelu

W naszych eksperymentach wykorzystamy prostą sieć neuronową zbudowaną z warstw gęstych. Architektura modelu będzie następująca:

```
class MnistModel(SeededRandom? random)
    : BaseModel<float[,], float[,]>(new SoftmaxCrossEntropyLoss(), random)
{
    private const float Dropout1KeepProb = 0.85f;
    private const float Dropout2KeepProb = 0.85f;

    private readonly Operation2D activationFunction1 = new ReLU();
    private readonly Operation2D activationFunction2 = new LeakyReLU();

    protected override LayerListBuilder<float[,], float[,]> CreateLayerListBuilder()
    {
        GlorotInitializer initializer = new(Random);
        Dropout2D? dropout1 = new(Dropout1KeepProb, Random);
        Dropout2D? dropout2 = new(Dropout2KeepProb, Random);

        return AddLayer(new DenseLayer(178, activationFunction1, initializer, dropout1))
            .AddLayer(new DenseLayer(46, activationFunction2, initializer, dropout2))
            .AddLayer(new DenseLayer(10, new Linear(), initializer));
    }
}
```

Listing 6.1. Definicja modelu MNIST z warstwami gęstymi

Model składa się z trzech warstw gęstych. Pierwsza warstwa zawiera 178 neuronów z funkcją aktywacji ReLU, druga warstwa ma 46 neuronów z funkcją aktywacji Leaky ReLU, a trzecia warstwa wyjściowa składa się z 10 neuronów (po jednym na każdą klasę cyfr) z liniową funkcją aktywacji. Dla pierwszych dwóch warstw zastosowano mechanizm dropout z prawdopodobieństwem zachowania neuronu równym 0.85, co pomaga w redukcji przeuczenia modelu. Funkcja straty użyta w modelu to Softmax Cross-Entropy, odpowiednia dla zadań klasyfikacji wieloklasowej.

W modelu zastosowano dwie różne funkcje aktywacji: ReLU (Rectified Linear Unit) oraz Leaky ReLU, jednak można eksperymentować z innymi funkcjami, takimi jak Sigmoid czy Tanh, aby ocenić ich wpływ na wyniki modelu.

Inicjalizacja wag

Dropout

Funkcje straty

Użytą w naszym modelu funkcją straty jest Softmax Cross-Entropy, która jest powszechnie stosowana w zadaniach klasyfikacji wieloklasowej.

6.2.2. Proces trenowania

6.2.2.1. Przygotowanie danych

6.2.2.2. Wyniki trenowania

6.3. Sieć konwolucyjna

Sieć konwolucyjna jest specjalnym rodzajem sieci neuronowej, która jest szczególnie skuteczna w analizie danych o strukturze przestrzennej, takich jak obrazy. W przeciwieństwie do warstw gęstych, które łączą każdy neuron z każdym innym neuronem w poprzedniej warstwie, warstwy konwolucyjne wykorzystują operacje konwolucji, które pozwalają na wykrywanie lokalnych wzorców w danych wejściowych.

Operacja konwolucji przebiega następująco:

1. Na wejściu mamy obraz reprezentowany jako macierz pikseli (np. 28x28 dla obrazów MNIST).
2. Nakładamy na obraz mały filtr (jądro konwolucyjne), który przesuwamy po obrazie, np. piksel po pikselu, wykonując operację iloczynu skalarnego między wartościami filtra a odpowiadającymi im wartościami pikseli w obrazie.
3. Wynikiem tej operacji jest nowa macierz (mapa cech), która reprezentuje wykryte wzorce w obrazie. Pojedynczą mapę cech nazywamy *kanałem*. W praktyce stosuje się wiele filtrów, co prowadzi do powstania wielu kanałów.

6.3.1. Architektura modelu

6.3.2. Proces i wyniki trenowania

Created: 2025-12-19

Last modified: 2026-01-10

Title: **6. Dane MNIST i warstwy gęste**

Tags: [C#] [Sieci neuronowe] [Biblioteka] [NeuralNetworks] [MNIST] [Warstwy gęste] [Dense Layers] [Fully Connected Layers]