

**Politechnika Wrocławskas
Wydział Elektroniki, Fotoniki i Mikrosystemów**

KIERUNEK: Elektronika i Telekomunikacja

**PRACA DYPLOMOWA
MAGISTERSKA**

Tytuł pracy: Zastosowanie narzędzi AI do
analizy danych elipsometrycznych

PROMOTOR:
dr hab. inż. Sergiusz Patela

[Ta strona została celowo pozostawiona pusta]

Spis treści

Spis treści	3
Streszczenie	4
Abstract	4
Wstęp.....	5
1. Wprowadzenie do pomiarów elipsometrycznych	6
1.1 Podstawy elipsometrii: Zasada działania i budowa układu pomiarowego	6
1.2 Definicja wielkości elipsometrycznych	8
1.3 Modelowanie danych elipsometrycznych	10
1.4 Ograniczenia i wyzwania elipsometrii	12
2. Wprowadzenie do uczenia maszynowego.....	13
2.1 Historia uczenia maszynowego	13
2.2 Regresja liniowa – zasada działania.....	14
2.3 Sieci neuronowe – zasada działania	17
3. Dane elipsometryczne – analiza	22
Podsumowanie	25
Bibliografia.....	26
Spis równań	27
Spis rysunków	27

Streszczenie

Celem niniejszej pracy było zbadanie możliwości zastosowania narzędzi sztucznej inteligencji oraz uczenia maszynowego do analizy danych elipsometrycznych. Celem było zbadanie możliwości zastosowania sieci neuronowych oraz regresji liniowej do wyznaczenia grubości badanej próbki – T – oraz współczynników Cauchy'ego, opisujących zmianę współczynnika załamania światła warstwy w funkcji długości fali elektromagnetycznej padającej na próbkę. W celu realizacji założeń pracy zdecydowano się na zastosowanie języka programowania Python oraz środowiska programistycznego Jupyter Notebook. W celu realizacji części pracy obejmującej zagadnienia związane z regresją zdecydowano się na zastosowanie biblioteki scikit-learn, a do części związanej z sieciami neuronowymi użyto biblioteki PyTorch. Przeprowadzane prace przyniosły obiecujące wyniki w przypadku prób zastosowania regresji liniowej w celu wyznaczania grubości próbki oraz w przypadku prób wyznaczania wartości grubości oraz parametrów A oraz B za pomocą sieci neuronowych.

Abstract

The aim of this thesis was to explore the potential application of artificial intelligence and machine learning tools for the analysis of ellipsometric data. The objective was to investigate the feasibility of using neural networks and linear regression to determine the thickness of the examined sample – T – as well as the Cauchy coefficients, which describe the variation of the refractive index of the layer as a function of the wavelength of the electromagnetic radiation incident on the sample. To achieve the goals of the thesis, the Python programming language and the Jupyter Notebook development environment were chosen. For the part of the work related to regression, the scikit-learn library was used, while for the neural network section, the PyTorch library was employed. The conducted studies yielded promising results, particularly in the attempts to use linear regression for determining the sample thickness and in the efforts to estimate the thickness and the A and B parameters using neural networks.

Wstęp

Elipsometria to precyzyjna, nieniszcząca technika pomiarowa, umożliwiająca analizę właściwości cienkowarstwowych struktur, takich jak grubość warstwy czy współczynnik załamania światła [1]. Jej rozwój był podkutowany ograniczeniami tradycyjnych metod pomiarowych, które zawodzą przy charakterystyce warstw o grubości rzędu nanometrów [2]. Wraz z postępem technologii i miniaturyzacją struktur – szczególnie w elektronice i fotonice – znaczenie elipsometrii stale rośnie. Pomimo jej ogromnego potencjału, analiza danych elipsometrycznych pozostaje procesem złożonym i wrażliwym na przyjęte modele teoretyczne. Kluczowe parametry próbki, takie jak grubość czy współczynniki modelu Cauchy'ego, nie są bezpośrednio mierzone, lecz muszą być wyznaczane na podstawie pośrednich parametrów Ψ i Δ . Proces ten wymaga doboru odpowiedniego modelu optycznego oraz ręcznej kalibracji, co bywa czasochłonne i podatne na błędy interpretacyjne – zwłaszcza w przypadku struktur wielowarstwowych lub nieregularnych.

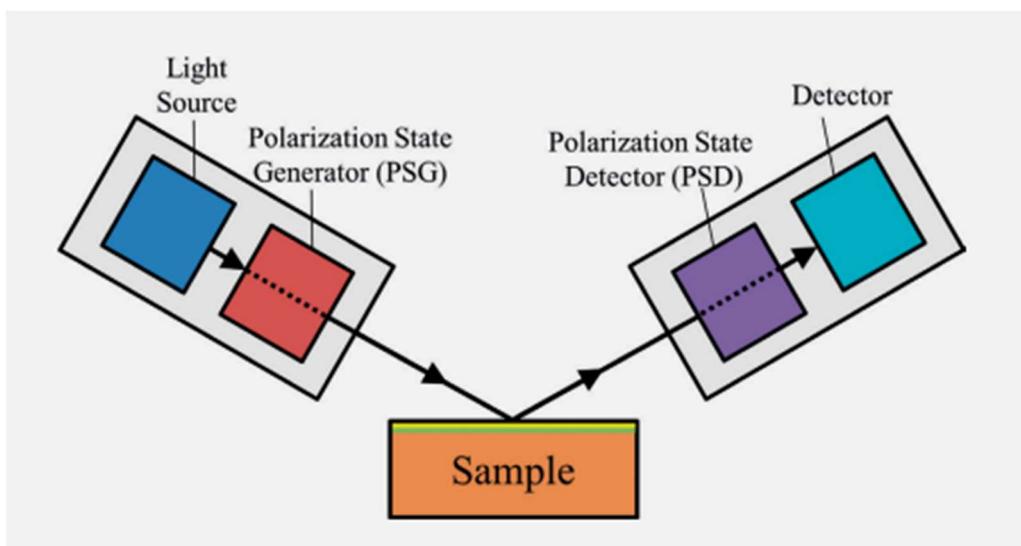
Współczesne osiągnięcia w dziedzinie sztucznej inteligencji, a zwłaszcza uczenia maszynowego, otwierają nowe możliwości w kontekście automatyzacji i usprawnienia interpretacji danych elipsometrycznych. Modele regresji, sieci neuronowe oraz inne algorytmy predykcyjne pozwalają wytrenować system, który potrafi estymować parametry fizyczne badanej próbki bez konieczności każdorazowego doboru modelu przez eksperta. Zastosowanie narzędzi AI może przyczynić się do znacznego skrócenia czasu analizy, zwiększenia jej dokładności oraz eliminacji subiektywnych błędów interpretacyjnych. Szczególnie obiecujące są tu sieci neuronowe, które dzięki swojej zdolności do modelowania nieliniowych zależności i pracy z dużymi zbiorami danych, mogą uczyć się bezpośrednio odwrotnej transformacji pomiędzy przestrzenią parametrów elipsometrycznych a fizycznymi właściwościami warstw.

Celem niniejszej pracy jest zbadanie możliwości zastosowania wybranych algorytmów AI – takich jak regresja liniowa i sieci neuronowe – do przewidywania grubości warstw oraz współczynników Cauchy'ego na podstawie danych elipsometrycznych. Rozważania te wpisują się w szerszy trend integracji technik sztucznej inteligencji z zaawansowaną analizą danych pomiarowych, stanowiąc krok w stronę bardziej autonomicznych i intelligentnych systemów badawczych.

1. Wprowadzenie do pomiarów elipsometrycznych

1.1 Podstawy elipsometrii: Zasada działania i budowa układu pomiarowego

Elipsometria to nieniszcząca technika pozwalająca na badanie właściwości warstw cienkich takich jak grubość bądź współczynnik załamania światła. Technika ta została zapoczątkowana na początku lat 50 XX wieku. Motywacją stojącą za jej opracowaniem był fakt, że konwencjonalne metody pomiarów grubości – tj. mikrometry bądź suwmiarki – są nieefektywne przy dokonywaniu pomiarów warstwach cienkich, których wymiary mogą zawierać się w zakresie od dziesiątych części mikrometra do nawet pojedynczych nanometrów. Metodom tym brak odpowiedniej precyzji, która zapewniłaby badaczowi pożądane informacje z adekwatną dokładnością. Dane na temat próbki uzyskuje się podczas pomiaru elipsometrycznego poprzez obserwację zmiany polaryzacji wiązki światła o znanej polaryzacji po odbiciu od badanej warstwy. Schemat elipsometrycznego układu pomiarowego przedstawiono na rysunku 1.



Rysunek 1 – Schemat elipsometrycznego układu pomiarowego [3]

Pierwszym z krytycznych elementów układu jest część odpowiedzialna za generację wiązki promieniowania o znanej polaryzacji. Polaryzacja ta nie musi być stała – może fluktuować i zmieniać się w czasie. Wiązka światła jest emitowana za pomocą źródła zdolnego do wytwarzania promieniowania o szerokim zakresie długości fali – z tego powodu nie używa

się zazwyczaj diod bądź laserów. Generowana wiązka nie musi mieć stałej intensywności – ważne jest jednak by intensywność wiązki była stała w okresie, podczas którego dokonywany jest pomiar dla danej długości fali.

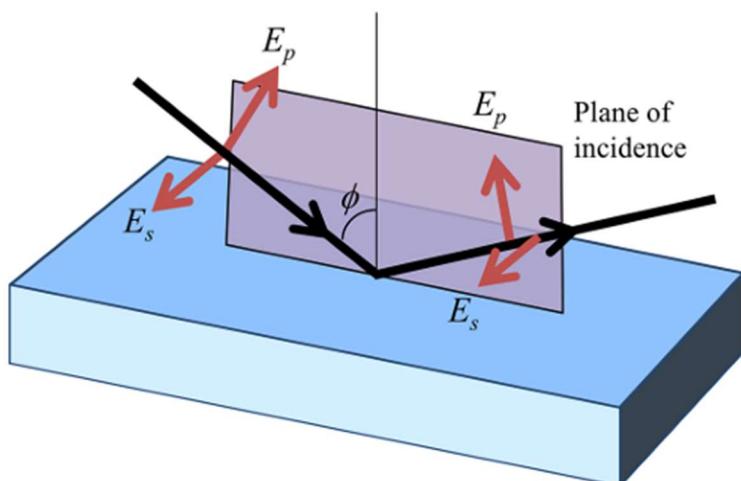
Drugim elementem, który należy wymienić odpowiada za nadanie generowanej wiązce światła konkretnej, znanej polaryzacji. W roli tej używa się:

- polaryzatorów – ich zadaniem jest zmiana polaryzacji padającej na nie wiązki na polaryzację liniową. Najczęstszym mechanizmem ich działania jest podwójne załamanie optyczne występujące w pryzmatach takich jak pryzmat Wollastona. Jakość pryzmatu jest oceniana poprzez porównanie natężenia wiązki światła opuszczającej pryzmat do natężenia wiązki światła padającej w kierunku prostopadłym do pożądanej osi [3].
- kompensatorów – działają one poprzez zmianę przesunięcia fazowego pomiędzy prostopadłymi do siebie orientacjami pola. W przypadkach, gdy kompensator zmienia fazę o kąt 90° , może on zmienić wiązkę o polaryzacji kołowej w wiązkę o polaryzacji liniowej itd. Zmiany przesunięcia fazowego dokonuje się poprzez transmisję wiązki przez materiał optycznie anizotropowy. Powoduje to, że fale o polaryzacji zbliżonej do tzw. osi spowolnionej rozchodzą się w ośrodku wolniej niż te które znajdują się bliżej osi szybkiej, co powoduje przesunięcie się grzbietów fal względem siebie [3].
- modulatorów fazy – operują na zasadzie zbliżonej do zasady działania kompensatora. Istotną różnicą jest fakt, że opóźnienie fazy przez modulator fazy jest zmienne. Zmiany tej dokonuje się np. poprzez poddanie materiału, z którego zbudowany jest modulator odpowiednim naprężeniom które wpłyną na jego właściwości optyczne, bądź poprzez zmianę położenia modulatora względem padającej na niego wiązki światła. W roli modulatorów fazy używa się także ciekłych kryształów, które posiadają zdolność do zmiany swoich osi optycznych pod wpływem przyłożonego do nich pola elektrycznego [3].

Rolę detektora pełni zazwyczaj fotodioda bądź matryca fotodiodowa. Zakres długości fali stosowanych podczas pomiaru jest ograniczony przez szerokość przerwy energetycznej półprzewodnika użytego jako detektor. Najpopularniejszym materiałem jest krzem, ale stosuje się także materiały takie jak InGaAs, MCT, DTGS. W elipsometrach w roli detektorów stosuje się także fotopowielacze – detektory optyczne charakteryzujące się niezwykle wysokim wzmacnieniem sygnału optycznego. Pomiar zależności natężenia promieniowania padającego na detektor dokonuje się w sposób analogiczny do sposobu tworzenia wiązki o odpowiedniej polaryzacji [4].

1.2 Definicja wielkości elipsometrycznych

Drgania pola elektrycznego stanowiące wiązkę padającą na badany materiał możemy opisać za pomocą dwóch wektorów oznaczanych zazwyczaj literami p i s . Litery te pochodzą od niemieckich słów *parallel* i *senkrecht* które oznaczają odpowiednio „prostopadły” i „równoległy”. Schemat przedstawiający wiązkę padającą na próbce przedstawiony jest na rysunku 3. Energia fali równoległej jest oznaczona na nim poprzez E_p a energia fali prostopadłej poprzez E_s .



Rysunek 2 – Zachowanie wiązki promieniowania przy interakcji z powierzchnią próbki [3]

Zarówno wektory opisujące fale padające na próbkę jak i wektory opisujące fale od niej odbite, mogą być przesunięte w fazie względem siebie. Przesunięcie w fazie między wektorami opisującymi wiązkę padającą oznaczone będzie przez δ_1 a przesunięcie w fazie pomiędzy wektorami opisującymi wiązkę odbitą oznaczone będzie przez δ_2 . Jak wynika z równań Fresnela, wraz ze zmianą kąta padania wiązki, zmianie ulegają także amplitudy energii pola elektrycznego drgającego w płaszczyznach poprzecznej i prostopadłej do płaszczyzny padania. Stosunek amplitud składowych wiązki padającej do amplitud składowych wiązki odbitej przedstawiony jest za pomocą całkowitych współczynników odbicia – R_S i R_P .

Bezpośrednimi wynikami pomiarów elipsometrycznych są Ψ – „PSI” – oraz Δ – „DELTA”. Parametr DELTA definiuje się jako różnicę pomiędzy przesunięciem fazowym pomiędzy falą równoległą – p – do powierzchni próbki a falą prostopadłą – s – do powierzchni próbki w wiązce padającej na próbkę a przesunięciem fazowym pomiędzy falami p i s w wiązce odbitej od próbki. Możemy go opisać wzorem:

$$\Delta = \delta_1 - \delta_2 \quad \text{Równanie 1}$$

Δ – parametr delta

δ_1 – przesunięcie fazowe pomiędzy wektorami pola p i s w wiązce padającej

δ_2 – przesunięcie fazowe pomiędzy wektorami pola p i s w wiązce odbitej

Parametr PSI zaś definiuje się jako stosunek amplitud fali p oraz fali s w wiązce odbitej i może zostać przedstawiony jako stosunek całkowitych współczynników odbicia R_P i R_S jak na wzorze poniżej:

$$\tan(\Psi) = \left| \frac{R_p}{R_s} \right| \quad \text{Równanie 2}$$

Fundamentalne równanie elipsometrii można przedstawić jako:

$$\rho = \tan(\Psi)e^{i\Delta} = \frac{R_p}{R_s} \quad \text{Równanie 3}$$

ρ – liczba zespolona równa stosunkowi całkowitych współczynników odbicia $\frac{R_p}{R_s}$

Zakładając, że urządzenie pomiarowe funkcjonuje poprawnie, wartości DELTA i PSI powinny zawsze być poprawne. Na ich podstawie jesteśmy w stanie wyznaczyć właściwości takie jak grubość czy chropowatość. Aby tego dokonać, konieczne jest poczynienie założeń na temat próbki – takich jak np. ilość warstw w próbce.

1.3 Modelowanie danych elipsometrycznych

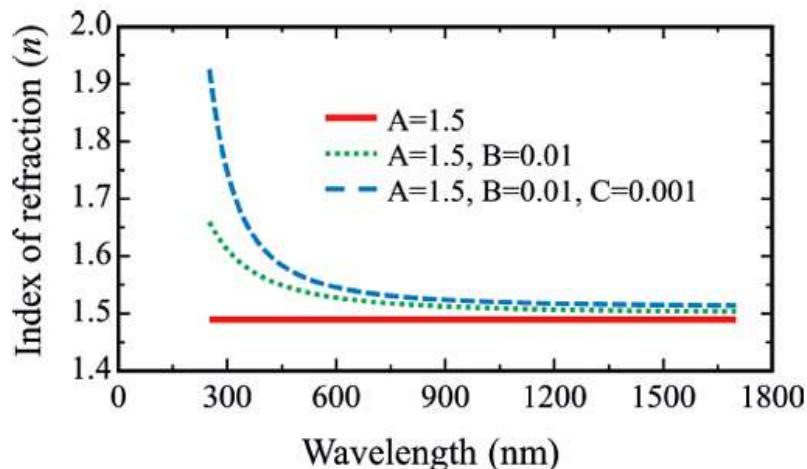
Parametry PSI i DELTA pozwalają na przedstawienie sposobu w jaki zmieniła się polaryzacja wiązki odbitej względem wiązki padającej, nie pozwalają jednak na bezpośrednie odczytanie parametrów próbki. Aby uzyskać interesujące dla użytkownika informacje konieczne jest zastosowanie odpowiedniego modelu. To, jaki model będzie najbardziej odpowiedni zależy od tego, jakie informacje posiadamy na temat badanej próbki.

Dla warstw przeźroczystych, współczynnik załamania maleje wraz ze wzrostem długości fali padającej na warstwie. Popularnym dla warstw przeźroczystych modelem przedstawiającym tą zależność jest empiryczny model Cauchy'ego [5]. Równanie modelu Cauchy'ego przedstawiono poniżej:

$$n(\lambda) = A + \frac{B}{\lambda^2} + \frac{C}{\lambda^4} \quad \text{Równanie 4}$$

Opisuje ono zależność współczynnika załamania próbki od długości fali, za pomocą parametrów zwanych współczynnikami Cauchy'ego.

Parametr A jest wyrazem wolnym i wpływa na przesunięcie zależności względem 0. Parametry B i C wpływają zaś na krzywiznę zależności. Wpływ poszczególnych współczynników na wygląd zależności współczynnika załamania n od długości fali przedstawiono na rysunku 4:



Rysunek 3 – Zależność współczynnika załamania światła od długości fali dla różnych wartości parametrów Cauchy'ego [3]

Model Cauchy'ego w swojej podstawowej formie zakłada, że współczynnik ekstynkcji próbki jest równy零. Dla próbek posiadających niewielką absorpcyjność w zakresie UV istnieje możliwość zmodyfikowania równania Cauchy'ego poprzez dodanie do niego równania Urbacha, które opisuje współczynnik ekstynkcji w funkcji długości fali. Równanie Urbacha przedstawiono poniżej [5]:

$$k(\lambda) = A_k e^{B_k(E - E_b)} \quad \text{Równanie 5}$$

E – energia fotonu [eV]

$$E_b = \frac{1240}{\lambda_b} [\text{eV}]$$

Zazwyczaj wartość λ_b określana jest przez badacza, a wartości A_k i B_k wyznaczane są numerycznie – na przykład poprzez regresję.

1.4 Ograniczenia i wyzwania elipsometrii

Pomimo swoich zalet – takich jak bycie techniką nieniszczącą oraz możliwa do uzyskania wysoka precyzaja – elipsometria nie pozostaje bez wad. Jedną z nich jest wysoka wrażliwość wyników pomiaru na niejednorodności i chropowatość powierzchni próbki. Wszelkie defekty obecne w próbce mogą wpływać znacząco na właściwości fali rejestrowanej przez elipsometr, co z kolei może przełożyć się na błędą interpretację pomiaru [6].

Pomiary elipsometryczne mają także ograniczone zastosowanie podczas badania próbek o niezwykle niewielkiej bądź niezwykle dużej grubości. Warstwy o niewielkich grubościach mogą powodować małe, trudne do zauważenia zmiany w polaryzacji, co może powodować trudności przy rozróżnianiu pomiędzy właściwościami podłoża i próbki [7].

Powierzchnia badana za pomocą elipsometru podczas jednej serii pomiarowej jest zazwyczaj niewielka – ma rozmiar kilku mm^2 . Sprawia to, że w przypadku występowania różnic we właściwościach materiałowych w obrębie jednego materiału, wnioski wyciągnięte na podstawie wyników otrzymanych z jednej tylko serii pomiarowej mogą być błędne [1].

Ograniczeniem, na którym skupiać się będzie ta praca jest jednak to, że elipsometria polega na pomiarach pośrednich. Zmiany polaryzacji wiązki światła odbitego od powierzchni próbki nie niosą informacji, które byłyby przydatne dla użytkownika same w sobie. Wymagają one odpowiedniej interpretacji za pomocą odpowiedniego modelu. Błędny wybór modelu będzie miał znaczący wpływ na ostateczne otrzymane przez użytkownika informacje na temat badanej próbki. Dobór odpowiedniego modelu może być szczególnie trudny w przypadku badania skomplikowanych, wielowarstwowych struktur i materiałów [4] [8].

2. Wprowadzenie do uczenia maszynowego

2.1 Historia uczenia maszynowego

Historia uczenia maszynowego sięga połowy XX wieku. W latach pięćdziesiątych psycholog Frank Rosenblatt stworzył grupę naukową, która zbudowała maszynę zdolną do rozpoznawania liter alfabetu. Podczas projektowania inspirował się budową ludzkiego układu nerwowego. Nazwał swoje dzieło „Perceptron”. Innowacją w perceptronie był sposób programowania, wzorowany na procesie nauki. W latach 60. i 70. rozwijały się modele deterministyczne i stochastyczne uczenia. Modele stochastyczne opierały się na losowości i metodach probabilistycznych, podczas gdy metody deterministyczne korzystały z matematycznych funkcji, a wyniki były ściśle określone [9] [10].

W latach 70. ukazała się książka autorstwa Marvin'a Minsky'ego i Seymour'a Papert'a pt. „*Perceptrons: An Introduction to Computational Geometry*”. Dyskutowano w niej ograniczenia Perceptronu, takie jak brak możliwości reprezentacji niektórych funkcji logicznych, np. XOR i NXOR. Odkrycie to spowodowało wstrzymanie badań nad Perceptronem aż do lat 80 [9] [10].

W roku 1980 Kunihiko Fukushima przedstawił model wielowarstwowej sieci konwolucyjnej, którą nazwał „Neocognition”. W połowie lat 80. odkryto mechanizm propagacji wstecznej, co doprowadziło do nowej fali innowacji w sieciach neuronowych. Postępy w uczeniu maszynowym nie spełniły jednak oczekiwania, co spowodowało kolejny przestój w badaniach [11].

Następny przełom nastąpił w roku 1995, kiedy opublikowano artykuł autorstwa Corinny Cortes i Vladimira Vapnika. Opisywał on nową wersję algorytmu SVM – *support vector machine* – znaną jako *support-vector networks*. Artykuł ten zrewolucjonizował spojrzenie na uczenie maszynowe, osiągając 22000 cytowań w 2019 roku [9] [10].

Rozwój uczenia maszynowego wspierały trendy, które zyskały na znaczeniu na początku XXI wieku. Pierwszym z nich było Big Data – gromadzenie dużych ilości danych, które wymagały algorytmów z dziedziny uczenia maszynowego. Drugim trendem był spadek cen mocy obliczeniowej oraz pamięci, a także usprawnienie obliczeń równoległych. Trzecim

trendem był rozwój nowych algorytmów związanych z uczeniem maszynowym, w tym algorytmy głębokiego uczenia oraz koncepcja głębokiej sieci neuronowej [9] [10].

Wszystkie te wydarzenia i odkrycia stanowiły fundament dla dynamicznego rozwoju uczenia maszynowego w XXI wieku. Dzięki rosnącej dostępności danych, postępu technologicznym oraz innowacjom algorytmicznym, uczenie maszynowe stało się kluczową dziedziną informatyki, znajdującą zastosowanie w niemal każdej gałęzi przemysłu i nauki [9] [10].

2.2 Regresja liniowa – zasada działania

Jednym z najstarszych i najprostszych modeli uczenia maszynowego jest model regresji liniowej [10] [11]. Jest to model uczenia nadzorowanego – tj. model, którego uczenia dokonuje się posiadając nie tylko przykładowy zbiór cech, na podstawie których dokonywane będą poszczególne przewidywania, ale także zbiór przykładowych etykiet będących wynikami tych przewidywań. Przykładem mógłby być zbiór danych zawierający wzrost i rozmiar buta grupy osób, gdzie cechą, na podstawie której dokonywano by przewidywania byłby wzrost a przewidywaną etykietą byłby rozmiar buta danej osoby. Zależność rozmiaru buta – etykiety – od wysokości można przedstawić za pomocą równania:

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n \quad \text{Równanie 6}$$

\hat{y} – przewidywana etykieta

n – liczba cech

x_n – wartości cech

θ_n – wagи poszczególnych cech

θ_0 – wyraz wolny

Celem szkolenia modelu jest zminimalizowanie wybranej przez użytkownika funkcji kosztu. Jednymi z bardziej popularnych funkcji kosztu są błąd średniokwadratowy – MSE – oraz

pierwiastek błędu średniokwadratowego – RMSE – które możemy wyznaczyć za pomocą wzorów przedstawionych poniżej:

$$MSE = \frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2 \quad \text{Równanie 7}$$

$$RMSE = \sqrt{MSE} \quad \text{Równanie 8}$$

Główymi metodami przeprowadzania regresji liniowej są metoda analityczna oraz metody prostego. Metoda analityczna pozwala uzyskanie rozwiązania w sposób bezpośredni. Dokonuje się tego poprzez rozwiązywanie tzw. równania normalnego:

$$\hat{\theta} = (X^T X)^{-1} X^T y \quad \text{Równanie 9}$$

$\hat{\theta}$ – wektor wag dla których funkcja kosztu jest najmniejsza

y – wektor poszukiwanych etykiet

X – wektor cech

X^T – transponowany wektor cech

Wynikiem rozwiązywania tego równania jest macierz zawierająca w sobie poszukiwane wagi oraz wyraz wolny.

Minusem tego rozwiązywania jest fakt, że złożoność obliczeniowa algorytmów służących do analitycznego rozwiązywania przedstawiona za pomocą notacji dużego O znajduje się zazwyczaj w zakresie od $O(2,4)$ do $O(3)$. Notacja dużego O jest używana do określenia jak zmieni się liczba operacji wykonanych przez dany algorytm w zależności od wielkości zbioru danych, na którym algorytm przeprowadzać będzie operację. $O(0)$ oznacza, że liczba operacji jest stała w zależności od rozmiaru zbioru, $O(1)$ oznacza, że rośnie ona liniowo wraz z rozmiarem zbioru a $O(3)$ wskazuje na wzrost sześcienny. Z racji tego, każde podwojenie liczby

cech używanych podczas szkolenia modelu spowoduje wzrost cykli procesora potrzebnych do wykonania obliczeń o od 2^{24} do 2^3 razy. Biorąc pod uwagę, że coraz częściej konieczne jest przeprowadzenia regresji na zbiorach posiadających nawet setki różnych cech, rozwiązywanie równania normalnego przestaje być najwydajniejszą metodą ekstrakcji istotnych danych.

Sposobem, który pozwala ograniczyć złożoność obliczeniową zastosowanie metod gradientowych z których prawdopodobnie najprostszą jest metoda gradientu prostego. Polega na krokowej aktualizacji wag (oraz wyrazu wolnego), polegającej na obliczeniu gradientu funkcji kosztu względem tych parametrów i przesunięciu ich w kierunku przeciwnym do gradientu. Ten kierunek wskazuje, gdzie funkcja kosztu zmniejsza się najszybciej. W każdej iteracji algorytm wykonuje aktualizację:

$$\theta_{n+1} = \theta_n - \alpha \nabla J(\theta_{n+1})$$

Równanie
10

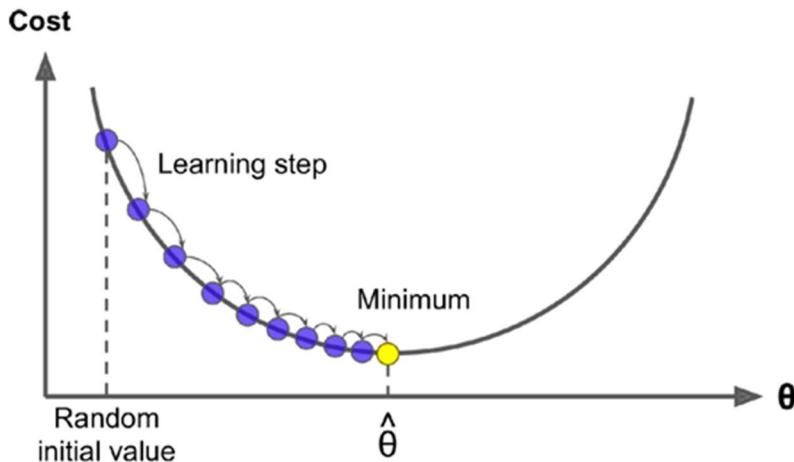
θ_{n+1} – wartości wag w następnym kroku

θ_n – wartości wag w obecnym kroku

$\nabla J(\theta_{n+1})$ – gradient funkcji kosztu względem wag

α – prędkość uczenia

Proces powtarza się, aż zmiany wartości funkcji kosztu będą dostatecznie małe. Intuicyjnie można to porównać do osoby poruszającej się w mgle po górach – badającej nachylenie terenu i schodzącej w kierunku największego spadku, aby odnaleźć dno doliny – minimum funkcji. Minimalizacja funkcji kosztu za pomocą algorytmu gradientu prostego została przedstawiona na rysunku poniżej:



Rysunek 4 – Minimalizacja funkcji kosztu za pomocą metody gradientu prostego [11]

2.3 Sieci neuronowe – zasada działania

Od lat wiele powstających technologii inspirowanych było przez obiekty i zjawiska wywodzące się z biologii. Działem nauki który zajmuje się emulowaniem naturalnych rozwiązań do ludzkich problemów jest mimetyka. Zawdzięczamy jej wynalazki takie jak:

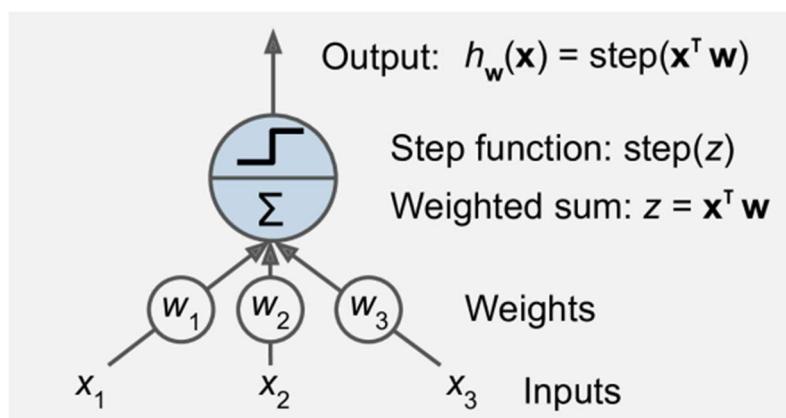
- Velcro (rzep) – rzep został zainspirowany owocami łopianu, które posiadają drobne haczyki umożliwiające przyczepianie się do sierści zwierząt lub ubrań ludzi, co ułatwia ich rozsiewanie. Wynalazca, George de Mestral, zauważył ten mechanizm podczas spaceru z psem i po latach badań stworzył funkcjonalny system zapinania i odpinania, który dziś jest powszechnie stosowany w odzieży, sprzęcie sportowym czy medycynie.
- Kształty samolotów – aerodynamika i kształt skrzydeł wielu samolotów były inspirowane obserwacjami lotu ptaków. Przykładowo, konstrukcja skrzydeł szybowców nawiązuje do rozpiętości i profilu skrzydeł ptaków drapieżnych, takich jak orły. Dzięki temu możliwe było zminimalizowanie oporu powietrza i zwiększenie wydajności lotu.
- Nici przednie – pajęczy jedwab, pomimo swojej lekkości, charakteryzuje się wyjątkową wytrzymałością – większą niż stal o tej samej masie. Wzbudziło to zainteresowanie

naukowców, którzy próbują stworzyć jego syntetyczną wersję. Potencjalne zastosowania obejmują nici chirurgiczne, kamizelki kuloodporne czy materiały kompozytowe w lotnictwie.

- Izolacja termiczna – niektóre nowoczesne materiały termoizolacyjne, np. aerogeły lub specjalne włókna, zostały opracowane na podstawie badań nad strukturą sierści niedźwiedzi polarnych czy futra pingwinów. Zwierzęta te żyją w ekstremalnych warunkach klimatycznych, a ich sierść lub pióra skutecznie zatrzymują ciepło.

Do kategorii tej należą także sieci neuronowe – koncept stojący w centrum tak zwanego uczenia głębokiego – dziedziny będącej podzbiorem uczenia maszynowego, wykorzystującej głębokie sieci neuronowe. Są one niezwykle potężnym narzędziem, używanym obecnie w wielu dziedzinach nauki tj. rozpoznawanie mowy bądź pisma, prognozowanie cen produktów bądź wartości akcji na giełdzie czy analiza wyników badań medycznych. Najprostszy element sieci neuronowej – neuron – wzorowany jest na swoim biologicznym odpowiedniku. Biologiczne neurony generują krótkie impulsy elektryczne w momentach które uzależnione są od sygnałów które dostarczane są do niego z zewnątrz przez neuroprzekaźniki. Pomimo tego, że zasada działania pojedynczego neurony wydaje się dość prosta, ich kolektywne działanie jest w stanie doprowadzić do występowania skomplikowanych, złożonych i różnorodnych efektów.

Jedną z najprostszych architektur sieci neuronowych jest Perceptron wynaleziony przez Rosenblatt'a. Jego wejścia i wyjście przyjmują postać liczb, a to w jaki sposób dane wejście wpłynie na wartość pojawiającą się na wyjściu determinowane jest przez przypisaną do niego wagę. Wartość wyjściowa jest ustalana poprzez aplikację do sumy ważonej wejść funkcji aktywacji. Schemat działania perceptronu przedstawiono na rysunku poniżej.



Rysunek 5 – Schemat działania perceptronu [11]

Pierwotnie stosowanymi funkcjami aktywacji były funkcje skokowe. Funkcja krokowa przyjmuje stałe wartości wyjściowe dla danych zakresów wartości wejściowych. Przykładami takich funkcji są np. funkcja signum bądź funkcja skokowa Heavside'a.

Obecnie najpopularniejszym i funkcjami aktywacji są:

- ReLu (rectified linear unit) – ciągła, nieróżniczkowalna w punkcie 0 funkcja określona wzorem:

$$ReLU(x) = \max(0, x) = \frac{x + |x|}{2} \quad \begin{matrix} Równanie \\ 11 \end{matrix}$$

Jej zaletą jest fakt, że nie posiada wartości maksymalnej. Jest w stanie przyjmować tylko nieujemne wartości.

- Sigmoidalna – ciągła, przyjmuje wartości z zakresu od 0 do 1. Posiada zdefiniowaną pochodną w całym swoim zakresie, co jest przydatne w przypadku zastosowania gradientu prostego. Jej wadą jest fakt, że „nasycza” się dla dużych wartości – powyżej pewnej wartości na wejściu nawet duże zmiany w tej wartości nie doprowadzą do znaczącej zmiany na wyjściu. Problem ten nazywa się problemem „zanikającego gradientu”. Określona jest wzorem:

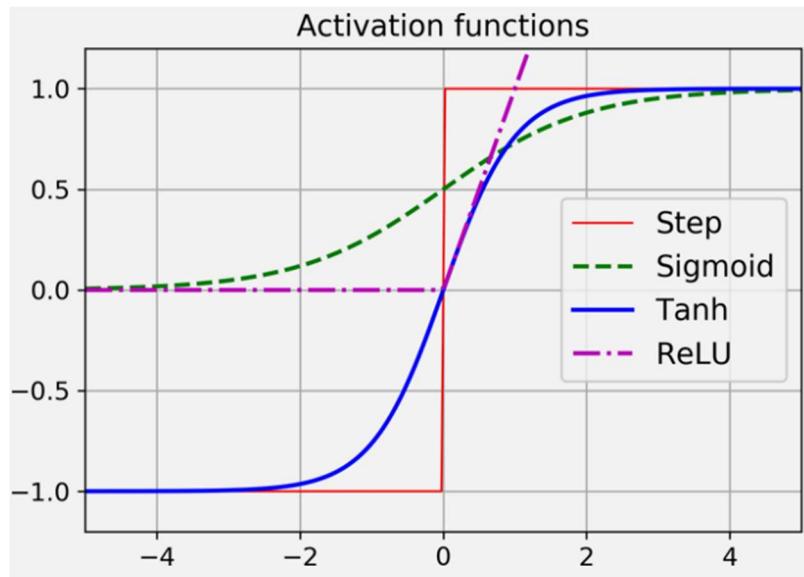
$$\sigma(x) = \frac{e^x}{1 + e^x} \quad \begin{matrix} Równanie \\ 12 \end{matrix}$$

- Tanh (tangens hiperboliczny) – funkcja ciągła przyjmująca wartości z zakresu od -1 do 1. Jest ciągła i różniczkowalna w całym zakresie. Podobnie jak funkcja Sigmoidalna

cierpi ona na problem zanikającego gradientu. Jej plusem jest fakt, że może przyjmować ujemne wartości. Określona jest wzorem:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad \begin{matrix} Równanie \\ 13 \end{matrix}$$

Wygląd wyżej opisanych funkcji przedstawiono na rysunku:



Rysunek 6 – Przebiegi najpopularniejszych funkcji aktywacji [11]

Inspiracją dla sposobu szkolenia perceptronu była tzw. zasada Hebbia, która została opisana przez psychologa Donalda Hebbia w książce „The Organization of Behavior” wydanej w roku 1949. Jest to reguła kciuka stanowiąca, że neurony które są aktywowane jednocześnie tworzą ze sobą połączenia synaptyczne – „cells that fire together, wire together”.

Zainspirowany został nią wzór, który wiąże zmianę wagi przy danym jego wejściu dla kroku $n+1$ z wartością błędu w kroku n . Wzór ten przedstawiono poniżej:

$$w_{i,j}^{(n+1)} = w_{i,j}^{(n)} + \eta(y_j - \hat{y}_j)x_i \quad \begin{matrix} Równanie \\ 14 \end{matrix}$$

$w_{i,j}^{(n+1)}$ – wartość wagi w kroku $n + 1$

$w_{i,j}^{(n)}$ – wartość wagi w kroku n

$(y_j - \hat{y}_j)$ – różnica pomiędzy wartością na wyjściu perceptronu a etykietą

x_i – wartość na wejściu perceptronu

η – prędkość uczenia

Znaczącą wadą pojedynczego perceptronu jest fakt, że nie jest on w stanie uczyć się wzorów bardziej skomplikowanych niż pojedyncza linia. Sprawia to, że nie są one w stanie rozwiązywać takich logicznych funkcji jak OR, XOR bądź NXOR. Okazuje się jednak, że niektóre z ograniczeń perceptronu da się ominąć poprzez połączenie ze sobą kilku perceptronów tworząc tym samym MLP – *Multilayer Perceptron*. Układ taki złożony jest z warstwy wejściowej, warstwy wejściowej oraz jednej lub więcej warstw ukrytych. Jeśli układ posiada odpowiednio wiele warstw ukrytych to nazywamy go układem głębokim. Pierwotnie MLP nie posiadały także mechanizmu propagacji wstecznej co sprawiało, że modyfikacji wag trzeba było dokonywać manualnie. Omówiony w roku 1986 algorytm propagacji wstecznej rozwiązał ten problem, pozwalając na aktualizację wag poszczególnych połączeń w zaledwie dwóch przejściach danych – od warstwy wejściowej do wyjściowej i z powrotem.

Algorytm ten:

- Analizuje jedną próbke bądź jedną partię próbek na raz.
- Po przejściu przez warstwę wejściową dane trafiają do warstw ukrytych. Po przejściu przez każdą warstwę ukrytą algorytm oblicza wartości na wyjściach wszystkich neuronów należących do warstwy. Wartości te są przekazywane do neuronów w następnej warstwie ukrytej (jeśli takowa istnieje) i proces jest powtarzany. Postępowanie to jest kontynuowane aż do momentu aż dane trafią do warstwy wyjściowej. Wartości wyjściowe każdej z warstw są zapamiętywane na poczet propagacji wstecznej. Proces opisany w tym punkcie nazywamy propagacją w przód.
- Po dotarciu do warstwy wyjściowej algorytm oblicza błąd pomiędzy wartościami oczekiwanyymi a wartościami otrzymanymi za pomocą sieci.

- Następnie oblicza w jakim stopniu wartości wyjściowe z każdego neuronu przyczyniły się do powstania błędy. Robi to poprzez zastosowanie reguły łańcuchowej – matematycznej reguły pozwalającej na obliczanie pochodnych funkcji złożonych w oparciu o twierdzenie o pochodnej funkcji złożonej.
- Algorytm dokonuje modyfikacji wag w sposób analogiczny do tego użytego w metodzie gradientu prostego – jeśli wpływ połączenia na błąd był niewielki to modyfikacja wagi przy tym połączeniu także będzie niewielka i *vice versa*.

Proces uczenia sieci neuronowej wymaga zastosowania algorytmu optymalizacyjnego – tzw. optimizera – który odpowiada za aktualizację wag sieci w kierunku minimalizacji funkcji kosztu. Najprostszym i najczęściej omawianym przykładem jest algorytm gradientu prostego, jednak w praktyce często stosuje się jego ulepszone wersje, takie jak Adam, RMSprop czy Adagrad, które dynamicznie dostosowują kroki uczenia się na podstawie historii gradientów i wartości funkcji kosztu. Wybór odpowiedniego optimizera ma istotny wpływ na szybkość konwergencji modelu oraz jakość uzyskanych wyników, dlatego stanowi ważny element projektowania i trenowania sieci neuronowej.

Sieci neuronowe można zastosować zarówno do zagadnień związanych z regresją jak i do zagadnień polegających na klasyfikacji danych. Jako że w tej pracy będą one wykorzystane do regresji to przykład przedstawiony w następnym punkcie tyczy się tego właśnie tematu.

3. Dane elipsometryczne – analiza

Dane elipsometryczne analizowane w ramach tej pracy zostały wygenerowane za pomocą programu służącego do analizy i generowania danych elipsometrycznych firmy J.A. Wollam – WVASE. Zostały one wygenerowane poprzez utworzenie struktury symulującej warstwę spełniającą model Cauchy'ego znajdującą się na warstwie krzemu o grubości 1mm. Podczas generowania danych zakładano, że grubość symulowanych próbek – oznaczona w dalszej części tekstu jako T – będzie zawierać się w zakresie od 1 nm do 200 nm, współczynnik A w zakresie od 1.25 do 1.65, współczynnik B w zakresie od 0.001 do 0.02 oraz współczynnik C w zakresie od 0.00001 do 0.0002. Wygenerowane dane eksportowano do plików tekstowych,

które w swoich nazwach zawierały wartości współczynników Cauchy'ego dla których zostały wygenerowane. Na przykład nazwę pliku 1.5771_1.5601_0.0018113_0.000167165.txt możemy zinterpretować jako nazwę pliku zawierającego dane dla hipotetycznej warstwy o grubości 1.5771 nm, oraz współczynnikach Cauchy'ego:

$$A = 1.5601$$

$$B = 0.0018113$$

$$C = 0.000167165$$

Dane wygenerowane dla każdej kombinacji wielkości współczynników Cauchy'ego po wyeksportowaniu do pliku tekstowego przybierały formę przedstawioną na rysunku poniżej:

	65	65	70	70	75	75
300	32.939	153.51	29.769	144.06	25.889	128.11
310	31.462	154.19	27.913	144.61	23.619	127.82
320	30.676	154.91	26.913	145.35	22.364	128.17
330	30.215	155.85	26.3	146.47	21.532	129.28
340	30.002	156.94	25.985	147.88	21.024	131
350	30.118	158.3	26.064	149.75	20.941	133.67
360	30.808	160.95	26.807	153.53	21.507	139.53
370	30.723	167.13	26.494	162	20.512	151.86
380	28.941	171.73	24.145	168.19	17.261	160.49
390	27.167	173.74	21.896	170.86	14.395	163.9
400	25.81	174.49	20.21	171.81	12.333	164.66
410	24.815	174.91	18.986	172.3	10.867	164.77

Rysunek 7 – Przykładowy wygląd wygenerowanych danych elipsometrycznych

Pierwsza kolumna odpowiada poszczególnym długościom fali dla których dokonywany byłby pomiar. Kolejne 6 kolumn odpowiadają wygenerowanym wartościom PSI i DELTA dla trzech różnych kątów pomiarowych – 65° , 70° i 75° . Od lewej do prawej kolumny te reprezentują – długość fali, PSI dla 65° , DEL dla 65° , PSI dla 70° , DEL dla 70° , PSI dla 75° , DEL dla 75° . Program WVASE nie daje możliwości szybkiego generowania danych dla większych ilości próbek, z racji tego, podczas generowania danych wspomagano się skryptem typu makro napisanym w języku Python, którego zadaniem było uzupełnianie odpowiednich rubryk zadanimi wartościami, generowanie nazw plików oraz eksportowanie danych do plików tekstowych. Wygenerowano kilka zestawów danych. Pliki należące do pierwszego z nich zawierały w sobie tylko wartości PSI. Zestaw ten otrzymał nazwę Si_jaw i składał się z 675

plików. Wartości parametrów T, A, B, C na podstawie których generowane były pliki otrzymywane były za pomocą zagnieżdżonych pętli for. Dużym minusem tego rozwiązania był fakt, że zbiór otrzymanych danych charakteryzował się dużą różnorodnością zmiennej która modyfikowana była w najbardziej zagnieżdżonej pętli for, i dużo mniejszą różnorodnością dla pozostałych zmiennych – przynajmniej w przypadku niewykonania do końca wszystkich 4 stopni pętli. Rozwiązanie to było więc albo zbyt czasochłonne, albo oferowało dostęp do danych o ograniczonej różnorodności kombinacji. Problem ten udało się ominąć poprzez wykorzystanie biblioteki random, w celu utworzenia wielu losowych kombinacji wartości T, A, B, C z zadanych przedziałów. Metoda ta została zastosowana przy tworzeniu drugiego zbioru danych o nazwie Si_jaw_delta. Drugi utworzony zbiór danych także był wadliwy. Wada była spowodowana niedopasowaniem niektórych wartości zmiennych B i C dla których generowane były dane do tych które podane były w nazwie pliku. Miało to związek z nieprawidłowym kasowaniem uprzednio wprowadzonych wartości parametrów B i C. Problem ten udało się zażegnać poprzez zmodyfikowanie skryptu służącego do generacji danych w taki sposób, by kasowanie wartości parametrów B i C przebiegało w sposób prawidłowy. W celu oceny jakości modelu, zdecydowano się na użycie współczynnika determinacji R^2 . Wybrano go ze względu na jego intuicyjną interpretację oraz powszechnie zastosowanie w analizie regresji. Współczynnik R^2 informuje, jaka część całkowitej zmienności zmiennej objaśnianej jest wyjaśniana przez model, co pozwala na bezpośrednią ocenę stopnia dopasowania modelu do danych. Dodatkowo, jest to miara bezwymiarowa, co ułatwia porównywanie jakości różnych modeli niezależnie od skali danych. Dzięki tym właściwościom R^2 stanowi praktyczne i efektywne narzędzie w analizie skuteczności modeli predykcyjnych. Współczynnik R^2 obliczyć można korzystając z następującego wzoru:

Dalsza część pracy zajmująca się analizą danych elipsometrycznych zostanie napisana z wykorzystaniem programu Jupyter Notebook, ze względu na fakt, że umożliwia on integrację tekstu oraz kodu, co pozwoli na lepsze przedstawienie i wytlumaczenie operacji wykonywanych na danych.

Regresja Liniowa - Implementacja i wyniki

Do reprezentacji danych elipsometrycznych użyto dwóch klas: training_sample oraz training_dataset. Klasa training_sample opisuje pojedynczy zestaw danych dotyczący jednej próbki. Zawiera on informacje o wartościach PSI oraz DELTA dla długości fal od 300 do 1000 z krokiem 10nm. Klasa training_dataset opisuje zaś zbiór próbek, które można wykorzystać do uczenia modeli. Klasy te zakładają, że dane na temat próbki - tj. grubość, oraz współczynniki A, B, C są zawarte w nazwie pliku, z którego dane są wczytywane, zgodnie ze wzorem opisanym w poprzednim akapicie. W celu zachowania czytelności tekstu oraz przedstawienia działania funkcji znajdujących się w poszczególnych klasach, każda zastosowana funkcja, której powstanie było wynikiem tej pracy zostanie przedstawiona przed wywołaniem jej w kodzie. Ze względu na brak natywnego wsparcia Jupyter Notebook dla importowania innych notatników do notatnika obecnie używanego, zastosowano bibliotekę `importnb`, która pozwala na importowanie innych notatników jako modułów Pythona. W celu poprawnego działania należy zainstalować tę bibliotekę, co można zrobić za pomocą polecenia `pip install importnb`. W celu dokonania analizy zbioru danych utworzono instancję klasy training_dataset. Konstruktor klasy training_dataset zamieszczono poniżej.

```
In [1]: def __init__(self, dataset_folder, col):
    self.samples = self.gather_samples(dataset_folder, col)
```

Jak widać funkcja ta przyjmuje jako argumenty datasets_folder - lokalizację zbioru danych które chcemy poddać analizie - oraz col - listę kolumn, które znajdują się będą w pojedynczej próbce należącej do zbioru danych. Konstruktor wywołuje należącą do tej samej klasy - gather_samples - która z kolei wywołuje funkcję konstruktora klasy training_sample dla każdego z plików tekstowych znajdujących się w folderze wskazanym w jako parametr konstruktora klasy training_dataset. Podczas tworzenia obiektu typu training_sample, dane zawarte w korespondującym do niego pliku zapisywane są w formie zmiennej typu dataframe, a wartości grubości - T - oraz parametrów A, B, C pozyskiwane są z nazwy pliku i zapisywane jako zmienny typu float. Opisywane funkcje zamieszczone poniżej.

```
In [2]: #funkcja gather_samples z klasy training_dataset
def gather_samples(self, dataset_folder, col):
    all_items = os.listdir(dataset_folder)
    files = [item for item in all_items if os.path.isfile(os.path.join(dataset_f
samples = []
for i in files:
    sample = training_sample(os.path.join(dataset_folder, i), col)
    samples.append(sample)
```

```

    return samples
#konstruktor klasy training_sample
def __init__(self, file_path, col = ['wavelength', 'psi65', 'del65', 'psi70', 'd
    # Initialize the training_sample object by decoding the file name to extract
    # and loading the data from the file.
    self.T = self.decode_sample(file_path)[0]
    self.A = self.decode_sample(file_path)[1]
    self.B = self.decode_sample(file_path)[2]
    self.C = self.decode_sample(file_path)[3]
    self.data = self.load_data(file_path, self.T, self.A, self.B, self.C, col)

#funkcja decode_sample z klasy training_sample odpowiada za odczytywanie wartości
def decode_sample(self, filename):
    filename = os.path.basename(filename)
    info = filename.split("_")
    T = info[0]
    A = info[1]
    B = info[2]
    C = info[3]
    C = C.removesuffix(".txt")
    return [T, A, B, C]

#funkcja load_data z klasy training_sample odpowiada za zapisywanie danych odczy
#w formie dataframe

def load_data(self, filename, T, A, B, C, col):
    dataHelper = pd.read_csv(filename, sep='\t', header=None, index_col=False)
    dataHelper = dataHelper.drop(index=[0])
    dataHelper = dataHelper.iloc[:, :-1]
    dataHelper.columns = col
    dataHelper['T'] = T
    dataHelper['A'] = A
    dataHelper['B'] = B
    dataHelper['C'] = C
    return dataHelper

```

Przed rozpoczęciem pracy konieczne jest zimportowanie odpowiednich bibliotek i klas. Biblioteka import_ipynb jest wykorzystywana do importowania innych notatników jako modułów Pythona, co pozwala na korzystanie z funkcji i klas zdefiniowanych w tych notatnikach. Klasa training_dataset zawiera kod opisujący zbiór danych, na którym pracowano. Klasa locations pozwala na łatwy dostęp do ścieżek do plików istotnych dla projektu. Biblioteka os pozwala na wykonywanie operacji na ścieżkach do plików.

```
In [1]: import import_ipynb
import training_dataset as td
import locations as l
import os
```

Pozyskujemy ścieżkę do interesującego nas zbioru danych oraz tworzymy obiekt training_dataset na jego podstawie.

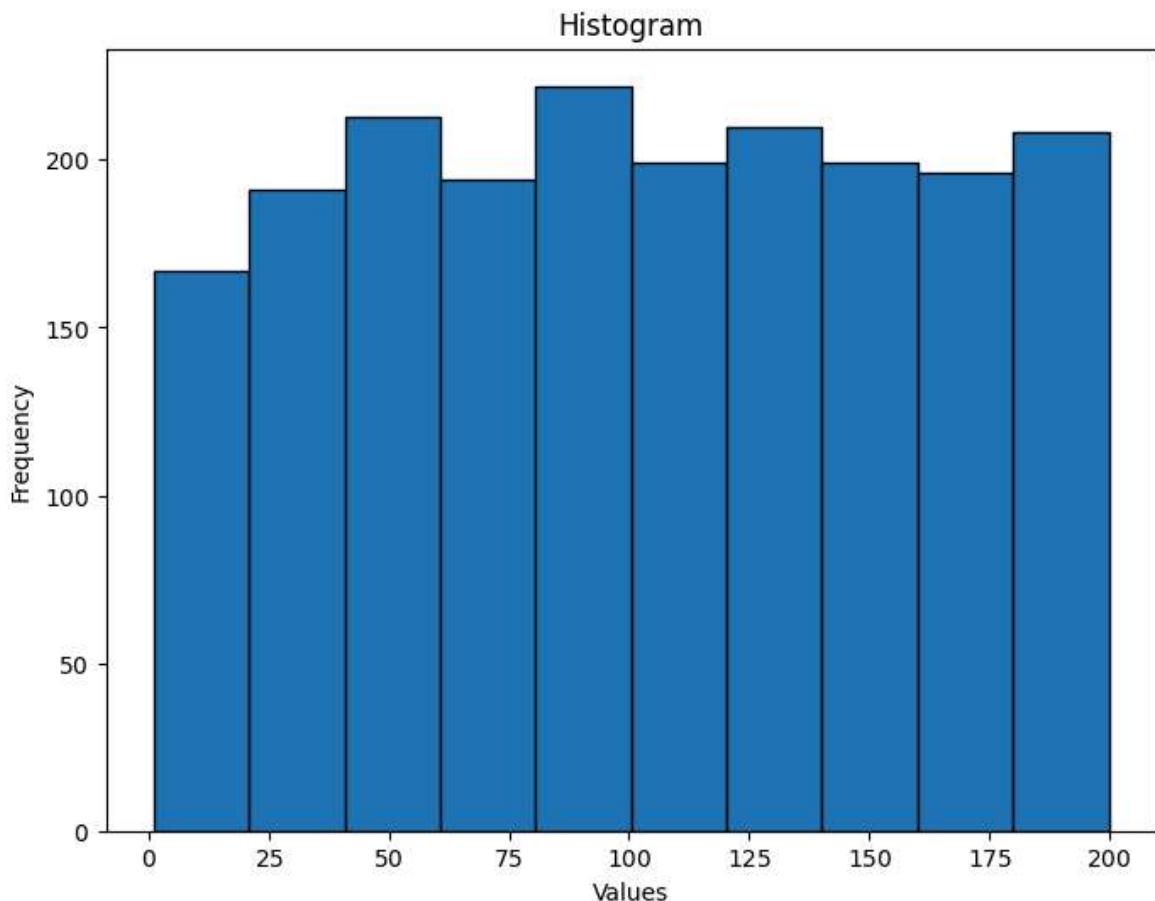
```
In [2]: l.locations.list_datasets()
```

```
Out[2]: ['.ipynb_checkpoints',
          'half_new_Si_jaw_delta',
          'lifesat',
          'new_Si_jaw_delta',
          'quarter_new_Si_jaw_delta',
          'R2_comparison_2.ipynb',
          'si02',
          'Si_jaw',
          'Si_jaw_delta']
```

```
In [3]: datasets_folder = l.locations.get_data_dir()
specific_dataset_location = os.path.join(datasets_folder, "new_Si_jaw_delta")
t_dataset = td.training_dataset(specific_dataset_location, col = ['wavelength',
```

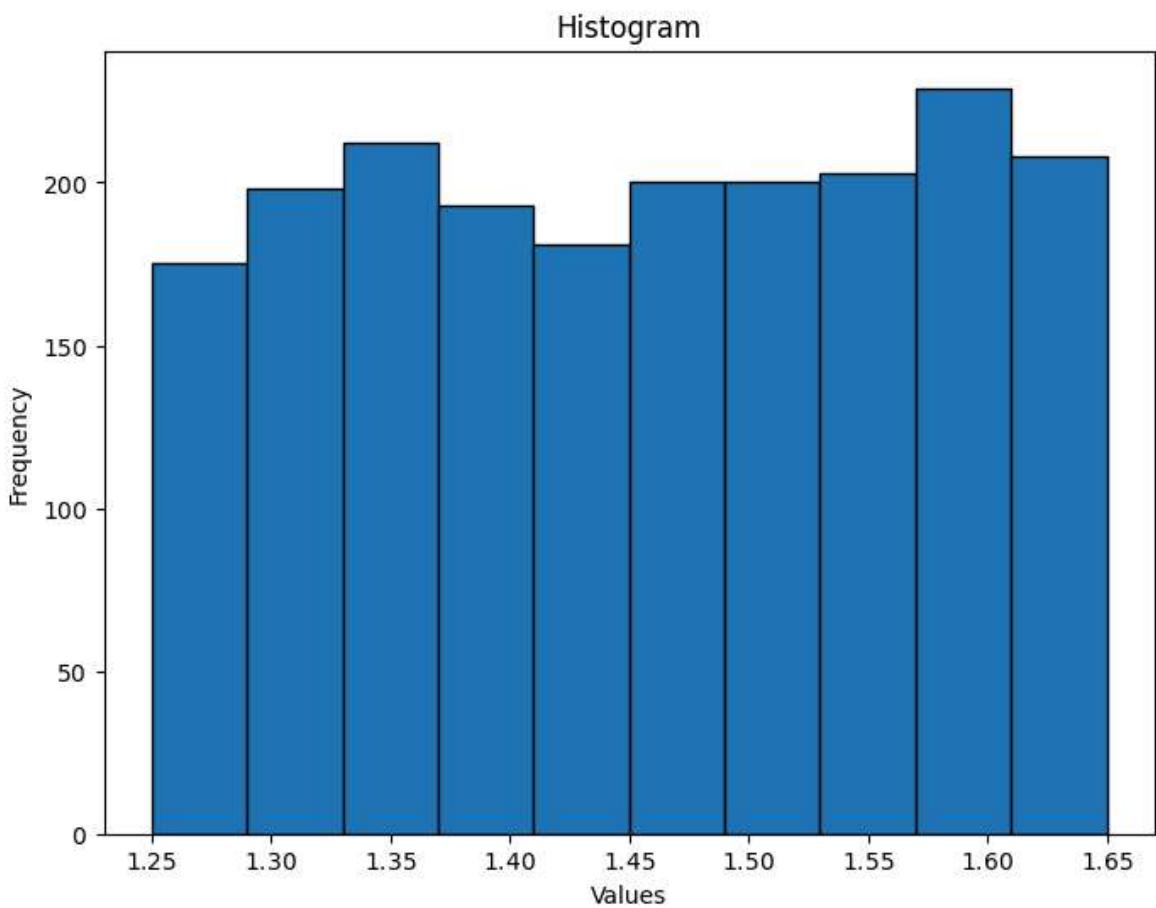
Za pomocą funkcji plot_data() zawartej w klasie training_dataset możemy zwizualizować dane zawarte w zbiorze danych. Funkcja ta przyjmuje jako argumenty nazwy interesujących nas parametrów. Następnie pobiera one interesujące nas dane z każdej instancji training_sample i prezentuje je w postaci histogramu

```
In [6]: t_dataset.plot_histogram(["T"])
```



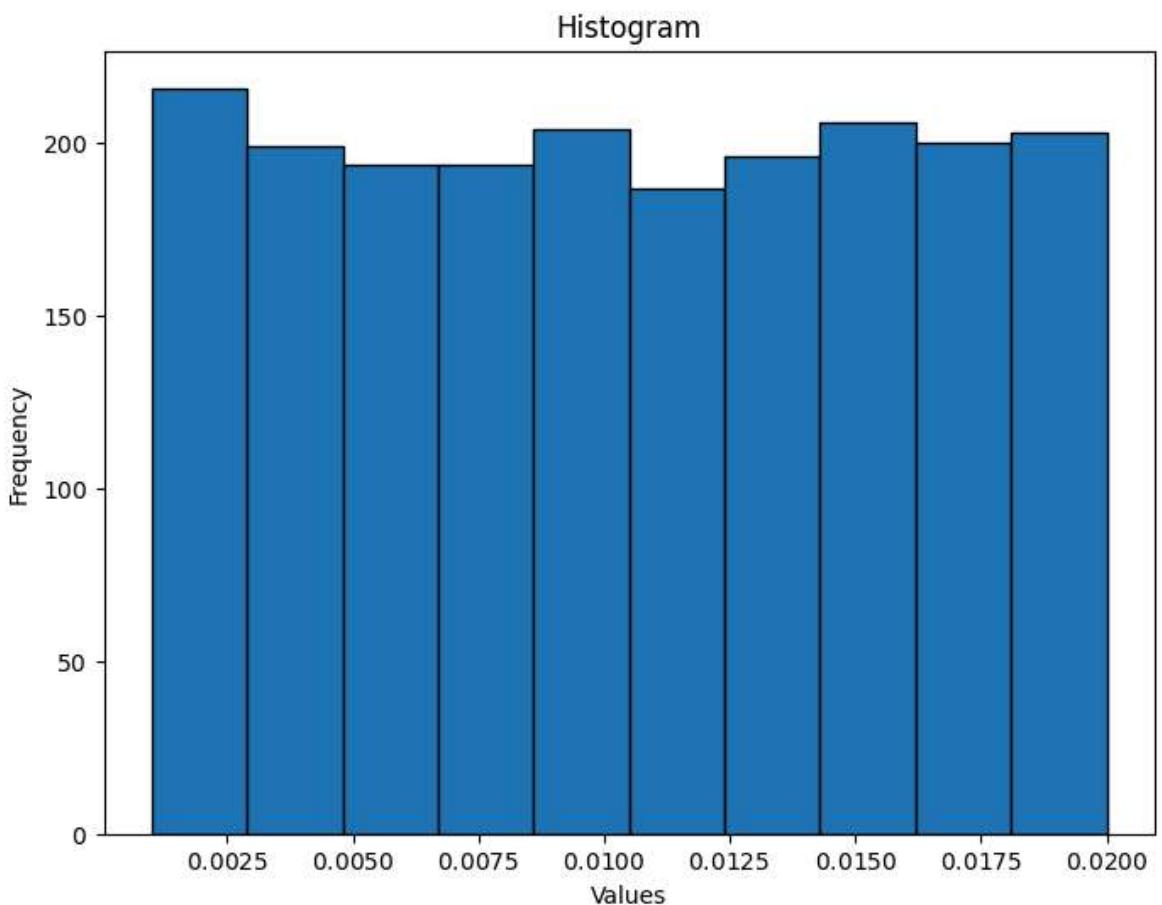
Rysunek 8 - Rozkład wartości grubości T w zbiorze danych

```
In [7]: t_dataset.plot_histogram(["A"])
```



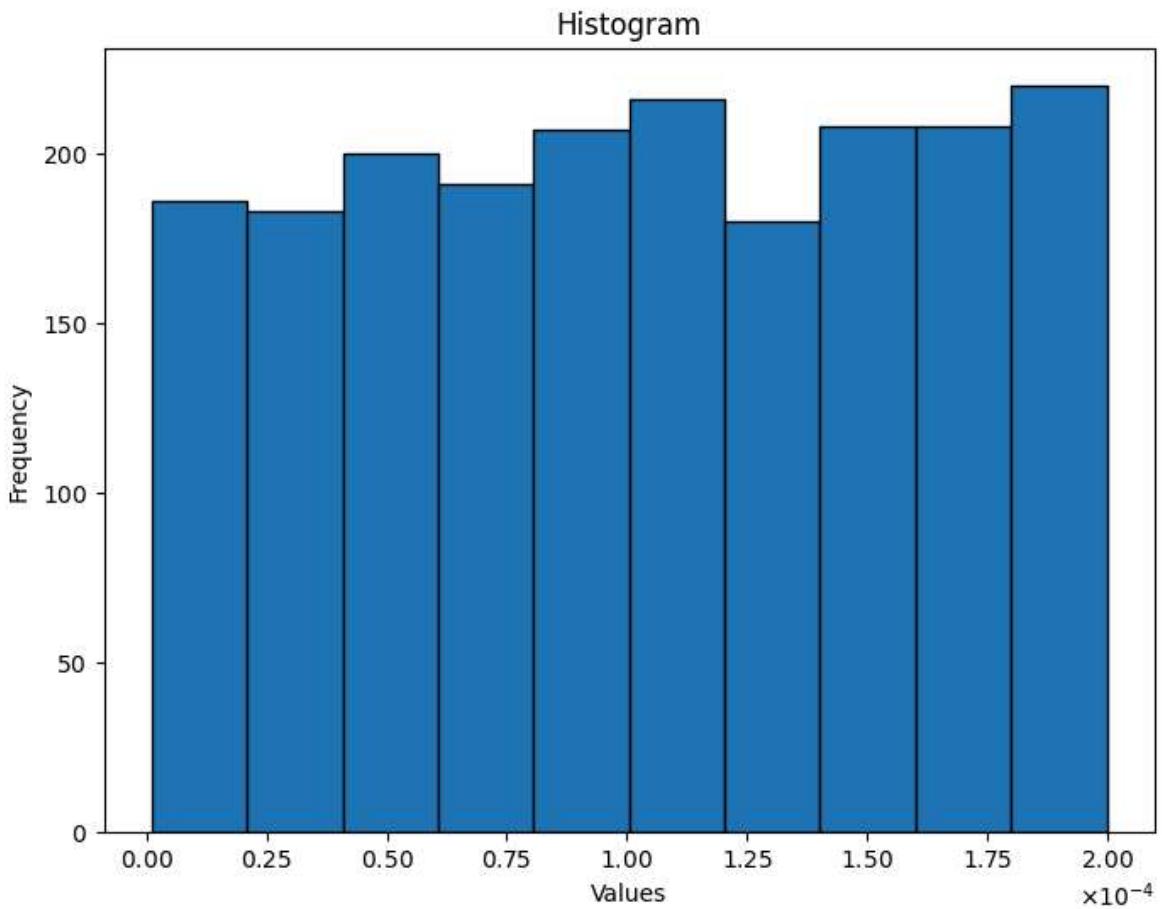
Rysunek 9 - Rozkład wartości współczynnika A w zbiorze danych

```
In [8]: t_dataset.plot_histogram(["B"])
```



Rysunek 10 - Rozkład wartości współczynnika B w zbiorze danych

```
In [9]: t_dataset.plot_histogram(["C"])
```



Rysunek 11 - Rozkład wartości współczynnika C w zbiorze danych

Jak widać dane rozłożone są równomiernie, co sprawia, że nie jest konieczne zastosowanie algorytmów służących do usuwania odstających wartości. Przystępujemy do analizy danych za pomocą regresji liniowej. Analizy dokonujemy na dwa sposoby - pierwszy sposób polega na analizie poszczególnych wierszy zbioru danych - tj. podstawową jednostką, która posłuży do wykonywania inferencji będzie pojedynczy wiersz należący do próbki. Wiersz ten będzie się składać z długości fali oraz wartości PSI i DELTA dla trzech różnych kątów padania wiązki. Drugi sposób polega na jednoczesnej analizie całej próbki - tj. podstawową jednostką będzie pojedyncza próbka, która będzie się składać z długości fali oraz wartości PSI i DELTA dla trzech kątów padania. Dane przedstawione w ten sposób w dalszej części pracy nazywane będą danymi spłaszczonymi. W celu przedstawienia próbki w ten sposób użyto funkcji `return_as_flat_df` należącej do klasy `training_sample` oraz funkcji o zblżonej konstrukcji, które zwracają dane innego typu np. zwracając je w formie tensorów bądź listy zmiennych typu float. Funkcję `return_as_flat_df` przedstawiono poniżej:

```
In [1]: def return_as_flat_df(self, feature_columns = ['wavelength', 'psi65', 'del65', ''],
                           target_columns = ['T', 'A', 'B', 'C']):
    features = self.data[feature_columns]
    targets = self.data[target_columns]
    targets = targets.iloc[:1]
    features = features.values.reshape(1, -1)
    targets = targets.values.reshape(1, -1)
    features = pd.DataFrame(features)
```

```

targets = pd.DataFrame(targets)
return features, targets

```

Przyjmuje ona jako argumenty listę kolumn, z których wartości będziemy chcieli użyć podczas uczenia jako etykiety oraz listę kolumn, z których wartości będziemy chcieli użyć jako cechy. Funkcja ta zwraca dwie ramki danych - pierwsza zawiera cechy, druga etykiety. Wartości w ramkach danych są spłaszczone, co oznacza, że każda próbka jest reprezentowana jako pojedynczy wiersz. W obu przypadkach będziemy chcieli przewidzieć grubość warstwy SiO₂ oraz współczynniki A, B, C. Regresję liniową przeprowadzimy za pomocą biblioteki scikit-learn, która jest jedną z najpopularniejszych bibliotek do uczenia maszynowego w Pythonie. Po utworzeniu instancji klasy, możemy ją przeprowadzić za pomocą funkcji, lin_reg, flat_lin_reg, lasso_reg, flat_lasso_reg, ridge_reg, flat_ridge_reg. Regresje grzbietowa [ang. "ridge"] i lasso przyjmują jako parametr także wartość alpha. Wartość ta wpływa na funkcję kosztu poprzez penalizację wysokich wartości współczynników regresji, co pozwala na uniknięcie przeuczenia modelu. Wzory na funkcje kosztu regresji lasso oraz grzbietowej przedstawione są poniżej:

$$\text{Funkcja kosztu regresji lasso: } \frac{1}{2n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 + \alpha \sum_{j=1}^p |\beta_j|$$

Równanie 11

$$\text{Funkcja kosztu regresji grzbietowej: } \sum_{i=1}^n (y_i - \hat{y}_i)^2 + \alpha \sum_{j=1}^p \beta_j^2$$

Równanie 12

Funkcję przeprowadzającą regresję typu lasso dla próbek przedstawionych jako pojedyncze wiersze przedstawiono poniżej.

```

In [11]: def flat_lasso_reg(self, feature_columns=['wavelength', 'psi65', 'del65', 'p
          # Split the dataset into training and testing sets
          data = self.flat_df_split(feature_columns, target_columns, test_size, ra

          # Initialize Lasso regression model
          lasso_reg = Lasso(alpha=alpha)

          # Fit the Lasso regression model on the training data
          lasso_reg.fit(data[0], data[2])

          # Predict on train and test sets
          pred_train = lasso_reg.predict(data[0])
          pred_test = lasso_reg.predict(data[1])

          # Calculate R^2 coefficients
          r2_train = r2_score(data[2], pred_train)
          r2_test = r2_score(data[3], pred_test)

          # Print R^2 coefficients

```

```

print(f"Train R2: {r2_train}")
print(f"Test R2: {r2_test}")

# Return R2 coefficients as a dictionary
return {"train_r2": r2_train, "test_r2": r2_test, "coefficients": lasso_

```

Funkcja ta przyjmuje jako argument listę kolumn, z których wartości będziemy chcieli użyć podczas uczenia jako cechy, listę kolumn, z których wartości będziemy chcieli użyć jako etykiety, rozmiar zbioru testowego, wartość random_state oraz wartość alpha. Funkcja ta dzieli zbiór danych na zbiór treningowy i testowy, następnie tworzy model regresji lasso, uczy go na zbiorze treningowym i dokonuje predykcji na zbiorze treningowym i testowym. Na końcu funkcja ta zwraca współczynnik determinacji R² dla zbioru treningowego i testowego oraz współczynniki regresji. Poniżej zamieszczono tabele zawierające wyniki różnych rodzajów regresji dla wybranych wartości alpha oraz dla różnych wartości poszukiwanych. Przedstawiono jako OLS - ordinary least square. Regresja przeprowadzaną na próbkach spłaszczonych została opisana jako "flat".

T						
alpha	R ²					
	OLS - treningowy	OLS - testowy	flat OLS - treningowy	flat OLS - testowy	lasso - treningowy	lasso - testowy
0,0000	0,3500	0,3500	0,9995	0,8447	0,3506	0,3506
0,0001	-	-	-	-	0,3506	0,3506
0,0010	-	-	-	-	0,3506	0,3506
0,0100	-	-	-	-	0,3506	0,3506
0,1000	-	-	-	-	0,3506	0,3506
1,0000	-	-	-	-	0,3500	0,3499
10,0000	-	-	-	-	0,3499	0,3497
alpha	R ²					
	flat lasso - treningowy	flat lasso - testowy	ridge - treningowy	ridge - testowy	flat ridge - treningowy	flat ridge - testowy
0,0000	0,9973	0,9958	0,3506	0,3506	0,9960	0,8396
0,0001	0,9973	0,9958	0,3506	0,3506	0,9995	0,8990
0,0010	0,9973	0,9958	0,3506	0,3506	0,9994	0,9627
0,0100	0,9973	0,9959	0,3506	0,3506	0,9994	0,9726
0,1000	0,9971	0,9960	0,3506	0,3506	0,9993	0,9845
1,0000	0,9967	0,9965	0,3506	0,3506	0,9992	0,9946
10,0000	0,9931	0,9931	0,3506	0,3506	0,9990	0,9973

Rysunek 12 - Wyniki regresji dla różnych rodzajów regresji oraz wartości alpha w przypadku wyznaczania grubości T.

A						
alpha	R ²					
	OLS - treningowy	OLS - testowy	flat OLS - treningowy	flat OLS - testowy	lasso - treningowy	lasso - testowy
0,0000	0,1642	0,1670	0,7884	0,7485	0,1642	0,1670
0,0001	-	-	-	-	0,1642	0,1670
0,0010	-	-	-	-	0,1640	0,1669
0,0100	-	-	-	-	0,1625	0,1654
0,1000	-	-	-	-	0,1420	0,1439
1,0000	-	-	-	-	0,0889	0,0895
10,0000	-	-	-	-	0,0000	0,0000
alpha	R ²					
	flat lasso - treningowy	flat lasso - testowy	ridge - treningowy	ridge - testowy	flat ridge - treningowy	flat ridge - testowy
0,0000	0,7884	0,7485	0,1642	0,1670	0,7357	-5,1031
0,0001	0,7884	0,7493	0,1642	0,1670	0,9339	-3,6888
0,0010	0,7815	0,7556	0,1642	0,1670	0,9309	-0,1342
0,0100	0,7208	0,7229	0,1642	0,1670	0,9266	0,4902
0,1000	0,6855	0,6863	0,1642	0,1670	0,9223	0,5188
1,0000	0,4677	0,4860	0,1642	0,1670	0,9168	0,5963
10,0000	0,0000	0,0000	0,1642	0,1670	0,8927	0,7010

Rysunek 13 - Wyniki regresji dla różnych rodzajów regresji oraz wartości alpha w przypadku wyznaczania współczynnika A.

alpha	B					
	R^2					
	OLS - treningowy	OLS - testowy	flat OLS - treningowy	flat OLS - testowy	lasso - treningowy	lasso - testowy
0,0000	0,3500	0,3500	0,2874	0,0529	0,0056	0,0057
0,0001	-	-	-	-	0,0055	0,0057
0,0010	-	-	-	-	0,0048	0,0048
0,0100	-	-	-	-	0,0022	0,0021
0,1000	-	-	-	-	0,0000	0,0000
1,0000	-	-	-	-	0,0000	0,0000
10,0000	-	-	-	-	0,0000	0,0000
alpha	R ²					
	flat lasso - treningowy	flat lasso - testowy	ridge - treningowy	ridge - testowy	flat ridge - treningowy	flat ridge - testowy
0,0000	0,2874	0,0529	0,0056	0,0057	0,2934	-44,7725
0,0001	0,2435	0,0723	0,0056	0,0057	0,7649	-28,5122
0,0010	0,1456	0,0850	0,0056	0,0057	0,7430	-6,9546
0,0100	0,0506	0,0620	0,0056	0,0057	0,7167	-4,5540
0,1000	0,0000	0,0000	0,0056	0,0057	0,6858	-9,3705
1,0000	0,0000	0,0000	0,0000	0,0000	0,6204	3,1880
10,0000	0,0000	0,0000	0,0000	0,0000	0,4735	-0,0717
100,0000	0,0000	0,0000	0,0000	0,0000	0,3546	0,0332
1000,0000	0,0000	0,0000	0,0000	0,0000	0,2745	0,0457
10000,0000	0,0000	0,0000	0,0000	0,0000	0,2745	0,0457

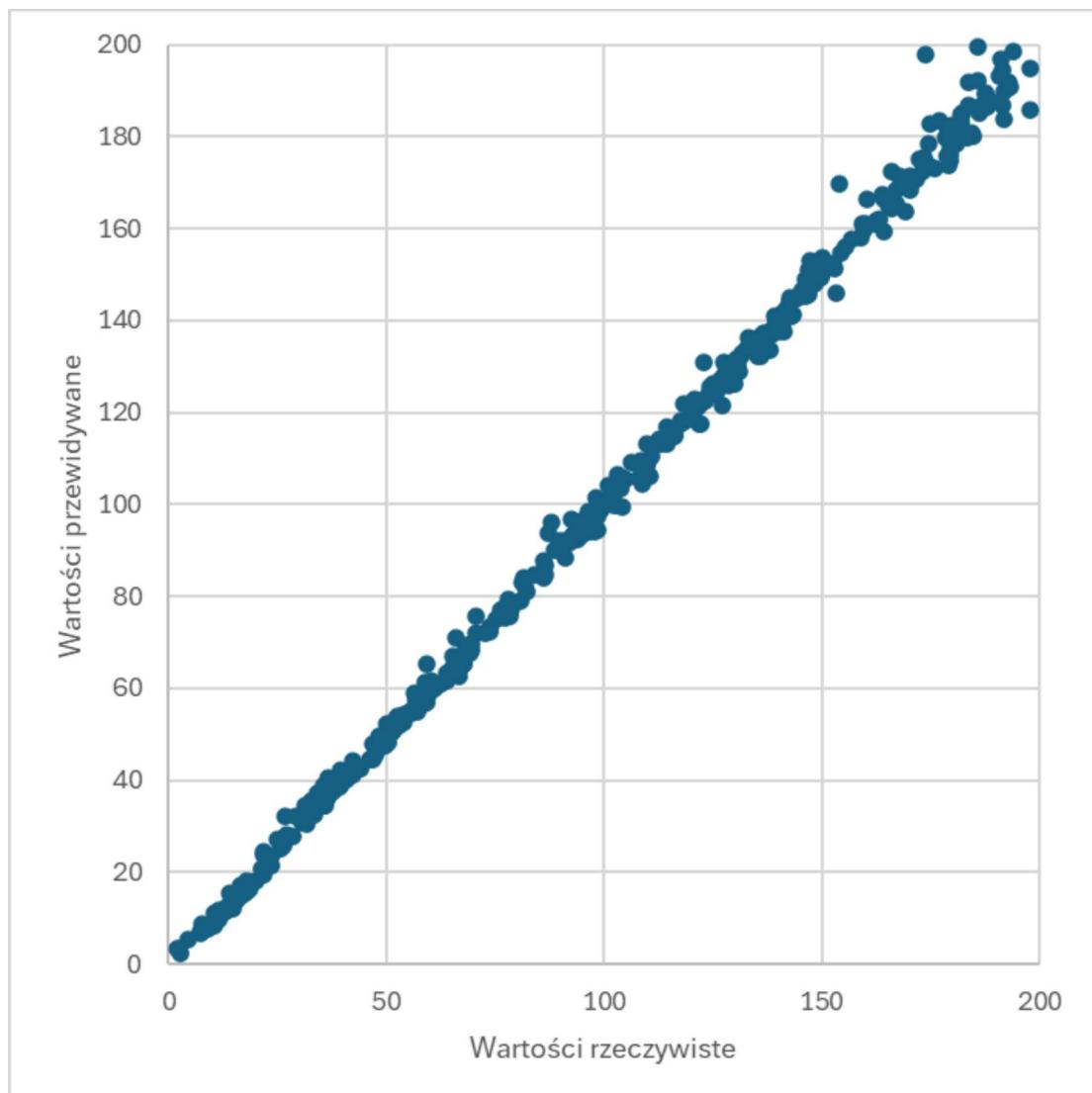
Rysunek 14 - Wyniki regresji dla różnych rodzajów regresji oraz wartości alpha w przypadku wyznaczania współczynnika B.

alpha	C					
	R^2					
	OLS - treningowy	OLS - testowy	flat OLS - treningowy	flat OLS - testowy	lasso - treningowy	lasso - testowy
0,0000	0,0000	0,0000	0,1409	-0,2103	0,0000	0,0000
0,0001	-	-	-	-	0,0000	0,0000
0,0010	-	-	-	-	0,0000	0,0000
0,0100	-	-	-	-	0,0000	0,0000
0,1000	-	-	-	-	0,0000	0,0000
1,0000	-	-	-	-	0,0000	0,0000
10,0000	-	-	-	-	0,0000	0,0000
alpha	R^2					
	flat lasso - treningowy	flat lasso - testowy	ridge - treningowy	ridge - testowy	flat ridge - treningowy	flat ridge - testowy
0,0000	0,0000	0,0000	0,0000	0,0000	-0,8024	-6,9740
0,0001	0,0000	0,0000	0,0000	0,0000	0,2574	-4,6058
0,0010	0,0000	0,0000	0,0000	0,0000	0,2457	-4,8218
0,0100	0,0000	0,0000	0,0000	0,0000	0,2367	-3,7773
0,1000	0,0000	0,0000	0,0000	0,0000	0,2254	-1,9863
1,0000	0,0000	0,0000	0,0000	0,0000	0,2114	-1,4707
10,0000	0,0000	0,0000	0,0000	0,0000	0,1891	-0,7254
100,0000	0,0000	0,0000	0,0000	0,0000	0,1629	-0,4021
1000,0000	0,0000	0,0000	0,0000	0,0000	0,1407	-0,2671
10000,0000	0,0000	0,0000	0,0000	0,0000	0,1223	-0,1694

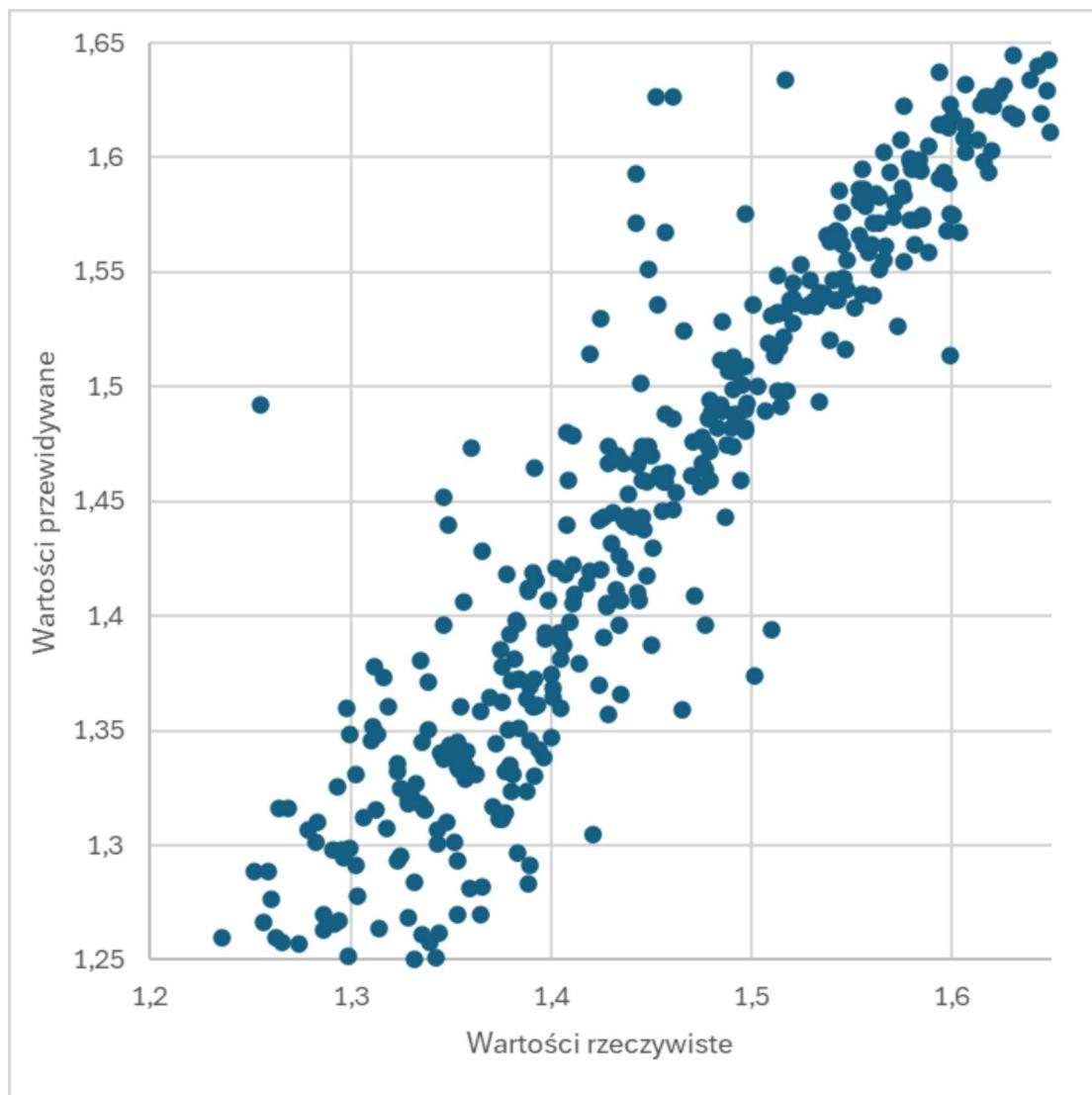
Rysunek 15 - Wyniki regresji dla różnych rodzajów regresji oraz wartości alpha w przypadku wyznaczania współczynnika C.

Ostatecznie, wyniki uzyskane za pomocą regresji uznano za niezadowalające. Dobre wyniki uzyskano jedynie przy wyznaczaniu wartości grubości za pomocą próbek spłaszczonych. Współczynnik determinacji dla wyników uzyskanych za pomocą tej metody był zbliżony do 0,99 zarówno dla regresji grzbietowej jak i dla regresji typu lasso. Wyniki uzyskiwane dla przewidywań dokonywanych za pomocą pojedynczych wierszy były słabe w każdym przypadku. Najlepsze wyniki dla tej metody uzyskano przy przewidywaniu grubości - współczynnik determinacji dla tych wyników wynosił w przybliżeniu 0,35. Pewną liniową zależność można było zauważać także podczas przewidywania współczynnika A, jednak współczynnik determinacji dla tych wyników wynosił około 0,1. W przypadku przewidywania współczynników B i C uzyskano wyniki

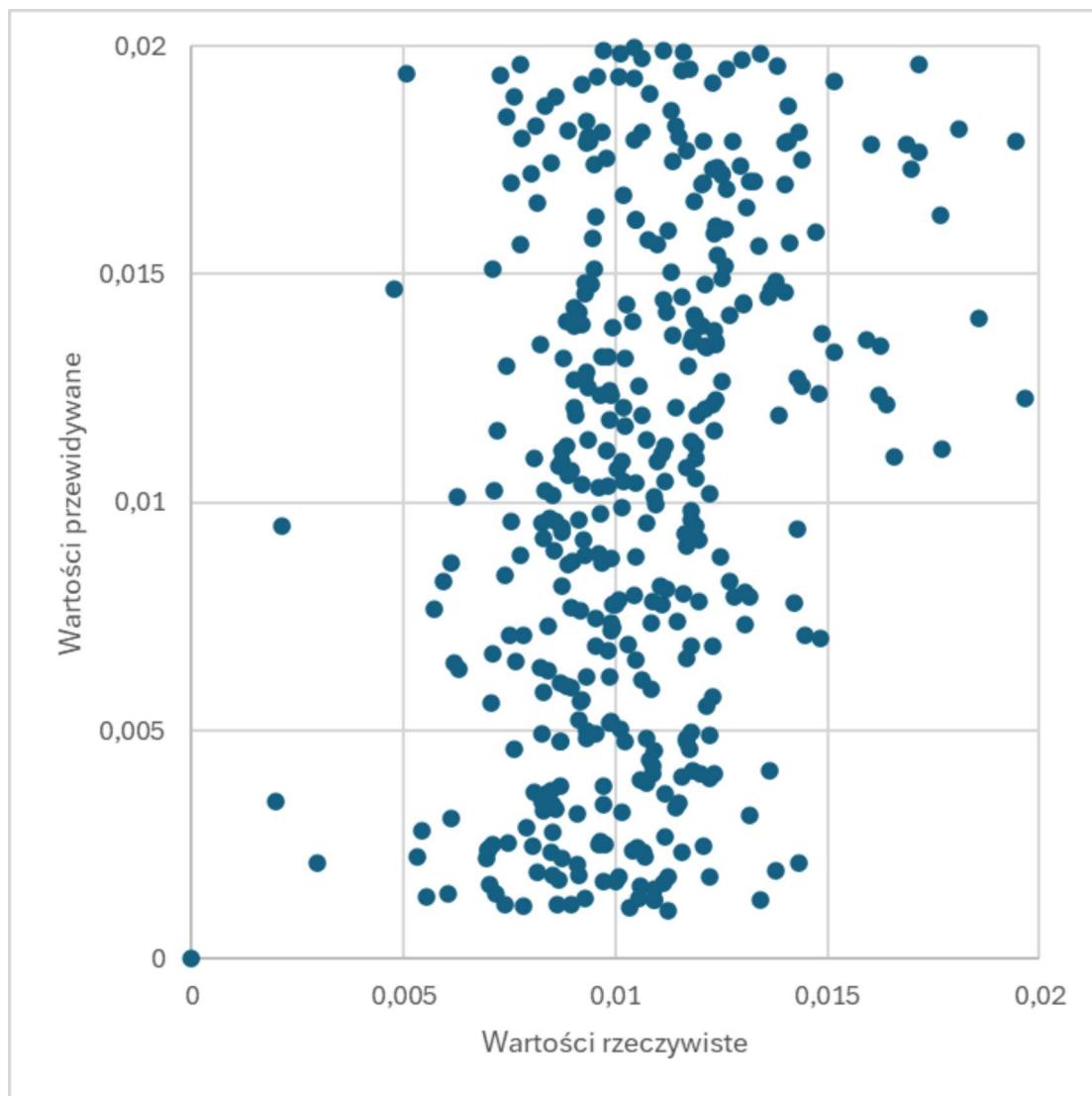
bliskie zeru. Wartym zauważenia wydaje się być fakt, że najlepiej z przewidywaniami zdaje się radzić sobie regresja liniowa grzbietowa z dużymi wielkościami współczynnika alpha. Może to wskazywać na fakt, że w celu przewidzenia wielkości A i B z odpowiednią dokładnością, konieczne jest kierowanie się szerokim zakresem danych o niewielkim, aczkolwiek niezerowym stopniu ważności. Poniżej zamieszczono wykresy rozproszenia dla regresji, które uzyskały najlepsze rezultaty dla każdej z poszukiwanych wartości z wyjątkiem C, jako że dla tej wartości najlepszy uzyskany współczynnik korelacji dla zbioru testowego miał wartość ujemną, co oznacza, że model został przeuczony na tyle silnie, że zależność między wartościami była odwrotna niż oczekiwana.



Rysunek 16 - Zależność wartości rzeczywistych od wartości przewidywanych dla grubości T.



Rysunek 17 - Zależność wartości rzeczywistych od wartości przewidywanych dla współczynnika A.



Rysunek 18 - Zależność wartości rzeczywistych od wartości przewidywanych dla współczynnika B.

Wyniki uzyskane za pomocą różnych wariantów regresji liniowej uznano za niedostateczne. Być może zastosowanie innych wariacji regresji tj. regresja wielomianowa, pozwoliłoby na uzyskanie lepszych wyników. Na potrzeby realizacji tej pracy jednak, zdecydowano się na zastosowanie sieci neuronowych, które pozwalają na modelowanie bardziej złożonych zależności. W kolejnej sekcji przedstawiono implementację sieci neuronowych oraz wyniki uzyskane za ich pomocą.

Sieci Neuronowe - Implementacja i wyniki

W niniejszej pracy zdecydowano się na wykorzystanie biblioteki PyTorch jako głównego narzędzia do implementacji i trenowania sieci neuronowych. PyTorch jest otwartoźródłowym frameworkiem rozwijanym przez Facebook AI Research, który zyskał

dużą popularność w środowisku naukowym i przemysłowym ze względu na swoją elastyczność oraz intuicyjność. Cechy te umożliwiają łatwe prototypowanie i modyfikowanie modeli, co jest szczególnie istotne w kontekście badań eksperymentalnych, gdzie często konieczne jest testowanie różnych architektur i podejść. Pytorch zapewnia także wsparcie dla akceleracji obliczeń z użyciem GPU za pomocą CUDA. Z racji na to, że karta graficzna używana do trenowania modeli jest kartą RADEON, w celu uzyskania odpowiedniego oprogramowania należało przeprowadzić szkolenie na systemie operacyjnym Linux, gdzie zainstalowano sterowniki AMD ROCm. W celu przeprowadzenia szkolenia na systemie Windows należałoby skorzystać z oprogramowania AMD ROCm for Windows, które jest wciąż w fazie rozwoju i nie jest jeszcze w pełni stabilne. W niniejszej pracy zdecydowano się na wykorzystanie PyTorch w wersji 2.0.1, która jest najnowszą stabilną wersją dostępną w momencie pisania tej pracy. W celu obsługi i zarządzania budową sieci neuronowej oraz przebiegiem procesu trenowania utworzono dwie klasy - klasę MLP opisaną w notatniku model_creator oraz klasę train_model opisaną w notatniku model_training. Z wykorzystaniem tych klas utworzono odpowiednie funkcje zawarte w klasie training_dataset. Pytorch oferuje dużą swobodę w doborze sposobu konstrukcji sieci neuronowej oraz w sposobie przeprowadzania jej szkolenia. Z racji tego, klasy MLP oraz model_training zostaną omówione poniżej.

Klasa MLP - Implementacja sieci neuronowej

Struktura klasy MLP z wyłączeniem przynależących do niej funkcji statycznych została przedstawiona poniżej:

```
In [ ]: import os
import torch
import import_ipynb
import torch.nn as nn
import locations as l

class MLP(nn.Module):
    def __init__(self, input_size, output_size, hidden_layers, activation_fn=nn.

        super(MLP, self).__init__()
        if len(hidden_layers) > 7:
            raise ValueError("The number of hidden layers cannot exceed 7.")

        self.layers = nn.ModuleList()
        prev_size = input_size

        # Create hidden layers
        for neurons in hidden_layers:
            self.layers.append(nn.Linear(prev_size, neurons))
            self.layers.append(activation_fn()) # Add the specified activation
        prev_size = neurons

        # Create output layer
        self.layers.append(nn.Linear(prev_size, output_size))
```

```
def forward(self, x):
    for layer in self.layers:
        x = layer(x)
    return x
```

Klasa ta dziedziczy po "nn.Module", co jest standardową praktyką w PyTorch do definiowania modeli sieci neuronowych. Konstruktor klasy przyjmuje następujące argumenty:

- **input_size**: liczba cech wejściowych, które będą podawane do sieci neuronowej. W przypadku analizy opartej na poszczególnych wierszach pochodzących z próbek, będzie ona przyjmować wartość 7. Cechami wtedy będą długość fali oraz wartości PSI i DELTA dla trzech kątów padania wiązki. W przypadku analizy opartej na próbkach spłaszczonych, będzie ona przyjmować wartość 497, ponieważ każda próbka będzie zawierać 7 cech dla każdej z trzech kątów padania wiązki dla 71 długości fali ($7 * 71 = 497$).
- **output_size**: liczba wartości, które chcemy przewidzieć. W przypadku przewidywania poszczególnych etykiet, będzie to liczba 1, ponieważ przewidujemy tylko jedną wartość - grubość, bądź jeden z parametrów A,B,C. Istnieje też możliwość przewidywania większej ilości parametrów jednocześnie - wtedy **output_size** będzie przyjmować wartość równą liczbie przewidywanych parametrów.
- **hidden_layers**: lista liczb całkowitych określających liczbę neuronów w każdej warstwie ukrytej. Maksymalna liczba warstw ukrytych wynosi 7, co jest ograniczeniem narzuconym przez konstruktor klasy. Każda warstwa może zawierać w sobie dowolną liczbę neuronów.
- **activation_fn**: funkcja aktywacji, która będzie stosowana w warstwach ukrytych. Domyślnie jest to LeakyReLU, ale można użyć dowolnej innej funkcji aktywacji dostępnej w PyTorch, np. ReLU, Sigmoid, Tanh itp. LeakyReLU jest funkcją aktywacji, która jest podobna do ReLU, ale pozwala na przepływ niewielkiej ilości sygnału dla wartości ujemnych, co może pomóc w uniknięciu problemu "zanikającego gradientu".

Funkcja **forward** jest odpowiedzialna za propagację w przód sieci neuronowej. Przechodzi przez wszystkie warstwy zdefiniowane w konstruktorze i zwraca wynik końcowy. Klasa ta pozwala na łatwe tworzenie sieci neuronowych o różnych architekturach poprzez modyfikację liczby warstw ukrytych oraz liczby neuronów w każdej z nich. Dzięki zastosowaniu "nn.ModuleList" możliwe jest dynamiczne dodawanie warstw do modelu, co czyni tę klasę elastyczną i łatwą w użyciu.

Klasa **model_training** - Implementacja procesu trenowania modelu

Z racji na obszerność klasy model_training w pracy zostanie umieszczona wyłącznie jej część umożliwiająca szkolenie bez użycia partii.

```
In [5]: import os
import import_ipynb
import matplotlib.pyplot as plt
import time
import torch
import IPython
from IPython.core.display_functions import clear_output
import locations as l

def train_model(model, loss_fn, optimizer, x_train, y_train, x_test, y_test, mod
os.environ['TERM'] = 'xterm'
best_loss = float('inf')

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)

model = model.to(device)

model_folder = l.locations.get_models_dir()
save_path = os.path.join(model_folder, model_name)

x_train, y_train = x_train.to(device), y_train.to(device)
x_test, y_test = x_test.to(device), y_test.to(device)

def r2_loss(y_pred, y_true):
    ss_total = torch.sum((y_true - torch.mean(y_true)) ** 2)
    ss_residual = torch.sum((y_true - y_pred) ** 2)
    r2 = 1 - (ss_residual / ss_total)
    return 1 - r2 # Loss is 1 - R^2

while True:
    IPython.display.clear_output(wait=True)
    # Forward pass
    model.train()
    outputs = model(x_train)
    loss = loss_fn(outputs, y_train)

    # Backward pass and optimization
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    model.eval()
    with torch.no_grad():
        test_pred = model(x_test)
        test_loss = loss_fn(test_pred, y_test)
        r2_test_loss = r2_loss(test_pred, y_test)

    if loss.detach().item() < best_loss:
        best_loss = loss.item()
        torch.save(model.state_dict(), save_path)
```

```
        print(f"New best loss: {best_loss:.4f}. Model saved to {save_path}\n")
        print(f"Current Loss: {loss.item():.8f}, Test Loss: {test_loss.item():.8f}")
        print(f"Current R2 Loss: {r2_loss(outputs, y_train).item():.8f}, Tes
```

Klasa ta przyjmuje następujące argumenty:

- model: instancja klasy MLP, która będzie trenowana. Model ten powinien być zdefiniowany wcześniej i zawierać odpowiednią architekturę sieci neuronowej.
- loss_fn: funkcja straty, która będzie używana do oceny jakości modelu. Może to być np. Mean Squared Error (MSE) lub inna funkcja straty odpowiednia dla problemu regresji.
- optimizer: optymalizator, który będzie używany do aktualizacji wag modelu. Może to być np. Adam, SGD lub inny optymalizator dostępny w PyTorch.
- x_train, y_train: dane treningowe, które będą używane do trenowania modelu. x_train to tensor zawierający cechy wejściowe, a y_train to tensor zawierający etykiety (wartości docelowe).
- x_test, y_test: dane testowe, które będą używane do oceny modelu po każdej epoce treningu. x_test to tensor zawierający cechy wejściowe dla danych testowych, a y_test to tensor zawierający etykiety (wartości docelowe) dla danych testowych.
- model_name: nazwa modelu, która będzie używana do zapisania wytrenowanego modelu.
- batch_size: rozmiar partii, który będzie używany podczas trenowania modelu. Jeśli batch_size jest równy 0, model będzie trenowany na całych danych treningowych bez podziału na partie. Jeśli batch_size jest większy od 0, dane treningowe zostaną podzielone na partie o określonym rozmiarze.

Klasa ta umożliwia szkolenie modelu zarówno za pomocą procesora (ang. CPU), jak i akceleratora graficznego (ang. GPU). W przypadku dostępności GPU, model zostanie przeniesiony na urządzenie GPU, co przyspieszy proces trenowania. Funkcja ta zapisuje najlepszy model na podstawie wartości funkcji straty, co pozwala na zachowanie najlepszego modelu uzyskanego podczas treningu. Funkcja ta jest elastyczna i może być dostosowana do różnych architektur sieci neuronowych oraz różnych problemów regresyjnych. Daje ona również możliwość monitorowania procesu trenowania poprzez wyświetlanie aktualnej wartości funkcji straty oraz współczynnika determinacji R^2 . Proces trenowania jest realizowany w pętli, która kontynuuje się do momentu przerwania przez użytkownika. W każdej iteracji pętli model jest trenowany na danych treningowych, a następnie oceniany na danych testowych. Jeśli wartość funkcji straty dla modelu na danych treningowych jest mniejsza niż dotychczasowa najlepsza wartość, model jest zapisywany na dysku. Bieżące wyświetlane są wartości funkcji straty oraz współczynnika determinacji R^2 dla danych treningowych i testowych, pozwala na oszacowanie właściwego momentu przerwania procesu trenowania. Predykce dokonywane w celu oszacowania błędu na zbiorze testowym są dokonywane przy użyciu `torch.no_grad()`, co

oznacza że nie są one uwzględniane w obliczeniach gradientu, co przyspiesza proces trenowania i zmniejsza zużycie pamięci. Sprawia to również, że model nie uczy się na zbiorze testowym, co jest zgodne z zasadami uczenia maszynowego. Funkcje odpowiedzialne za trenowanie modelu na danym zestawie danych zostały umieszczone w klasie training_dataset. Funkcje te - funkcja train(), flat_train() oraz ich wariacje - przyjmują jako argumenty listę kolumn z których wartości będziemy chcieli użyć podczas uczenia jako cechy oraz listę kolumn z których wartości będziemy chcieli użyć jako etykiety. Przyjmują one także jako argumenty funkcję używaną do obliczenia strat oraz tabelę opisującą liczbę neuronów znajdujących się w poszczególnych warstwach. Z racji na fakt że podczas przeprowadzania badań użyto wielu architektur sieci neuronowych, przyjęto konwencję co do nazewnictwa modeli. W celu sprawnego generowania nazw modeli które zawierają informacje o architekturze sieci neuronowej, oraz o danych użytych podczas szkolenia utworzono funkcję generate_model_name. Funkcja ta została przedstawiona poniżej:

```
In [6]: def generate_model_name(self, feature_columns, target_columns, is_standardized):
    # Join feature and target column names
    features_part = "_".join(feature_columns)
    targets_part = "_".join(target_columns)

    # Add standardization information
    standardization_part = "standardized" if is_standardized else "non_standar

    # Add hidden layer information
    hidden_layers_part = "_".join(map(str, hidden_layers))

    # Combine all parts into the model name
    model_name = f"{prefix}_{features_part}_to_{targets_part}_{standardizati
    return model_name
```

Funkcja ta jest wywoływana wewnętrz metod odpowiedzialnych za trenowanie modeli zawartych w klasie training_dataset. Przyjmuje ona jako argumenty listę kolumn, z których wartości będziemy chcieli użyć podczas uczenia jako cechy, listę kolumn, z których wartości będziemy chcieli użyć jako etykiety, informację o tym czy dane zostały znormalizowane, oraz listę liczb całkowitych określających liczbę neuronów w każdej warstwie ukrytej. Funkcja ta zwraca nazwę modelu, zawierającą informacje na temat architektury sieci neuronowej oraz danych użytych podczas szkolenia. Nazwa modelu jest tworzona w formacie "model_{features}_to_{targets}_{standardization}_layers_{hidden_layers}.pth", gdzie {features} to nazwy cech wejściowych, {targets} to nazwy etykiet, {standardization} to informacja o tym czy dane zostały znormalizowane, a {hidden_layers} to liczba neuronów w poszczególnych warstwach ukrytych. Funkcja ta pozwala na łatwe generowanie nazw modeli, które są czytelne i zawierają istotne informacje na temat modelu. Na przykład nazwa modelu "model_wavelength_psi65_del65_psi70_del70_psi75_del75_to_A_non_standardized_layers_64_" oznacza model, który przewiduje wartość A na podstawie cech wejściowych takich jak długość fali oraz wartości PSI i DELTA dla trzech kątów padania wiązki. Dane użyte przy szkoleniu tego modelu nie były standaryzowane. Model taki składa się z trzech warstw

ukrytych, z których pierwsza zawiera 64 neurony, druga 32 neurony, a trzecia 16 neuronów. Modele które były trenowane z użyciem spłaszczonych próbek zawierają w nazwie modelu dodatkowy przedrostek "flat", co pozwala na łatwe odróżnienie ich od modeli trenowanych na pojedynczych wierszach. Przykładowa nazwa takiego modelu to "flat_model_wavelength_psi65_del65_psi70_del70_psi75_del75_to_A_non_standardized_layers". W celu oceny jakości modeli trenowanych z użyciem pojedynczych wierszy utworzono funkcje takie jak `get_median_r2_score` i `get_mean_r2_score`. Obliczają one medianę i średnią współczynnika determinacji R^2 dla przewidywań dokonywanych na wszystkich wierszach zawartych w danej próbce. Następnie obliczają one wartości R^2 dla wszystkich próbek zawartych w zbiorze danych. W celu oceny jakości modeli trenowanych z użyciem spłaszczonych próbek utworzono funkcje takie jak `flat_test_r2`. Funkcja ta dokonuje przewidywań na wszystkich próbkach zawartych w zbiorze danych, a następnie oblicza na nich podstawie współczynników determinacji R^2 .

Sieci neuronowe - rezultaty

Jak wspomniano w powyższym podpunkcie, sieci trenowano zarówno z wykorzystaniem poszczególnych wierszy jak i spłaszczonych próbek. W obu przypadkach stosowano różne architektury sieci neuronowych, jednak w przypadku szkolenia opartego na poszczególnych wierszach wyniki były dużo gorsze. W celu uzyskania miarodajnego porównania jakości poszczególnych sieci w tej pracy zdecydowano się na umieszczenie wyników tylko ich poszczególnych wersji. Pierwsze podjęte próby dotyczyły możliwości przewidywania wartości poszukiwanych parametrów na podstawie pojedynczych wierszy. W tym celu wyszkolono modele o architekturach $7 \times 64 \times 32 \times 16 \times 1$, $7 \times 32 \times 16 \times 8 \times 1$ oraz $7 \times 16 \times 8 \times 4 \times 1$. Zarówno wartości cech jak i wartości etykiet nie poddawano skalowania. Wszystkie modele trenowano przez okres jednej godziny, w celu uzyskania miarodajnego porównania. Jako funkcji strat użyto błędu średniokwadratowego, a zastosowanym optymizatorem był Adam. Podczas wielokrotnego powtarzania procesu uczenia zauważono, że uczenie przynosi lepsze rezultaty w przypadkach, gdy zastosowana prędkość uczenia jest niska. W celu miarodajnego porównania dla wszystkich wyuczonych modeli zastosowano prędkość uczenia równą 0,00001. Tabelę zawierającą współczynniki korelacji uzyskane dla zbiorów treningowych, testowych oraz dla zbiorów połączonych zamieszczono w tabeli poniżej:

topologia	etykieta											
	T			A			B			C		
	R2 treningowy	R2 testowy	R2 łączony	R2 treningowy	R2 testowy	R2 łączony	R2 treningowy	R2 testowy	R2 łączony	R2 treningowy	R2 testowy	R2 łączony
64x32x32x16	0,9493	0,9519	0,9498	0,7991	0,7964	0,7986	-3,3	-3,27	-3,29	-3,3	-3,28	-3,29
48x24x24x12	0,944	0,9454	0,9443	0,8162	0,8106	0,815	-0,003	-0,018	-0,006	-300	-297	-299
32x16x16x8	0,8935	0,8966	0,8942	0,7497	0,7481	0,7494	-0,007	-0,0021	-0,0019	-480	-500	-484

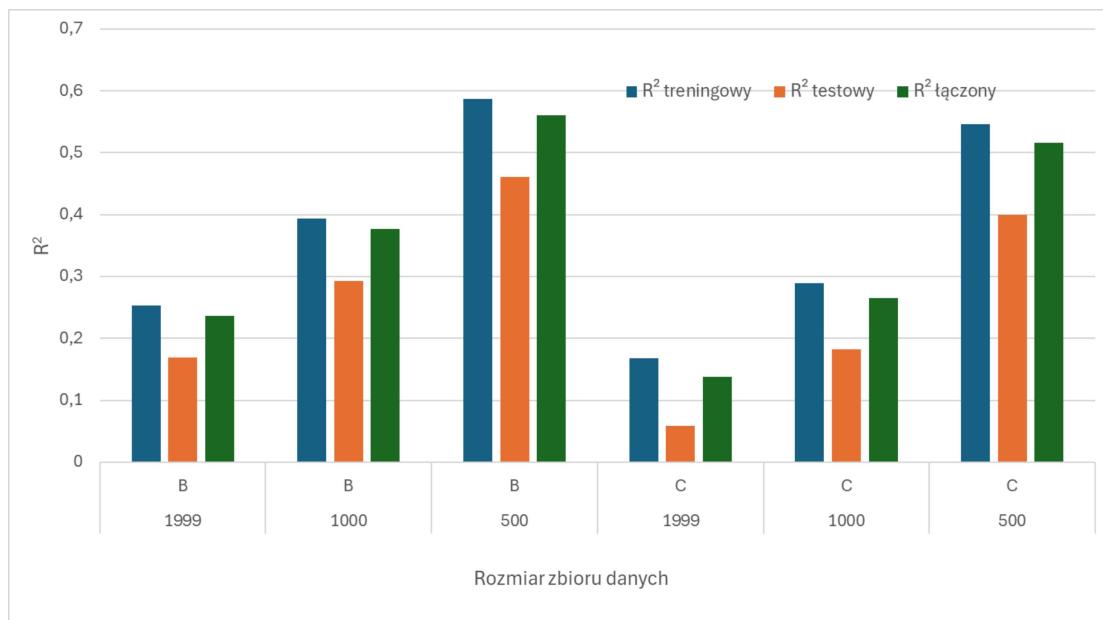
Rysunek 19 - Wyniki regresji dla różnych architektur sieci neuronowych w przypadku przewidywania wartości T, A, B, C na podstawie pojedynczych wierszy.

Powyższe wyniki dla trzech różnych topologii sieci ($64 \times 32 \times 32 \times 16$, $48 \times 24 \times 24 \times 12$, $32 \times 16 \times 16 \times 8$) wyraźnie pokazują, że modele o takiej samej architekturze radzą sobie

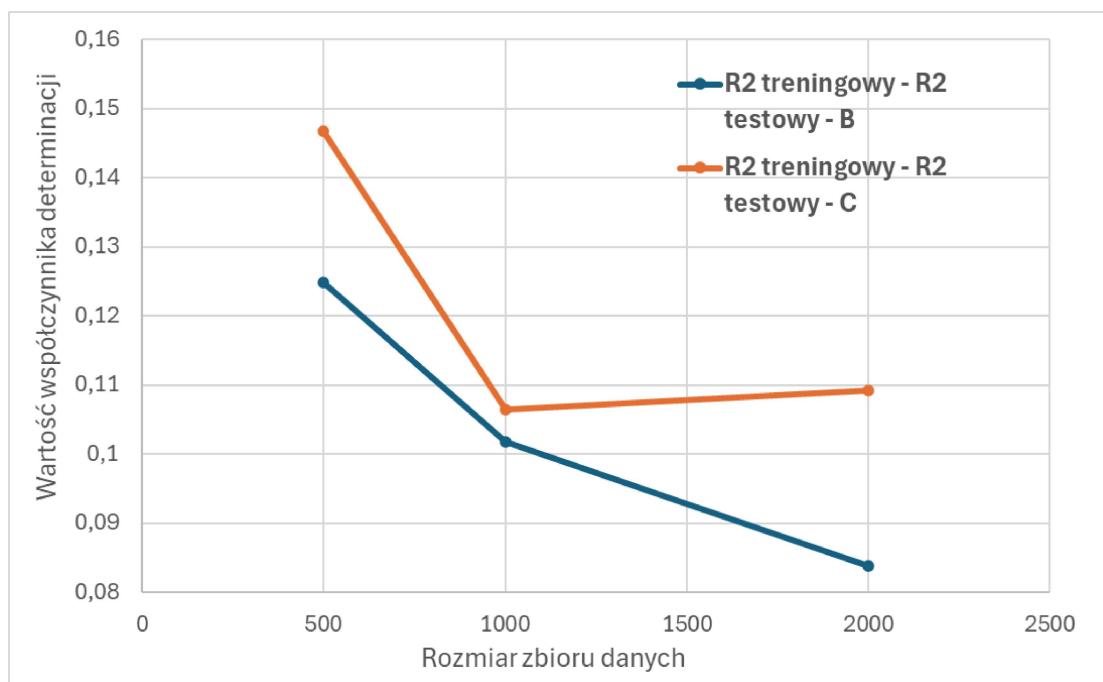
bardzo dobrze z predykcją etykiety T, nieco gorzej z A, a praktycznie nie uczą się etykiet B i C. Dla T wartości współczynnika determinacji zarówno na zbiorze treningowym, jak i testowym utrzymują się na poziomie ok. 0,89–0,95, przy czym różnice między wartościami treningowymi i testowymi są minimalne, co świadczy o braku przeuczenia i stabilnej zdolności generalizacji. W przypadku etykiety A R^2 wynosi ok. 0,75–0,82, co wskazuje na umiarkowanie dobrą jakość dopasowania; ponownie niewielka różnica między $R^2_{\text{treningowym}}$ i R^2_{testowym} sugeruje, że model nie ulega znacznemu przeuczeniu. Natomiast dla etykiet B oraz C wszystkie topologie osiągają wartości R^2 dalece ujemne (od około -3 w najszerzej sieci do nawet -480 w najmniejszej), zarówno na zbiorze treningowym, jak i testowym, co oznacza, że zachowanie modelu jest gorsze niż prosty model średniej. Dla B i C nie widać znaczcej poprawy topologiami, co świadczy o całkowitym niedouczeniu – sieć nie jest w stanie uchwycić żadnych zależności pomiędzy cechami a tymi etykietami. Można zatem wnioskować, że przewidywanie T i A wykorzystuje informacje zawarte w danych skutecznie, podczas gdy B i C wymagają albo zupełnie innego podejścia (np. modelu sekwencyjnego lub uwzględnienia dodatkowych cech), albo wskazują na istotnie niższą jakość lub brak wyjaśnialnych wzorców w danych dla tych etykiet. Jako jedną z możliwych przyczyn takiego stanu rzeczy rozważano fakt, że parametry B oraz C przyjmują niewielkie wartości. Może to powodować zaistnienie tzw. "problemu zanikającego gradientu", który jest częstym problemem w sieciach neuronowych, zwłaszcza gdy wartości etykiet są bardzo małe. W celu sprawdzenia tej hipotezy, przeprowadzono eksperyment polegający na skalowaniu wartości etykiet za pomocą StandardScaler zawartego w bibliotece scikit-learn. Skalowanie to polega na przekształceniu wartości etykiet tak, aby miały średnią równą 0 i odchylenie standardowe równe 1. Skalowaniu podlegały zarówno wartości etykiet jak i wartości cech. Użyte w tym celu scalery zapisano w folderze "scalers" znajdującym się w folderze z code_data_models. Czynności te wykonano tylko w celu zdiagnozowania możliwego problemu. W przypadku chęci zastosowania podobnego narzędzia w przyszłości, preferowanym rozwiązaniem byłoby zastosowanie scalera, który nie zmienia wartości etykiet, a jedynie przekształca je w taki sposób, aby były bardziej zrozumiałe dla modelu. Kolejnymi rozważanymi powodami uzyskiwania tak słabych wyników były niedostateczny rozmiar zbioru danych. W celu przetestowania tych hipotez jednocześnie, powtórzono proces szkolenia dla zbiorów przeskalowanych danych o różnych wielkościach - 500, 1000 i 2000 próbek. Wyniki przedstawiono poniżej.

rozmiar zbioru danych	Etykiety							
	B				C			
	R2 treningowy	R2 testowy	R2 łączony	ΔR^2	R2 treningowy	R2 testowy	R2 łączony	ΔR^2
1999	0,2529	0,1691	0,2369	0,0838	0,1678	0,0585	0,138	0,1093
1000	0,3943	0,2925	0,3764	0,1018	0,2893	0,1828	0,2649	0,1065
500	0,5864	0,4615	0,56	0,1249	0,5467	0,4	0,5157	0,1467

Rysunek 20 - Wyniki regresji dla różnych rozmiarów zbioru danych oraz różnych architektur sieci neuronowych w przypadku przewidywania wartości B oraz C na podstawie pojedynczych wierszy.



Rysunek 21 - Wyniki regresji dla różnych rozmiarów zbioru danych oraz różnych architektur sieci neuronowych w przypadku przewidywania wartości B oraz C na podstawie pojedynczych wierszy.



Rysunek 22 - Zależność przeuczenia modelu od rozmiaru zbioru danych dla etykiet B i C.

Jak widać na załączonych rysunkach, dla obu etykiet wraz ze zmniejszaniem rozmiaru zbioru obserujemy wyraźny wzrost wartości współczynnika determinacji zarówno na zbiorze treningowym, jak i testowym – najniższe R^2 (odpowiednio ok. 0,25 dla B i 0,17 dla C) występuje na największym zbiorze (1999 próbek), a najwyższe (0,59 dla B i 0,55 dla C) na najmniejszym (500 próbek). Jednocześnie różnice pomiędzy R^2 treningowym i testowym pozostają stosunkowo niewielkie, co świadczy o stabilnym uogólnianiu modelu bez nadmiernego przeuczenia. Zatem, mimo że bardziej rozbudowany zbiór danych

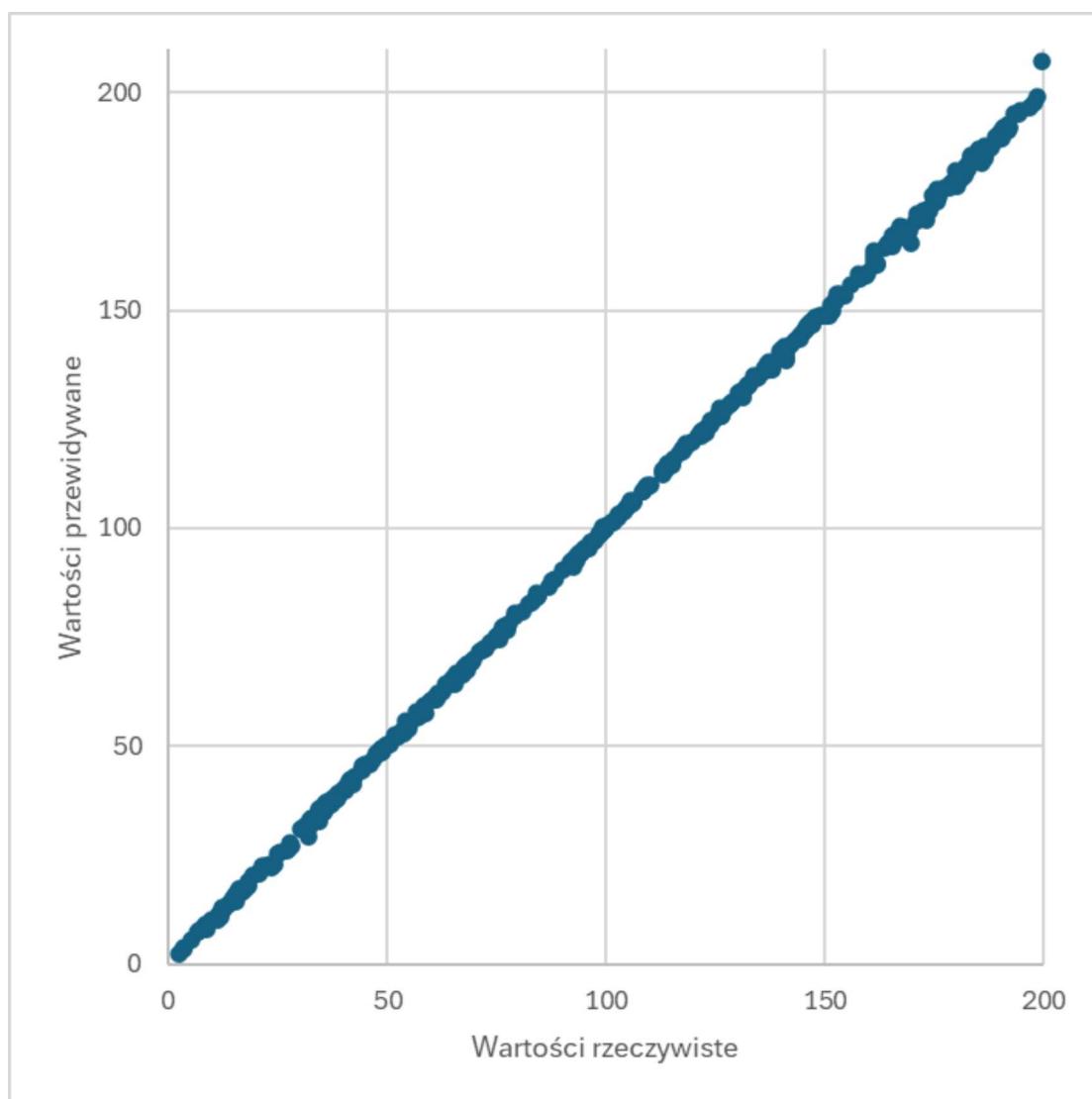
zazwyczaj sprzyja lepszej generalizacji, w omawianym przypadku mniejsza liczba próbek – po standaryzacji – pozwala modelowi osiągnąć wyższe dopasowanie, co może wynikać z obniżonej różnorodności szumu oraz lepszego wykrywania wzorców w mniej zróżnicowanym zbiorze. Może to sugerować konieczność zastosowanie większych, bardziej rozbudowanych sieci neuronowych, które być może będą w stanie uchwycić bardziej złożone zależności między cechami a etykietami B i C. Warto również zauważyć, że wartości R^2 dla etykiet B i C pozostają nadal stosunkowo niskie, co może sugerować, że te etykiety są trudne do przewidzenia na podstawie dostępnych cech. W celu dalszej analizy i poprawy wyników, warto rozważyć zastosowanie innych architektur sieci neuronowych, takich jak sieci konwolucyjne, które mogą lepiej radzić sobie z bardziej złożonymi zależnościami w danych.

W dalszej części pracy zdecydowano się na przeprowadzenie analizy z wykorzystaniem spłaszczonych próbek. W tym celu utworzono funkcję flat_train, która przyjmuje jako argumenty listę kolumn z których wartości będziemy chcieli użyć podczas uczenia jako cechy oraz listę kolumn z których wartości będziemy chcieli użyć jako etykiety. Modele trenowane w wersjach przewidującej wszystkie parametry - T, A, B, C - jednocześnie, a także w wersjach przewidującą wartości osobno. Proces uczenia przebiegał szybciej niż w wersji z pojedynczymi wierszami - można to wytlumaczyć znacznym zmniejszeniem rozmiaru zbioru treningowego, ponieważ każda próbka jest reprezentowana jako pojedynczy wiersz. W celu uzyskania miarodajnych wyników, modele trenowane przez okres połowy godziny. Z racji na większą liczbę użytych cech, zdecydowano się na nieznaczne zwiększenie rozmiarów stosowanych modeli. Zastosowano architektury $497 \times 128 \times 64 \times 32 \times 16 \times 8 \times 4$, gdzie 497 to liczba cech wejściowych, a 4 to liczba neuronów w warstwie wyjściowej. W celu uzyskania miarodajnych wyników, modele trenowane przez okres jednej godziny, w celu uzyskania miarodajnego porównania. Jako funkcji strat użyto błędu średniokwadratowego, a zastosowanym optymalizatorem był Adam. Podczas wielokrotnego powtarzania procesu uczenia zauważono, że uczenie przynosi lepsze rezultaty w przypadkach, gdy zastosowana prędkość uczenia jest niska. W celu miarodajnego porównania dla wszystkich wyuczonych modeli zastosowano prędkość uczenia równą 0,00001. W celu przetestowania efektów uczenia użyto funkcji flat_test_r2. Tabelę zawierającą współczynniki determinacji uzyskane dla zbiorów treningowych dla zbiorów połączonych zamieszczono poniżej:

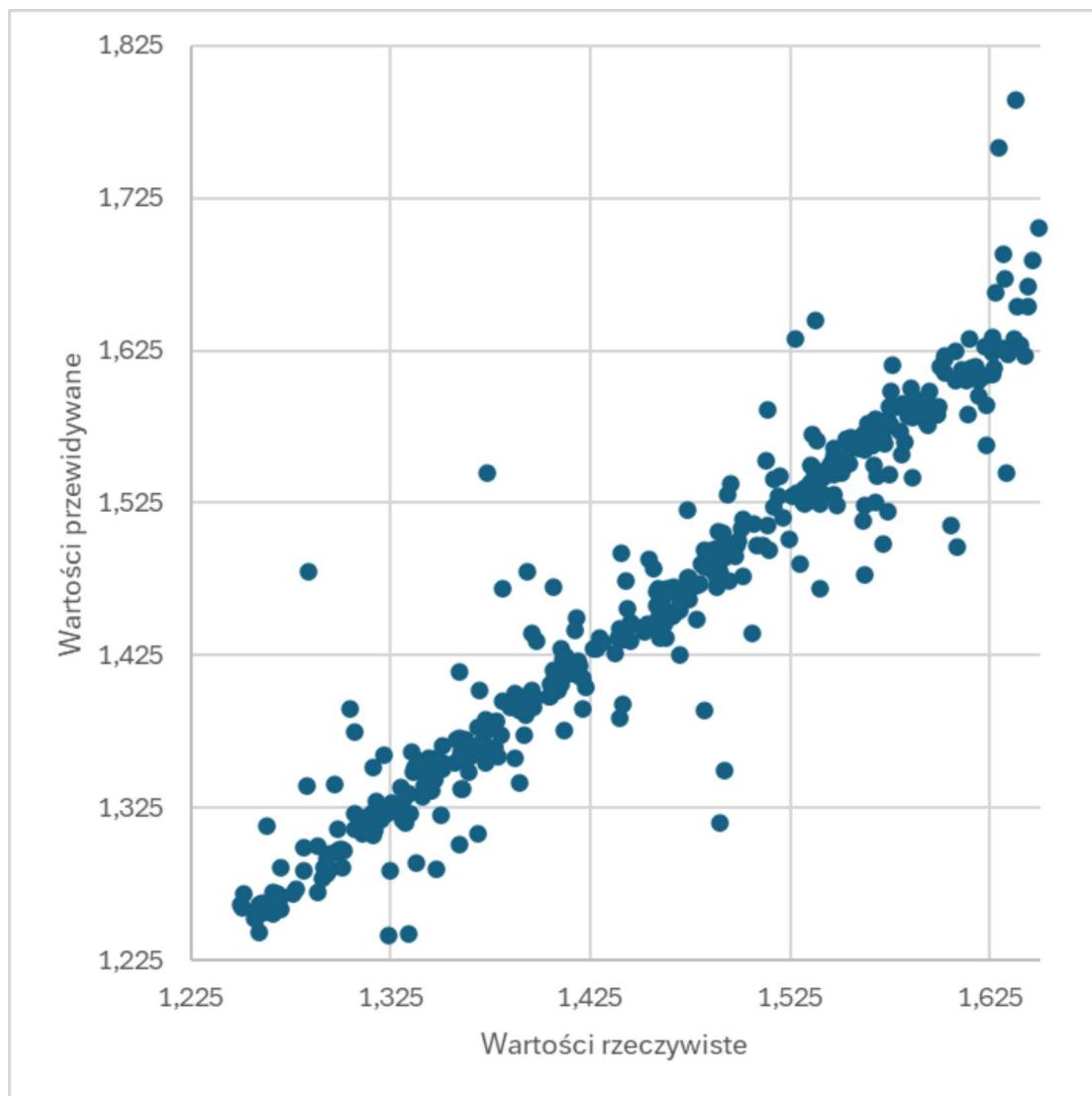
topologia	etykiety							
	T		A		B		C	
	R^2 treningowy	R^2 testowy						
128x64x32x16	0,9999	0,9997	0,9632	0,7739	0,9497	0	0	0
96x48x24x12	0,9999	0,9998	0,9648	0,8718	0,7514	0,045	0	0
72x36x18x9	0,9996	0,9994	0,9747	0,9197	0,6428	0,1231	0	0
48x24x12x6	0,9999	0,9997	0,9784	0,8677	0,9386	0,5124	0	0

Rysunek 23 - Wyniki regresji dla różnych architektur sieci neuronowych w przypadku przewidywania wartości T, A, B, C na podstawie spłaszczonych próbek.

Poniżej zamieszczono także wykresy rozproszenia dla najlepszych wyników uzyskanych dla wartości T i A.



Rysunek 24 - Wykres rozproszenia dla wartości T uzyskanych za pomocą sieci o architekturze 497x128x64x32x16x1.



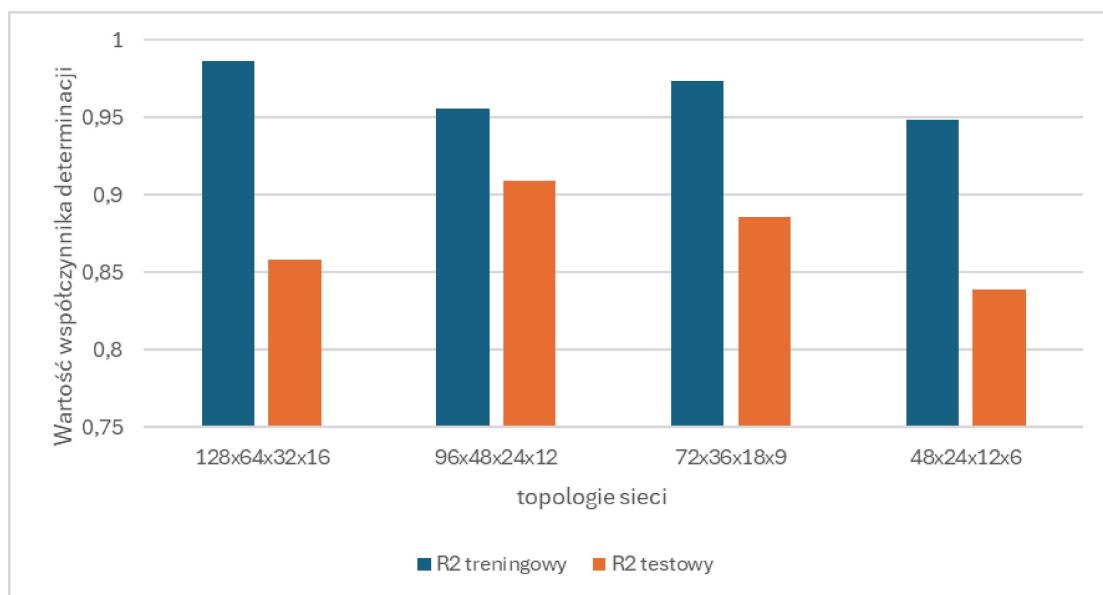
Rysunek 25 - Wykres rozproszenia dla wartości A uzyskanych za pomocą sieci o architekturze 497x72x36x18x9x1.

Jak możemy zauważyć, sieć wydaje uzyskuje dobre i bardzo dobre wyniki podczas przewidywania wartości grubości T oraz współczynnika A. W przypadku przewidywania wartości grubości T, współczynnik determinacji R^2 dla zbioru treningowego wynosi około 0,99, zarówno dla zbioru testowego jak i treningowego. W przypadku przewidywania wartości współczynnika A, współczynnik determinacji R^2 dla zbioru treningowego waha się w okolicach 0,9 dla obu zbiorów. Osiągnięcie optymalnych wyników wydaje się być kwestią zastosowania odpowiedniej architektury sieci oraz przerwania uczenia w odpowiednim momencie. Wskazane może być zastosowanie algorytmów wcześniego zatrzymywania, które pozwalają na przerwanie uczenia w momencie, gdy model przestaje się uczyć bądź zaczyna zachodzić przeuczenie. Pozwoliłoby to na uzyskanie optymalnego stosunku wartości R^2 dla zbioru treningowego do R^2 dla zbioru testowego. W przypadku przewidywania wartości B, współczynnik determinacji R^2 dla zbioru treningowego przyjmuje wartości powyżej 0,9, ale dla zbioru testowego wartości te są znacznie niższe i wynoszą około 0. W przypadku przewidywania wartości C, współczynnik determinacji R^2 zarówno dla zbioru treningowego, jak i testowego jest bliski零.

Ponownie postanowiono przeskalać wartości B i C aby sprawdzić czy będzie miało to wpływ na wyniki. W tym celu utworzono funkcji flat_train_1000 oraz flat_train_100000 które mnożą etykiety przez 1000 oraz 100000. Wartości te zostały dobrane w taki sposób, aby rzędy wartości etykiet B i C były porównywalne z rzędami wartości etykiet T i A. Dla wartości B przeprowadzono szkolenie z użyciem funkcji flat_train_1000, a dla wartości C z użyciem funkcji flat_train_100000. Utworzono także funkcje flat_test_1000 oraz flat_test_100000, które pozwalają na przeprowadzenie testów na zbiorach przeskalowanych. Wykonano także jedną próbę z wartościami A przemnożonymi przez 1000. Wyniki, oraz wykresy rozproszenia dla wybranych przypadków zamieszczone poniżej.

topologia	etykiety			
	B		C	
	R ² treningowy	R ² testowy	R ² treningowy	R ² testowy
128x64x32x16	0,9866	0,8586	0,5896	-0,101
96x48x24x12	0,9563	0,9093	0,3961	-0,5159
72x36x18x9	0,9735	0,886	0,453	-0,316
48x24x12x6	0,9486	0,8389	0	0

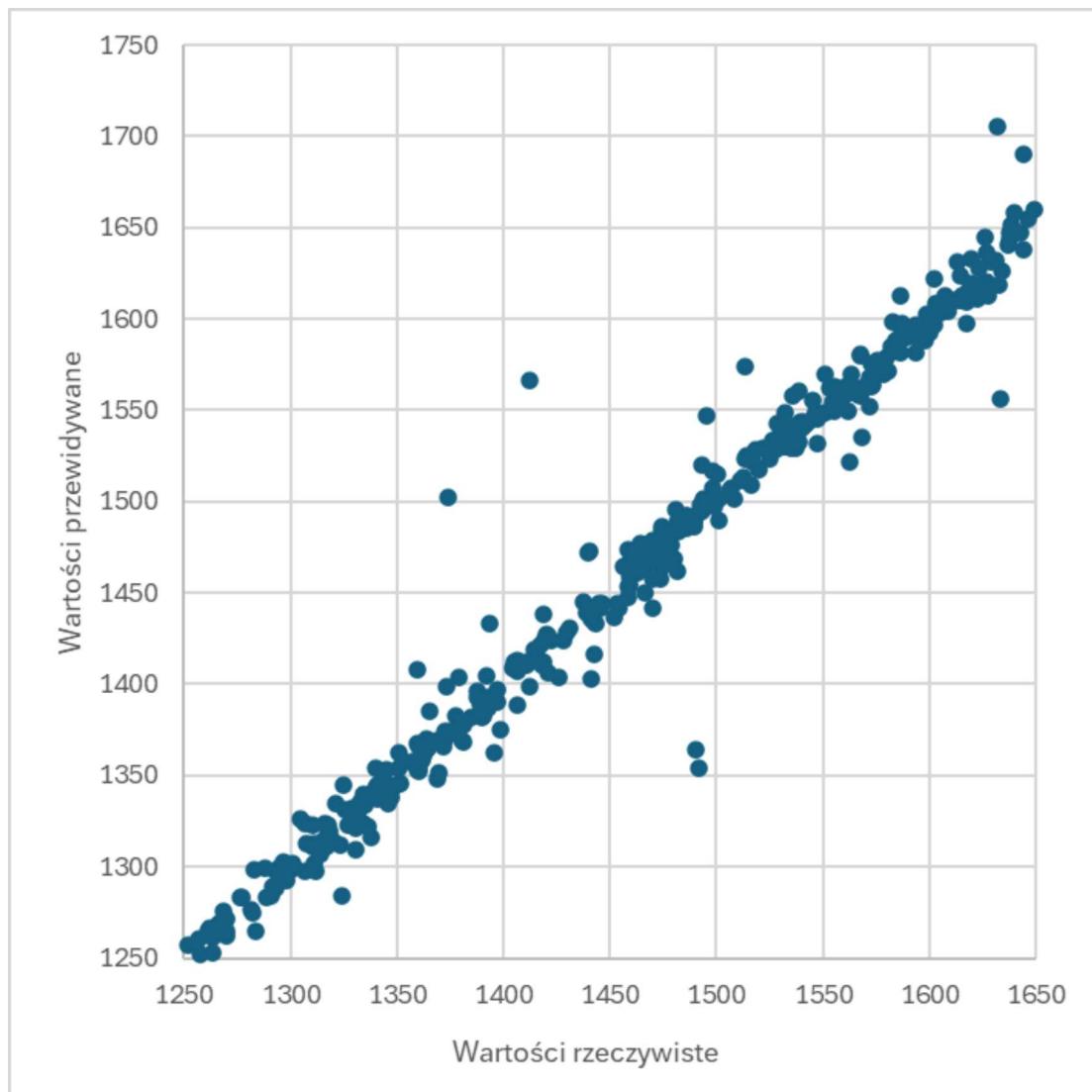
Rysunek 26 - Wyniki regresji dla różnych architektur sieci neuronowych w przypadku przewidywania wartości B i C na podstawie spłaszczonych próbek z przeskalowanymi wartościami



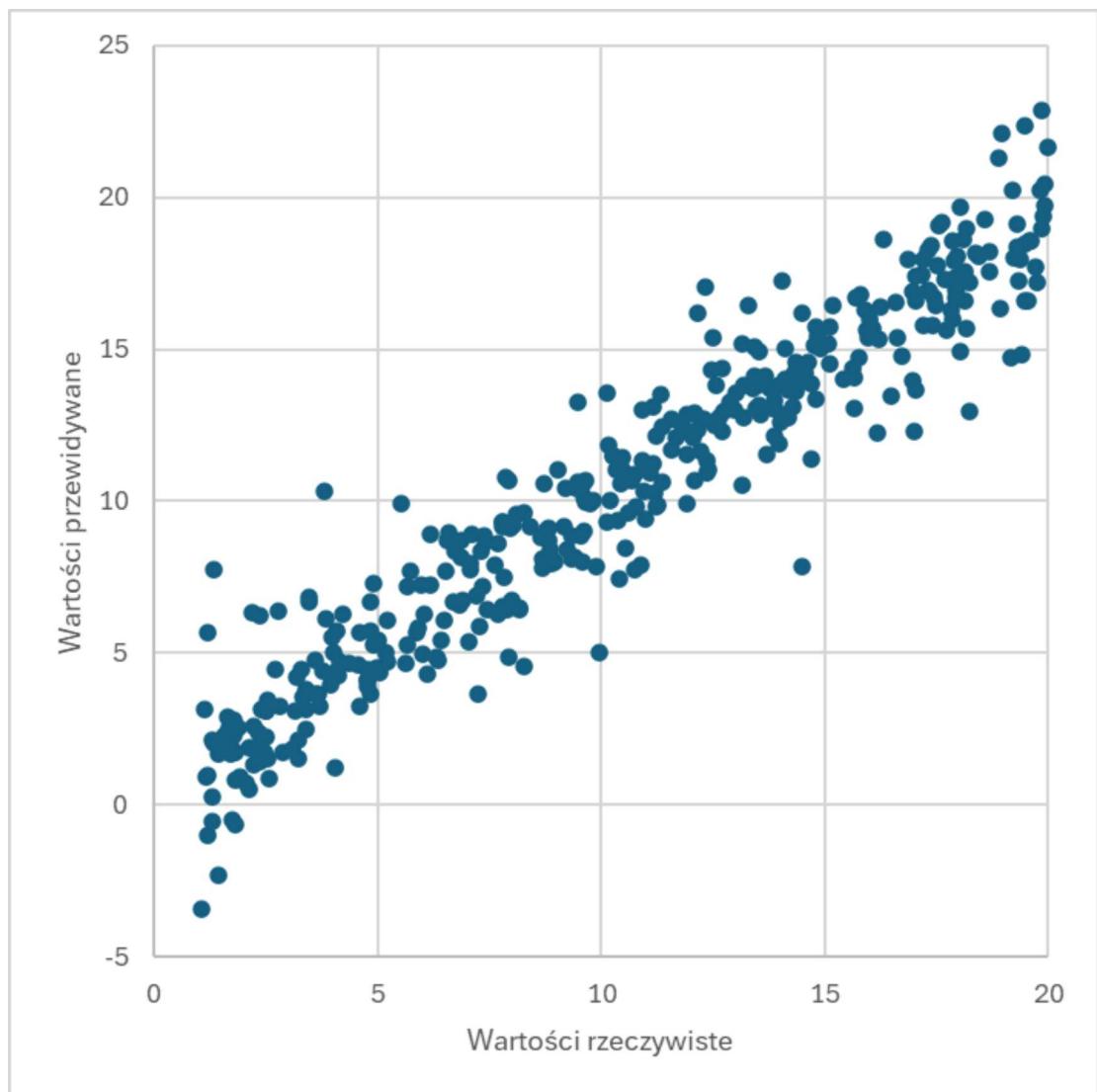
Rysunek 27 - Wyniki regresji dla różnych architektur sieci neuronowych w przypadku przewidywania wartości B i C na podstawie spłaszczonych próbek z przeskalowanymi wartościami

Dla parametru A wyszkolono tylko jedną sieć o topologii 128x64x32x16. Uzyskany wynik był lepszy niż dla wersji bez skalowania. Współczynnik determinacji wynosił 0,9884 dla

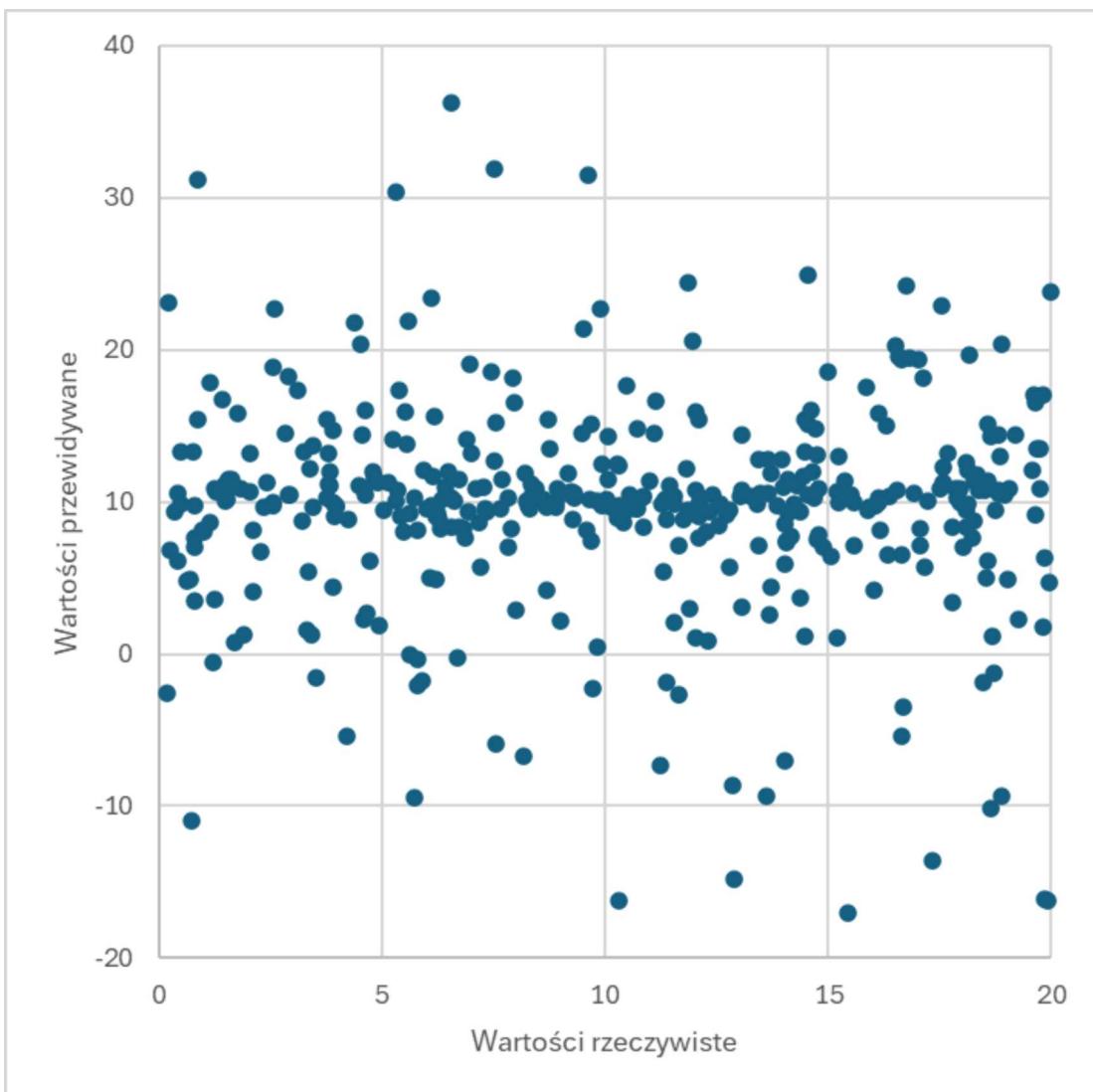
zbioru treningowego oraz 0,9726 dla zbioru testowego. Poniżej zamieszczono wykresy rozproszenia dla parametrów A, B i C.



Rysunek 28 - Wykres rozproszenia dla wartości A uzyskanych za pomocą sieci o architekturze $497 \times 128 \times 64 \times 32 \times 16 \times 1$ z przeskalowanymi wartościami.



Rysunek 29 - Wykres rozproszenia dla wartości B uzyskanych za pomocą sieci o architekturze $497 \times 96 \times 48 \times 24 \times 12 \times 1$ z przeskalowanymi wartościami.



Rysunek 30 - Wykres rozproszenia dla wartości C uzyskanych za pomocą sieci o architekturze $497 \times 128 \times 64 \times 32 \times 16 \times 1$ z przeskalowanymi wartościami.

Analiza wyników przedstawionych w tabeli oraz na wykresach wskazuje, że spośród rozważanych topologii sieci neuronowej najlepsze rezultaty dla etykiety B osiągnięto przy strukturze $96 \times 48 \times 24 \times 12$, dla której wartość współczynnika determinacji dla zbioru testowego wyniosła 0,9093, co świadczy o bardzo dobrej zdolności generalizacji modelu. Pozostałe konfiguracje również uzyskały wysokie wartości, jednak widoczny jest spadek dokładności na zbiorze testowym w przypadku topologii $128 \times 64 \times 32 \times 16$ oraz $48 \times 24 \times 12 \times 6$, co może wskazywać na częściowe przeuczenie modelu. W przypadku etykiety C nie zaobserwowano skutecznego dopasowania modelu dla żadnej z testowanych topologii, co sugeruje konieczność ponownej analizy jakości danych wejściowych, wyboru cech lub przemyślenia struktury modelu dla tej klasy wyjściowej. Problem jest na tyle widoczny że może on wskazywać na problem z samą generacją danych lub ich jakością, co może wymagać dalszych badań i ewentualnej modyfikacji procesu generowania danych. W szczególności istotna wydaje się być zależność pomiędzy danymi wprowadzanymi w poszczególne pola w programie VWASE a danymi umieszczanymi w nazwach plików zawierających dane na temat poszczególnych próbek.

Podsumowanie

W celu zbadania możliwości wykorzystania uczenia maszynowego do interpretacji danych elipsometrycznych wykorzystano podejście obiektowe. Utworzono klasy odpowiedzialne za reprezentację odpowiednich modeli, zbiorów danych oraz poszczególnych próbek. Pozwalają one na badanie możliwości opisania zestawu danych zarówno za pomocą regresji liniowej i jej wariacji, jak i za pomocą modeli opartych na sieciach neuronowych o różnych architekturach. Samo szkolenie przyniosło obiecujące wyniki w przypadku przewidywania wartości grubości oraz parametrów A i B, co zostało zademonstrowane poprzez obliczenie współczynników determinacji dla zbiorów treningowych i testowych. Dla wartości T, A i B współczynnik ten znajdował się w najlepszych przypadkach w zakresie powyżej 0,9, a wielokrotne próby szkolenia pokazały, że istnieje możliwość uzyskania wyższych jego wartości, jeśli zostaną zaimplementowane algorytmy wczesnego zatrzymywania procesu uczenia, bądź zostaną wprowadzone odpowiednie modyfikacje do architektury sieci neuronowej. Obiecujące rezultaty przyniosło także skalowanie etykiet do postaci, w której nie przyjmują one wartości większe niż 10^{-3} . Może to sugerować, że przekształcenie ich w inny odwracalny sposób – np. poprzez zlogarytmowanie ich – mogłoby przynieść jeszcze lepsze wyniki. Podczas przeprowadzania prac dostrzeżono także elementy które można by poprawić. Jednym z nich był fakt, że do tej pory stan wyuczonego modelu zapisywany jest poprzez zastosowanie funkcji `torch.save(model.state_dict(), save_path)`. Funkcja ta zapisuje obecny stan modelu oraz jego wag. Alternatywą dla niej jest zastosowanie funkcji `torch.save(model, save_path)`. Wersja ta oprócz obecnego stanu modelu oraz wag zapisuje także budowę modelu. Zmodyfikowanie kodu tak aby korzystał z drugiej wersji pozwoliłoby na zrezygnowanie z rozbudowanej konwencji nazewnicy której była stosowana podczas realizacji tej pracy. Brak sukcesu w stworzeniu modelu, który z sukcesem przewidywałby wartości parametru C, oraz sposób, w który przebiega uczenie modeli, w których parametr ten jest etykietą może wskazywać na błąd występujący w procesie generacji danych bądź w konstrukcji modeli. Być może lepsze rezultaty dla wartości C przyniosłoby poddanie ich odpowiedniemu skalowania bądź zastosowanie modelu większego niż te które stosowane były do tej pory.

Bibliografia

- [1] C. V. Grazia Giuseppina Politano, „Spectroscopic Ellipsometry: Advancements, Applications and Future Prospects in Optical Characterization,” *spectroscopy journal*, nr 1, p. 163–181, 2023.
- [2] H. Fujiwara, Spectroscopic Ellipsometry: Principles and Applications, John Wiley & Sons, Ltd, 2007.
- [3] J. N. H. HARLAND G. TOMPKINS, SPECTROSCOPIC ELLIPSOMETRY - Practical Application to Thin Film Characterization, NEW YORK: MOMENTUM PRESS, 2016.
- [4] D. Likhachev, „A practitioner's approach to evaluation strategy for ellipsometric measurements of multilayered and multiparametric thin-film structures,” *Thin Solid Films*, nr 595, pp. 113 - 117, 2015.
- [5] „Cauchy and related Empirical Dispersion Formulae - Technical Note,” [Online]. Available:
https://www.horiba.com/fileadmin/uploads/Scientific/Downloads/OpticalSchool_CN/TN/ellipsometer/Cauchy_and_related_empirical_dispersion_Formulae_for_Transparent_Materials.pdf. [Data uzyskania dostępu: 16 05 2025].
- [6] R. W. C. Hiroyuki Fujiwara, „Effect of Roughness on Ellipsometry Analysis,” w *Spectroscopic Ellipsometry for Photovoltaics - Volume 1: Fundamental Principles and Solar Cell Characterization*, Springer, 2018.
- [7] „Ellipsometry FAQ,” J. A. Wollam, 13 05 2025. [Online]. Available:
<https://www.jawoollam.com/resources/ellipsometry-faq>.
- [8] B. F. Z. H. Z. G. J. Budai, „On determining the optical properties and layer structure from spectroscopic ellipsometric data using automated artifact minimization method,” *Thin Solid Films*, nr 567, pp. 14-19, 2014.
- [9] U. Rossow, „A Brief History of Ellipsometry,” *Phys. Status Solidi*, 2019.

- [10] J. A. A. a. E. Rosenfeld, Talking Nets - An Oral History of Neural Networks, The MIT Press, 2000.
- [11] A. Géron, Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow, Sebastopol: O'Reilly Media, Inc, 2019.
- [12] A. A. L. N. M. B. M. C. Marco Ramilli, „Silicon Photomultipliers,” w *Photodetectors*.

Spis równań

Spis rysunków