Politechnika Warszawska

WYDZIAŁ MATEMATYKI
I NAUK INFORMACYJNYCH

Praca dyplomowa licencjacka

na kierunku Matematyka i Analiza Danych

Zastosowania sieci neuronowych do rozwiązywania równań różniczkowych zwyczajnych

Jan Pogłód

320575

promotor
dr inż. Łukasz Błaszczyk

Warszawa 2024

Streszczenie

Zastosowania sieci neuronowych do rozwiązywania równań różniczkowych zwyczajnych

Praca licencjacka bada porównanie tradycyjnych metod rozwiązywania równań różniczkowych zwyczajnych (RRZ) z zastosowaniem sieci neuronowych jako uniwersalnych aproksymatorów. Sieci neuronowe oferują nowoczesne podejście, wykorzystując teoretyczną zdolność do aproksymacji dowolnej funkcji ciągłej przy odpowiedniej liczbie neuronów. Niniejsza praca bada potencjał sieci neuronowych w modelowaniu rozwiązań RRZ, podkreślając ich przewagę nad tradycyjnymi metodami numerycznymi. Badanie składa się z trzech głównych części: przeglądu wybranych klas RRZ i metod numerycznych, omówienia twierdzenia o Uniwersalnej Aproksymacji oraz analizy architektur sieci neuronowych do efektywnej aproksymacji rozwiązań. Wizualizacje i analizy porównawcze podkreślają możliwości sieci neuronowych jako potężnego narzędzia w rozwiązywaniu równań różniczkowych zwyczajnych.

Słowa kluczowe: równania różniczkowe zwyczajne, sztuczna sieć neuronowa, model perceptronu wielowarstwowego, metody numeryczne, Teoria Uniwersalnej Aproksymacji, funkcja kosztu

Abstract

Applying artificial neural networks for solving ordinary differential equations

This thesis explores the comparison between traditional methods for solving ordinary differential equations (ODEs) and the application of neural networks as universal approximators. Neural networks offer a modern approach, leveraging their theoretical capacity to approximate any continuous function given sufficient neurons. This work investigates neural networks' potential to model ODE solutions, highlighting their advantages over traditional numerical methods. The study is divided into three main sections: an overview of ODEs and numerical methods, a discussion on the Universal Approximation Theorem, and an exploration of neural network architectures for efficient solution approximation. Visualizations and comparative analyses underscore the neural networks' capabilities and potential as a powerful tool in solving ODEs.

Keywords: ordinary differential equations, artificial neural network, multilayer perceptron model, numerical methods, Universal Approximation Theory, loss function

Spis treści

Wstęp

Praca dyplomowa poświęcona jest porównaniu tradycyjnych metod rozwiązywania równań różniczkowych zwyczajnych i zastosowaniu sieci neuronowych jako uniwersalnych aproksymatorów w tej dziedzinie. W kontekście ponad 300-letniej historii teorii równań różniczkowych zwyczajnych [8], badanie to oparliśmy o poszukiwanie skutecznych i nowocześniejszych od tradycyjnych metod numerycznych sposobów aproksymacji rozwiązań. Coraz trudniejsze problemy modelowania matematycznego wymuszają potrzebę rozwoju i opracowania bardziej zaawansowanych technik. Sieci neuronowe, jako potężne narzędzie uczenia maszynowego, zapewniają nowe perspektywy w rozwiązaniu tych złożonych problemów matematycznych [10]. W ramach tej pracy, skoncentrowaliśmy się również na uzupełnieniu szczegółów dowodu Teorii Uniwersalnej Aproksymacji, która twierdzi, że sieci neuronowe o odpowiedniej liczbie neuronów w jednej warstwie ukrytej są zdolne do przybliżania każdej funkcji ciągłej z dowolną precyzją [3]. To teoretyczne podejście stanowi grunt do zbadania zdolności sieci neuronowych w modelowaniu rozwiązań równań różniczkowych.

Równania różniczkowe zwyczajne są nieodłączną częścią fizyki [9], matematyki [10, 11] i wielu innych nauk ścisłych. Stanowią one podstawę wielu praw opisujących oddziaływania w naszym świecie, potrafią analizować dynamikę schematów mechanicznych i elektrycznych [2] lub badać stabilność układów. Ponadto są również podwaliną obecnej nauki o biologi [2] i medycynie, służą do modelowania ruchu w populacji czy rozprzestrzeniania się chorób zakaźnych [7]. Bez odkrycia tych równań nie umielibyśmy opisywać praw istniejących w naszym świecie. Matematycy, fizycy i inżynierowie od setek lat uczą się jak je tworzyć i rozwiązywać, aby dostać odpowiedź na postawione zagadnienia. Jednakże, jak pokażemy, istnieją równania trudne lub niemożliwe do rozwiązania za pomocą dostępnych twierdzeń, metod i schematów analitycznych. W celu ich rozwiązywania od XVIII wieku powstają metody numeryczne [5], które potrafią aproksymować rozwiązania równań różniczkowych, również tych, których nie da się obliczyć analitycznie. Prawdziwy przełom w rozwoju metod numerycznych nastąpił dopiero w drugiej połowie XX wieku wraz z erą cyfrową oraz rozwojem komputerów. Era ta dała nowe perspektywy dla tworzenia bardziej skomplikowanych, efektywniejszych metod numerycznych, co z kolei przyczyniło się do szybkiego postępu w wielu dziedzinach nauki i techniki. W erze cyfrowej człowiek dysponując

potęgą obliczeniową komputerów jest w stanie numerycznie aproksymować rozwiązania. Często jednak korzystanie z narzędzi komputerowych wiąże się z utratą dokładności przybliżanej funkcji, błędami ekstrapolacji i długim czasem obliczania.

W naszej pracy skupimy się na sposobie implementacji efektywnych architektur sztucznych sieci neuronowych zdolnych apkroksymować rozwiązania wybranych klas równań różniczkowych zwyczajnych. Wyniki przedstawione w tej pracy porównują tradycyjne metody numeryczne wraz z możliwościami sieci neuronowych co poszerza zrozumienie ich teoretycznych możliwości jako uniwersalnych aproksymatorów. Praca podzielona jest na trzy główne części. Pierwsza część omawia pojęcia z dziedziny równań różniczkowych zwyczajnych oraz przykłady wykorzystujące metody numeryczne jako narzędzia do ich rozwiązywania. Druga część poświęcona jest dowodowi twierdzenia o Teorii Uniwersalnej Aproksymacji podczas gdy trzecia część odnosi się do pojęć z teorii sztucznych sieci neuronowych i kroków implementacji modelu skutecznego perceptronu wielowarstwowego. W części tej rozważaliśmy również różne metody polepszające wyniki aproksymacji i czas szukania rozwiązań oraz porównywaliśmy między sobą odmienne architektury i parametry sieci neuronowych. Następnie przedstawiliśmy wizualizacje i osiągnięcia zaimplementowanych narzędzi, aby zbadać potencjał sieci neuronowych nad metodami numerycznymi oraz znaleźć sieć neuronową, która najefektywniej realizuje proces uczenia.

1. Wybrane klasy równań różniczkowych zwyczajnych

Wyróżniamy wiele klas równań różniczkowych zwyczajnych. Równaniem różniczkowym rzędu $n \geqslant 1$ nazwiemy wyrażenie postaci:

$$F\left(\frac{d^n y}{dt^n}, \frac{d^{n-1} y}{dt^{n-1}}, \ldots, \frac{dy}{dt}, y, t\right) = 0, \tag{1.1}$$

gdzie $F\colon \mathbb{R}^{(n+1)\cdot m+1} \to \mathbb{R}$ jest zadaną funkcją, a jego rozwiązaniem szczególnym jest odpowiednio gładka funkcja $y\colon \mathbb{R} \supset I \to \mathbb{R}^m$, która po podstawieniu spełnia powyższe równanie dla każdego $t \in I$ [11].

W przypadku gdy $n = 1$ powiemy, że równanie różniczkowe jest rzędu pierwszego, a przykładem takiego równania jest model w termodynamice opisujący proces ochładzania substacji, gdzie $T(t)$ opisuje zmianę jej temperatury w funkcji czasu t. Proces ten opisujemy wzorem (1.2) i metodą rozdzielenia zmiennych i całkowania dostajemy jawny wzór na jego analityczne rozwiązanie jak w (1.3):

$$\frac{dT}{dt} = -k(T - T_{\text{otoczenia}}), \tag{1.2}$$

$$T(t) = T_{\text{otoczenia}} + (T_0 - T_{\text{otoczenia}}) \cdot e^{-kt}, \tag{1.3}$$

gdzie k jest stałą ochładzania zależną od wyboru substacji zaś $T_{\text{otoczenia}}$ oraz T_0 opisują stałą temperaturę, w której znajduje się ciało i temperaturę początkową w chwili $t = 0$.

Równania, dla których $n = 2$ nazywamy równaniami drugiego rzędu i jest nim na przykład równanie opisujące ruch oscylatora harmonicznego które mówi, że położenie ciała w funkcji czasu x jest w następujący sposób zależne od częstości kołowej w oscylacji:

$$\frac{d^2 x}{dt^2} + \omega^2 x = 0.$$

Podstawiając pod x formułę $x(t) = A\cos(\omega t) + B\sin(\omega t)$ dostajemy rozwiązanie ogólne równania ruchu oscylatora harmonicznego spełnione dla każdego t. W tym przykładzie stałe A, B można wyznaczyć poprzez odpowiednie, zadane warunki początkowe.

Równania różniczkowe zwyczajne o rzędzie $n > 2$ to równania, które zawierają pochodne rzędu większego niż dwa. Oznacza to, że w takich równaniach występują pochodne trzeciego, czwartego i wyższych rzędów.

Szczególnym przypadkiem równań różniczkowych są równania macierzowe postaci $x' = Ax$, które potrafią na przykład opisywać procesy dynamiczne. Jednym z takich procesów może być zmiana dynamiki dwóch gatunków x_1 i x_2, podczas gdy macierz A reprezentuje interakcje między tymi gatunkami, na przykład:

$$\begin{bmatrix} x_1' \\ x_2' \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}.$$

Wówczas, po znalezieniu wartości własnych macierzy A, w powyższym przykładzie będą to $\lambda_1 = \frac{5+\sqrt{33}}{2}$ i $\lambda_2 = \frac{5-\sqrt{33}}{2}$, dostaniemy rozwiązanie ogólne powyższego równania. Jest nim kombinacja liniowa rozwiązań własnych tzn:

$$\begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = c_1 e^{\lambda_1 t} \begin{bmatrix} \frac{6}{-3-\sqrt{33}} \\ 1 \end{bmatrix} + c_2 e^{\lambda_2 t} \begin{bmatrix} \frac{6}{-3+\sqrt{33}} \\ 1 \end{bmatrix},$$

gdzie c_1 i c_2 są stałymi określonymi przez warunki początkowe.

Innym zjawiskiem, które możemy modelować w analogiczny sposób może być na przykład zależność położenia punktu (x_1, x_2) w przestrzeni $\mathbb{R}^2$ w zależności od czasu. Rozpatrzmy układ, w którym położenie takiegu punktu zależy od składowych w następujący sposób:

$$\begin{cases} \frac{dx_1}{dt} = x_2, \\ \frac{dx_2}{dt} = -x_1, \end{cases} \qquad \begin{bmatrix} x_1' \\ x_2' \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}.$$

Wówczas rozwiązaniem, a więc wzorem na położenie konkretnej składowej w przestrzeni w zależności od czasu to dla x_1: $x_1(t) = A\cos(t) + B\sin(t)$ oraz dla $x_2 : x_2(t) = -A\sin(t) + B\cos(t)$, gdzie ponownie stałe A i B są wyznaczane za pomocą warunków początkowych.

Możemy zauważyć, że warunki początkowe odgrywają fundamentalną rolę w rozwiązywaniu równań różniczkowych. Za ich pomocą zamiast nieskończonej liczby funkcji dobranych na nieskończoną ilość stałej A potrafimy otrzymać dobrze zdefiniowaną funkcję. Są one także warunkiem startowym, początkiem w którym rozwiązanie równania nabiera sensu i jest podstawą do działania tradycyjnych metod numerycznych. Oczywiście zazwyczaj znamy warunki początkowe i dla nich chcemy otrzymać jednoznaczne rozwiązanie. Może to być początkowa temperatura $T_0 = 60°\mathrm{C}$ lub początkowe położenie punktu $(x_1, x_2)_{|t=0} = (0, 0)$. Warunek początkowy jest niezbędny, aby uzyskać jednoznaczną traktorię rozwiązania. O jednoznaczności rozwiązania mówi nam twierdzenie Picarda-Lindelöfa [10, 11].

Twierdzenie 1.1 (Picarda-Lindelöfa). Rozważmy równanie postaci $\frac{dy}{dt} = f(y, t)$, gdzie $f : H \subset \mathbb{R}^{m+1} \to \mathbb{R}^m, y_0 \in \mathbb{R}^m, t_0 \in \mathbb{R}, a, b > 0$, a także:

$$H = B(y_0, b) \times [t_0 - a, t_0 + a].$$

Przypuśćmy, że f jest ciągła na H i lipschitzowsko ciągła względem zmiennej y ze stałą L, wtedy:

$$\forall_{t\in[t_0-a,t_0+a]}\forall_{y_1,y_2\in B(y_0,b)} \quad ||f(y_1,t)-f(y_2,t)|| \leqslant L||y_1-y_2||.$$

Niech $M = \sup\left(||f(y,t)|| : (y,t) \in H\right)$, wybierzmy $\alpha > 0$, takie że $\alpha < \min\left(\frac{b}{M}, \frac{1}{L}, a\right)$.

Wówczas istnieje dokładnie jedno rozwiązanie y określone na $[t_0-\alpha, t_0+\alpha]$ i spełniające warunek początkowy $y(t_0) = y_0$.

W powyższym twierdzeniu przytoczona norma $||\cdot||$ to klasyczna norma euklidesowa. Warunki początkowe są również istotne w analizie równań różniczkowych zwyczajnych ze względu na zapewnienie stabilności i przewidywalności rozwiązań. Gwarantują to twierdzenie o ciągłej zależności rozwiązania od warunków początkowych oraz twierdzenie o różniczkowalnej zależności od warunków początkowych. Pierwsze z nich mówi, że niewielkie zmiany w tych warunkach nie prowadzą do dużych zmian w rozwiązaniach, podczas gdy drugie pozwala na dokładne obliczenie, jak te zmiany wpływają na równanie i tym samym całe zjawisko [10]. Dzięki temu możemy bardziej precyzyjnie modelować i analizować różnorodne zjawiska.

Pokażemy, że dany warunek początkowy oprócz zastosowania analitycznego jest także potrzebny w implementacji sieci neuronowej – określać on będzie w jej kontekście przesunięcie wektora wyjścia, a więc rozwiązania równania. Będziemy również badać wpływ różnych warunków początkowych na proces uczenia sieci neuronowej oraz jej zbieżność do poprawnego rozwiązania. W następnym kroku przedstawimy implementacje dwóch wybranych metod numerycznych, które wraz z rozwiązaniem analitycznym będą nam służyć jako punkt odniesienia dokładności sztucznej sieci neuronowej. Będziemy rozpatrywać metodę Eulera oraz metodę Rungego-Kutty czwartego rzędu.

2. Wybrane metody numeryczne

Problemami jakie stają na drodze matematykom w szukaniu rozwiązania szczególnego równań różniczkowych są trudności całkowania, konieczność znajomości głębokich zagadnień matematycznych oraz co najistotniejsze, brak możliwości rozwiązania lub istnienie całek nieelementarnych, których, jak udowodniono, w żaden sposób nie da się obliczyć analitycznie [10]. Przykładem równań, dla których trudno jest znaleźć jawne rozwiązanie analityczne są modele nieliniowe, jak na przykład model wzrostu populacji organizmów wyrażony wzorem:

$$\frac{dP}{dt} = r \cdot P \cdot \left(1 - \frac{P}{K}\right).$$

We wzorze tym $P(t)$ reprezentuje rozmiar populacji w danej chwili czasu t, a r i K to odpowiednio współczynnik przyrostu i maksymalna liczba organizmów, jakie środowisko jest w stanie utrzymać.

Poza równaniami nieliniowymi istnieją również równania różniczkowe, których nie możemy rozwiązać przez całki elementarne. Podamy przykład takiego równania, w którym całkując je obustronnie otrzymujemy wyrażenie, gdzie po prawej stronie stoi całka $\int \frac{e^x}{x}\,dx$, której nie możemy obliczyć analitycznie, ponieważ nie da jej się przedstawić w terminach funkcji elementarnych:

$$\frac{dx}{dt} = e^{-x}x,$$
$$\int dt = \int \frac{e^x}{x}\,dx.$$

Wówczas, w celu aproksymacji rozwiązań takich równań możemy stosować metody komputerowe. Pierwszą metodą numeryczną, z której będziemy korzystać w naszej pracy ze względu na jej prostotę i efektywność, jest metoda Eulera. Działa ona poprzez przybliżanie wartości funkcji w zadanym przedziale na podstawie części wartości z jego zakresu – kroków czasowych t_1,...,t_n.

2.1. Metoda Eulera

Załóżmy, że mamy równanie postaci $y' = f(t, y)$ oraz warunki początkowe (t_0, y_0) wraz z kolejnymi punktami na osi (z krokiem jej podziału h):

$$t_{i+1} = t_i + h, \qquad i = 0, ..., n-1.$$

Z definicji pochodnej $y'(t_i) = \frac{\Delta y}{h} + o(h)$, gdzie $\Delta y = y_{i+1} - y_i$, a z równania różniczkowego $y'(t_i) = f(t_i, y_i)$, więc:

$$y_{i+1} = y(t_{i+1}) = y(t_i + h) \approx y_i + hf(t_i, y_i) + \frac{f^{(2)}(\Psi)}{2}h^2,$$

gdzie $\Psi \in (t_i, t_{i+1})$ oraz skorzystaliśmy z rozwinięcia funkcji $y(t_i + h)$ ze wzoru Taylora. Przy eksperymentach konieczne będzie, wzięcie pod uwagę, że aproksymacja wartości $y(t_{i+1})$ ma błąd rzędu h^2, co daje, że mniejszy krok podziału daje lepszy wynik aproksymacji. Ze względu na swoją prostotę, metoda Eulera ma niskie wymagania obliczeniowe. Jest szybka i efektywna dla małych problemów, lecz aby uzyskać dokładne wyniki, konieczne jest użycie dużej liczby kroków czasowych, co może znacząco zwiększyć czas obliczeń. W równaniu: $y'(t_i) = \frac{\Delta y}{h} + o(h)$ wyrażenie $o(h)$ oznacza, że dla $h \to 0$, wyraz ten zanika szybciej niż h, czyli: $o(h) = \lim_{h\to 0} \frac{o(h)}{h} = 0$.

2.2. Metoda Rungego-Kutty czwartego rzędu

Drugą z rozpatrywanych metod numerycznych będzie bardziej zaawansowana metoda Rungego-Kutty czwartego rzędu [5]. Rząd czwarty tej metody oznacza, że jej błąd globalny maleje w tempie $O(h^4)$ wraz ze zmniejszeniem kroku podziału h tzn. jeśli $E(h)$ oznacza błąd globalny tej metody to wyrażenie $O(h^4)$ oznacza, że istnieje stała C taka, że: $|E(h)| \leqslant C \cdot h^4$ dla wystarczająco małego h. Jest to również metoda czterostopniowa co oznacza, że używamy czterech oszacowań pochodnych w każdym kroku czasowym do przybliżenia wartości funkcji. Takie podejście wymaga większej mocy obliczeniowej w szczególności dla bardziej skomplikowanych funkcji, jednakże algorytm ten potrzebuje mniejszej liczby podziałów odcinka do uzyskania dobrych apkrosymacji, co w eksperymentach może stawiać go jako dobrego konkurenta sieci neuronowej. Algorytm Rungego-Kutty zaczynamy od tych samych założeń co w metodzie Eulera, rozpoczynając go od wyrażenia $\Delta y = hf(t_i, y_i)$ i wprowadzając następujące oszacowania:

$$o_1 = hf(t_i, y_i), \qquad o_2 = h \cdot f\left(t_i + \frac{h}{2}, y_i + \frac{o_1}{2}\right),$$

$$o_3 = h \cdot f\left(t_i + \frac{h}{2}, y_i + \frac{o_2}{2}\right), \qquad o_4 = h \cdot f\left(t_i + h, y_i + o_3\right).$$

W kolejnym kroku aproksymujemy y_{i+1} poprzez wyrażenie ψ, gdzie

$$\psi := \frac{1}{6}(o_1 + 2o_2 + 2o_3 + o_4).$$

W rezultacie dla obu metod (E) Eulera i (R) Rungego-Kutty dostajemy następujące oszacowania, z których dostajemy, że błąd lokalny w metodzie Eulera maleje w tempie $O(h^2)$, podczas gdy błąd lokalny w metodzie Rungego-Kutty w tempie $O(h^5)$, co stawia ją jako dokładniejszą przy zmniejszaniu kroku podziału h:

$$(E) \quad y_{i+1} \approx y_i + hf(t_i, y_i) + O(h^2),$$

$$(R) \quad y_{i+1} \approx y_i + \psi + O(h^5).$$

2.3. Praktyczne zastosowania metod numerycznych

W naszych dalszych testach i porównaniach będziemy rozważać obie z wprowadzonych metod. Widzimy, że metody numeryczne oraz używanie mocy obliczeniowej komputera jest bardzo przydatnym narzędziem do rozwiązywania równań – im mniejsze podziały odcinków zastosujemy tym bliżej będziemy dokładnego oszacowania punktu y_i. Pokażemy, że metody aproksymowania rozwiązań są niezbędnym narzędziem, gdy nie możemy przedstawić ich analitycznie. W tym celu skorzystamy z twierdzenia Picarda-Lindelöfa. Rozważmy równanie różniczkowe postaci:

$$y' = \cos(y - x), \qquad y(0) = 1.$$

Dalej po rozpisaniu go dostajemy równanie, w którym nie możemy rozdzielić zmiennych x i y, tzn.

$$\frac{dy}{dx} = \cos(x)\cos(y) + \sin(x)\sin(y).$$

Udowodnimy jednak, że rozwiązanie tego równania istnieje na podstawie przytoczonego twierdzenia o jednoznaczności rozwiązań równań różniczkowych. Oczywiście funkcja $f(x, y) = \cos(y - x)$ jest funkcją ciągłą na całej swojej dziedzinie jako funkcja elementarna. Korzystając z twierdzenia Lagrange'a pokażemy, że jest również lipschitsowsko ciągła względem y z $L \leqslant 1$:

$$\begin{aligned} |f(x, y_1) - f(x, y_2)| &= |\cos(y_1 - x) - \cos(y_2 - x)| \\ &= |f'(x, c)||y_1 - y_2| = |\sin(c - x)||y_1 - y_2|, \end{aligned}$$

gdzie c to stała znajdująca się pomiędzy wartościami y_1 i y_2.

Otrzymaliśmy sprowadzenie do funkcji $|\sin(c-x)|$, a z kolei wiemy, że $\forall_{x\in\mathbb{R}} |\sin(c-x)| \leqslant L = 1$ zatem wyrażenie stojące przy $|y_1 - y_2|$ jest skończone, ponadto niewiększe od 1 i założenia twierdzenia Picarda-Lindelöfa są spełnione. Istnieje więc dokładnie jedno rozwiązanie powyższego

równania różniczkowego. Jednakże, choć powyższe równanie da się rozwiązać analitycznie to istnieją również takie typy równań, których nie możemy przedstawić w postaci funkcji elementarnych. Rozpatrzmy:

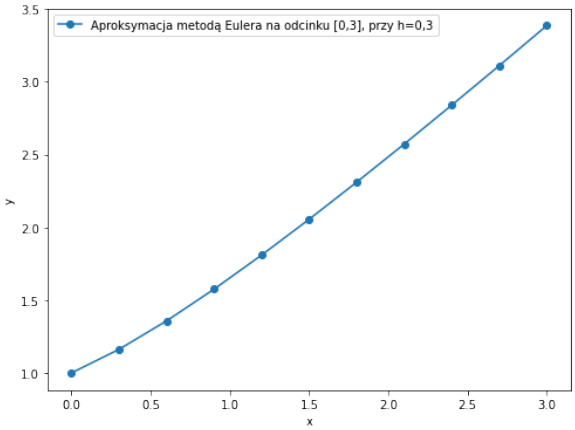
$$y' = \sin(x) - \frac{1}{10}y^3, \qquad y(0) = 0.$$

Podobnie funkcja $f(x, y) = \sin(x) - \frac{1}{10}y^3$ jest funkcją ciągłą na całej swojej dziedzinie, więc sprawdzimy teraz warunek Lipschitza na ograniczonym przedziale (a, b):

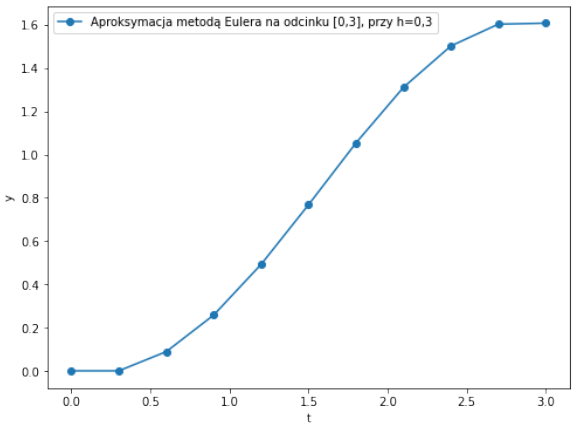
$$\begin{aligned} |f(x, y_1) - f(x, y_2)| &= |(\sin(x) - \frac{1}{10}y_1^3) - (\sin(x) - \frac{1}{10}y_2^3)| = \frac{1}{10}|y_2^3 - y_1^3| \\ &= \frac{1}{10}|y_2 - y_1||y_1^2 + 2y_1y_2 + y_2^2| \leqslant \frac{1}{10}3M^2|y_1 - y_2|, \end{aligned}$$

gdzie w ostatnim przejściu skorzystaliśmy z faktu, że na ograniczonym przedziale możemy znaleźć skończoną stałą M taką, że $|y| \leqslant M$, a stąd $|y_1^2 + 2y_1y_2 + y_2^2| \leqslant 4M^2$. Ponownie z twierdzenia Picarda-Lindelöfa dostajemy, że rozwiązanie tego równania istnieje nawet gdy wiemy, że nie da się go przedstawić funkcjami analitycznymi.

Powyższe równania, mimo że są trudne bądź niemożliwe do rozwiązania analitycznie możemy rozwiązać na przykład na przedziale $[0, 3]$ używając implementacji metody Eulera dzieląc ten przedział na 10 części. Wówczas na rysunku 2.1 możemy zaobserwować następujące wyniki.



(a) Aproksymacja rozwiązania równania $y' = \cos(y - x)$, $y(0) = 1$ metodą Eulera



(b) Aproksymacja rozwiązania równania $y' = \sin(t) - \frac{1}{10}y^3$, $y(0) = 0$ metodą Eulera

Rysunek 2.1: Aproksymacje wybranych równań metodą Eulera

Zauważmy, że w obu przypadkach otrzymane rozwiązanie jest jedynie aproksymacją – ciągiem punktów połączonych łamaną, niegładką funkcją, oraz bez możliwości dokładnej ekstrapolacji wartości spoza danego zakresu. Choć powyższe metody numeryczne dają dobre aproksymacje (błędy lokalne wynoszą $O(h^2)$ i $O(h^5)$) i sprawdzają się w wielu problemach, naukowcy szukają nowych technik na miarę XXI wieku, które rozwiną granice metod numerycznych. Najistotniejszym kluczem do tego aby nowo odkryta metoda okazała się użyteczna jest jej efektywność,

dokładność oraz czas działania. W obecnych czasach stawiane są przed nami coraz trudniejsze zagadnienia modelowania matematycznego, które wymagają efektywnych metod. Jednym z nowoczesnych podejść do aproksymacji rozwiązań równań różniczkowych jest użycie sztucznych sieci neuronowych. Skontruowanie takiej sieci wymaga odpowiedniego zbadania wielu czynników takich jak metody optymalizowania, architektury sieci, dawkowania danych czy wyboru funkcji aktywacji i funkcji próby. Z powodu iż sieci neuronowe odniosły duży sukces w wielu dziedzinach nauki [4], mogą być one kolejną rewolucją w aproksymowaniu rozwiązań równań różniczkowych. Ich duże osiągnięcia wynikają w dużej mierze z Teorii Uniwersalnej Aproksymacji.

3. Teoria Uniwersalnej Aproksymacji

Teoria Uniwersalnej Aproksymacji mówi, że sieć neuronowa z jedną warstwą ukrytą o odpowiedniej liczbie neuronów może przybliżyć dowolną funkcję ciągłą na ograniczonym przedziale z dowolną dokładnością [3]. Model sieci neuronowej definiujemy w tym twierdzeniu jako funkcję $G(x)$, przy ustalonym wektorze wejść x. W dalszych sekcjach na podstawie definicji funkcji $G(x)$ implementować będziemy sztuczną sieć neuronową aproksymującą rozwiązania równań różniczkowych, gdzie funkcja sigmoidalna odgrywać będzie kluczową rolę. Wpierw zdefiniujemy kilka pojęć oraz twierdzeń, które przytaczamy za [3, 10, 11].

Definicja 3.1. Mówimy, że funkcja σ jest sigmoidalna, jeśli spełnia:

$$\begin{cases} \lim\limits_{t\to+\infty} \sigma(t) = 1, \\ \lim\limits_{t\to-\infty} \sigma(t) = 0. \end{cases}$$

Definicja 3.2. Mówimy, że całkowalna funkcja σ jest funkcją dyskryminacyjną jeśli dla pewnej domkniętej n–wymiarowej kostki $[0,1]^n$ oznaczonej jako I i pewnej miary $\mu \in B(I)$ całka:

$$\forall_{y\in\mathbb{R}^n,\theta\in\mathbb{R}} \quad \int_I \sigma(y^T x + \theta) d\mu(x) = 0$$

implikuje, że $\mu = 0$.

Stwierdzenie 3.3. Miara borelowska $\mu \in B(I)$ na skończonym przedziale I jest ograniczona.

Twierdzenie 3.4 (Hahna-Banacha). Niech V będzie unormowaną przestrzenią wektorową nad ciałem liczb rzeczywistych, a W podprzestrzenią liniową przestrzeni V. Jeśli $p : V \to \mathbb{R}$ jest funkcjonałem podliniowym na V, tj. takim że

$$\forall_{x\in V,\alpha\in\mathbb{R}} \quad p(\alpha x) = \alpha p(x), \qquad \forall_{x,y\in V} \quad p(x+y) \leqslant p(x) + p(y),$$

to dla każdego liniowego funkcjonału $f : W \to \mathbb{R}$ takiego, że $f(y) \leqslant p(y)$ dla każdego $y \in W$, istnieje liniowy funkcjonał $F : V \to \mathbb{R}$ taki, że $F(y) = f(y)$ dla każdego $y \in W$, a dodatkowo $F(x) \leqslant p(x)$ dla każdego $x \in V$.

Definicja 3.5. Mówimy, że przestrzeń V^* jest przestrzenią sprzężoną do przestrzeni V nad ciałem K, jeśli $V^* =: B(V, K)$, tj. jest przestrzenią wszystkich ciągłych funkcjonałów liniowych określonych na danej przestrzeni V.

Twierdzenie 3.6 (Riesza-Frecheta). Niech V będzie przestrzenią wektorową z iloczynem skalarnym. Dla każdego liniowego, ograniczonego funkcjonału F na V^*, istnieje dokładnie jedno y w V takie, że dla każdego x w V, $F(x) = \langle x, y\rangle$, gdzie $\langle\cdot,\cdot\rangle$ oznacza iloczyn skalarny w V.

Twierdzenie 3.7 (o Uniwersalnej Aproksymacji). Dla ciągłej funkcji sigmoidalnej i dyskryminacyjnej σ zbiór funkcji w postaci G (dla dowolnych N, n, $\alpha_j, w_{i,j}, b_j$)

$$G(x) = \sum_{j=1}^{N} \alpha_j \sigma \left(\sum_{i=1}^{n} x_i w_{i,j} + b_j \right), \tag{3.1}$$

gdzie $x = (x_1, x_2, ..., x_n)$ jest gęsty w przestrzeni wszystkich funkcji ciągłych $[0, 1]^n \to [0, 1]$.

Wniosek 3.8. Dla danej funkcji ciągłej $f : [0, 1] \to \mathbb{R}$, istnieje warstwa ukryta H z wystarczającą liczbą neuronów oraz odpowiednie wagi, takie że funkcja wyjściowa $G(x; \theta)$ sieci neuronowej może przybliżyć $f(x)$ z dowolną dokładnością, tzn:

$$\forall_{\epsilon>0}, \exists_{\theta\in\mathbb{R}} \forall_{x\in[0,1]} \quad |G(x; \theta) - f(x)| < \epsilon. \tag{3.2}$$

Dowód twierdzenia 3.7. (1) Niech $S \subset C([a, b])$ będzie zbiorem wszystkich funkcji postaci $G(x)$.

Pokażemy, że S jest podprzestrzenią liniową $C([a, b])$.

a) S zawiera taką funkcję, która dla każdego $x \in [a, b]$ jest zero. Istotnie jeżeli przyjmiemy $\alpha_1 = ... = \alpha_N = 0$ otrzymamy $\forall_{x\in[a,b]}\ G(x) = 0$.

b) Pokażemy, że S jest zamknięta na dodawanie. Niech G_1, G_2 należą do S. Wówczas

$$G_1(x) + G_2(x) = \sum_{j_1=1}^{N_1} (\alpha_{j_1}) \sigma \left(\sum_i x_i w_{i,j_1} + b_{j_1} \right) + \sum_{j_2=1}^{N_2} (\mu_{j_2}) \sigma \left(\sum_i x_i \theta_{i,j_2} + \beta_{j_2} \right).$$

Widzimy, że $G_1 + G_2$ nadal jest funkcją postaci G, ponieważ jest to suma funkcji σ przeskalowanych przez stałe, co odpowiada definicji funkcji G. Możemy to także zapisać jako:

$$G_1(x) + G_2(x) = \sum_{k=1}^{N_1+N_2} g_k \cdot \sigma(h_k \cdot x + i_k),$$

gdzie g_k to pewne nowe stałe, które wynikają z sumy α_{j_1} i μ_{j_2}, zaś h_k oraz i_k są nowymi parametrami pochodzącymi z w_{j_1}, b_{j_1}, θ_{j_2} i β_{j_2}.

c) Ostatecznie niech c będzie dowolnym skalarem. Zauważmy, że

$$cG(x) = \sum_{j=1}^{N} (c\alpha_j)\sigma\left(\sum_i x_i w_{i,j} + b_j\right),$$

co pokazuje, że S jest zamknięte na mnożenie przez skalar, zatem z **a)**,**b)**,**c)** istotnie S jest podprzestrzenią liniową $C([a,b])$.

(2) Teraz pokażemy, że domknięcie przestrzeni S jest całą przestrzenią $C([a,b])$. Załóżmy w tym celu, że domknięcie S nie jest całą przestrzenią $C([a,b])$. Wówczas domknięcie zbioru S, oznaczmy je jako S_d, jest domkniętą właściwą podprzestrzenią przestrzeni $C([a,b])$. W kolejnym kroku pokażemy, że z twierdzenia Hahna-Banacha istnieje funkcjonał L na $C([a,b])$, taki że $L \neq 0$ na $C([a,b])$ oraz spełnia on, że $L(w) = 0$ dla każdego $w \in S_d$.

Mamy, że $S_d < C([a,b])$. Wprowadźmy funkcjonał $p(f) = \sup_{a\leqslant x\leqslant b} |f(x)|$ na przestrzeni $C([a,b])$. Zauważmy, że spełnia on warunek podlinowości na $C([a,b])$.

Podaddytywność: Weźmy dwie funkcje $f, g \in C([a,b])$. Chcemy pokazać, że dla p zachodzi $p(f+g) \leqslant p(f) + p(g)$. Mamy:

$$p(f+g) = \sup_{a\leqslant x\leqslant b} |f(x)+g(x)| \leqslant \sup_{a\leqslant x\leqslant b} |f(x)| + \sup_{a\leqslant x\leqslant b} |g(x)| = p(f) + p(g).$$

Zatem $p(f+g) \leqslant p(f) + p(g)$, co pokazuje podaddytywność funkcjonału p.
Homogeniczność: Chcemy pokazać, że dla dowolnej funkcji $f \in C([a,b])$ i dowolnego skalaru α, zachodzi $p(\alpha f) = |\alpha| \cdot p(f)$.

$$p(f) = \sup_{a\leqslant x\leqslant b} |\alpha f(x)| = |\alpha| \sup_{a\leqslant x\leqslant b} |f(x)| = |\alpha| p(f),$$

więc $p(\alpha f) = |\alpha| \cdot p(f)$, co dowodzi homogeniczności funkcjonału p.

Z powyższych rozważań wynika, że funkcjonał p zdefiniowany powyżej jest podliniowy na przestrzeni $C([a,b])$.

Wówczas z twierdzenia Hahna-Banacha dostajemy, że każdy funkcjonał liniowy $l \in S_d$, dla którego $\forall_{w\in S_d}\ l(w) \leqslant p(w)$ możemy przedłużyć na całą przestrzeń $C([a,b])$ poprzez funkcjonał $L \in \text{lin}(C([a,b]), R)$, który ponadto spełnia $\forall_{f\in C([a,b])}\ L(f) \leqslant p(f) = \sup_{a\leqslant x\leqslant b} |f(x)|$ co oznacza, że jest ograniczony. Konstruujemy l w taki sposób, że $l(w) = 0$ dla każdego $w \in S_d$, oczywiście zachodzi wówczas $l(w) \leqslant p(w)$.

Funkcjonał L jest przedłużeniem l z S_d, a ponieważ $\forall_{w \in S_d}\ l(w) = 0$ to mamy $L(w) = l(w) = 0$ dla każdego $w \in S_d$. Ponieważ założyliśmy, że S_d jest właściwą podprzestrzenią, to istnieje funkcja $f_\alpha \in C([a,b])$, która nie należy do S_d. Dla tej funkcji $L(f_\alpha) \neq 0$, zatem $L \neq 0$.

Zgodnie z twierdzeniem Riesza-Frecheta, ponieważ funkcjonał L jest liniowy i ograniczony na przestrzeni $C([a,b])$ to może on być przedstawiony w postaci iloczynu skalarnego względem pewnej miary borelowskiej $\mu \in B([a,b])$, jako

$$\forall_{f \in C([a,b])} \quad L(f) = \int_a^b f(x)\, d\mu(x),$$

gdzie całka pochodzi od iloczynu skalarnego w przestrzeni $C(a,b)$. Ponieważ $\forall_{y \in \mathbb{R}^n, \theta \in \mathbb{R}}\ \sigma(y^T x + \theta)$ jest w S_d, a $L(w)$ jest równe 0 dla każdego $w \in S_d$ otrzymujemy, że

$$\forall_{y \in \mathbb{R}^n, \theta \in \mathbb{R}} \quad \int_a^b \sigma(y^T x + \theta) d\mu(x) = 0.$$

Jednakże w twierdzeniu zakładamy, że funkcja σ jest dyskryminacyjna, co daje nam, że $\mu = 0$. Otrzymaliśmy w tym kroku, że każdy element $h \in C([a,b])$ należy do domknięcia przestrzeni S, co jest sprzeczne z naszym założeniem, że domknięcie S nie jest całą przestrzenią $C([a,b])$. Stąd zbiór S jest gęsty w $C([a,b])$. □

Twierdzenie o Uniwersalnej Aproksymacji daje nam teorytyczne uzasadnienie, że implementacja sieci neuronowej w sposób, jaki jest opisany w twierdzeniu może być bardzo efektywnym narzędziem w dziedzinie rozwiązywania równań różniczkowych. Daje nam ono również nowy sposób dochodzenia do rozwiązań. W przeciwieństwie do metod numerycznych, w których przybliżeń dokonujemy wychodząc od definicji pochodnej, w przypadku twierdzenia tworzymy funkcję $G(x)$, której bliskość od rozwiązania zależy od odpowiednio dobranych wag w_i i b_i. Pomysł G. Cybenko [3] był zdecydowanie nowym spojrzeniem i przełomem w narzędziach do szukania funkcji rozwiązań równań różniczkowych zwyczajnych. W dalszych sekcjach będziemy stosować to twierdzenie, aby badać skuteczność sieci neuronowych jako uniwersalnych aproksymatorów.

4. Podstawowe pojęcia z dziedziny sieci neuronowych

4.1. Pojęcie neuronu i perceptronu

Aby skontruować wieloneuronową sieć konieczne jest zdefiniowanie matematycznego modelu pojedyńczego neuronu. Pierwszy matematyczny model neuronu opracowany został w 1943 roku przez naukowców, na których cześć nadano mu nazwę *McCulloch-Pitts neuron*, co dało korzenie do implementacji współczesnych sieci neuronowych. Neuron ten przyjmuje na wejściu dwa sygnały, 0 albo 1 i oblicza sumę ważoną (4.1) wszystkich tych wejść. Założeniami modelu jest, że jeżeli suma ta przekroczy pewną progową wartość, neuron aktywuje się, w przeciwnym razie pozostanie nieaktywny [4]. Sumę tę można zapisać za pomocą symboli x_i – wartość sygnału na i-tym wejściu, w_i – waga przypisana na i-tym wejściu oraz b – próg aktywacji powszechnie nazywany jako bias *(pl. wyraz wolny)*. W rezultacie, informacją o wszystkich wejściach neuronu dla danego wektora wag $\theta := (b, [w_1, ..., w_n])$ będzie poniższa suma:

$$\sum_{i=1}^{n}(x_i \cdot w_i) + b. \tag{4.1}$$

Następnie na wyjście neuronu kładziemy powyższą sumę obłożoną funkcją aktywacji. Na jej podstawie neuron ten decyduje czy aktywować siebie czy nie. W modelu McCullocha i Pittsa na sumę ważoną nakłada się funkcję skokową, która zwraca 1, jeżeli neuron przekroczy pewien próg aktywacji τ, a w przeciwnym razie zwraca 0:

$$y = \begin{cases} 1 & \text{gdy } \sum_{i=1}^{n}(x_i \cdot w_i) + b > \tau, \\ 0 & \text{w przeciwnym razie.} \end{cases}$$

Powyższy model neuronu jest stosowany współcześnie, a za sprawą rozwoju coraz to lepszych komputerów zaczęto rozważać różne połączenia, architektury ułożenia neuronów jak i funkcje aktywacji, aby osiągnąć najbardziej skuteczne sieci neuronowe dla konkretnego problemu. Perceptronem nazywamy model składający się z jednego bądź wielu niezależnych neuronów McCullocha-Pittsa. Model sieci, w którym jest wiele neuronów ułożonych w kilka warstw nazywamy modelem Perceptronu Wielowarstwowego *(ang. Multi Layer Perceptron)*. Implementacje takiego modelu przedstawiliśmy w poniższych krokach.

4.2. Implementacja Perceptronu Wielowarstwowego

We współczesnej specyfikacji rozbudowanej sieci neuronowej może być umieszczona duża liczba neuronów oraz warstw ukrytych. Wówczas każdy neuron jest najczęściej połączony z każdym neuronem z kolejnej warstwy ukrytej lub warstwy wyjściowej, tworząc w ten sposób sekwencję funkcji, które mogą się aktywować lub być nieaktywne w zależności od wejść i wag połączonych z konkretnym neuronem. Przejście przez całą sieć neuronową, od wejścia przez wszystkie jej warstwy ukryte i otrzymanie wyliczonego z nich wyjścia, nazywamy propagacją wprzód *(ang. forward propagation)* [1, 6]. W zagadnieniu implementacji sztucznej sieci neuronowej dla problemu rozwiązywania równań różniczkowych zwyczajnych jako wejście dysponujemy odcinkiem na którym chcemy otrzymać rozwiązanie równania – pewnym wektorem $X = [x_1, ..., x_n]$, na podstawie którego chcemy przewidzieć jak wyglądać będzie funkcja rozwiązująca równanie – wektor wartości $Y = f(X) = [f(x_1), ..., f(x_n)]$ – wyjścia. Niezbędne do dalszej pracy jest wprowadzenie poniższych interpretacji wektorów wag (ozn. W) oraz wektorów biasów (ozn. b). W poprzedniej sekcji gdzie zdefiniowaliśmy $\theta = (b, [w_1, ..., w_n])$ dla modelu jednego neuronu tym razem musimy określić postać Θ dla liczby l warstw (nazywanych warstwami ukrytymi), w których mamy kolejno $k_1, k_2, ..., k_l$ neuronów. Aby zapewnić, by wektor wyjściowy był wymiaru n ustawimy $k_l = n$. Wówczas jako bias dla k-tego neuronu w j-tej warstwie ukrytej stoi b_k^j zaś jako wartość wagi k-tego neuronu w j-tej warstwie ukrytej stoi $w_{k,i}^j$, gdzie i to pozycja neuronu w poprzedniej warstwie ukrytej (lub wektorze wejściowym). Takie połączenia implikują następującą postać wektora wag Θ dla jednej warstwy ukrytej (modelu perceptronu jednowarstwowego, w którym jako poprzednia „warstwa" stoi wektor wejściowy X:

$$\Theta = (b, W) = (b^1, W^1) = \left([b_1^1, b_2^1, ..., b_{k_1}^1], \begin{bmatrix} w_{1,1}^1 & w_{2,1}^1 & \cdots & w_{k_1,1}^1 \\ \vdots & \vdots & \ddots & \vdots \\ w_{1,n}^1 & w_{2,n}^1 & \cdots & w_{k_1,n}^1 \end{bmatrix} \right),$$

oraz postać wektora wag dla l warstw ukrytych (a więc modelu perceptronu wielowarstwowego):

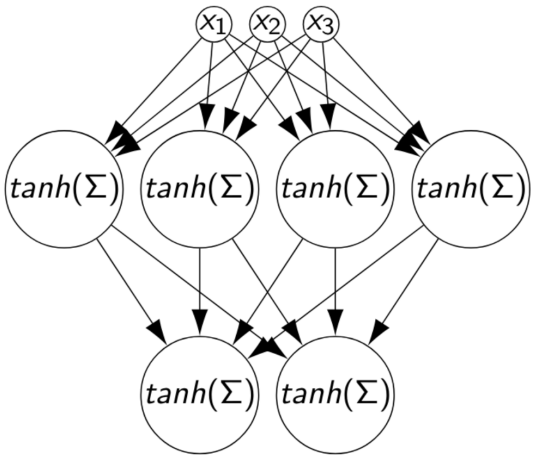
$$\Theta = (b, W) = (b^1, W^1, b^2, W^2, \cdots, b^l, W^l) = \left([b_1^1, b_2^1, ..., b_{k_1}^1], \begin{bmatrix} w_{1,1}^1 & w_{2,1}^1 & \cdots & w_{k_1,1}^1 \\ \vdots & \vdots & \ddots & \vdots \\ w_{1,n}^1 & w_{2,n}^1 & \cdots & w_{k_1,n}^1 \end{bmatrix} \right.,$$

$$\left. [b_1^2, b_2^2, ..., b_{k_2}^2], \begin{bmatrix} w_{1,1}^2 & w_{2,1}^2 & \cdots & w_{k_2,1}^2 \\ \vdots & \vdots & \ddots & \vdots \\ w_{1,k_1}^2 & w_{2,k_1}^2 & \cdots & w_{k_2,k_1}^2 \end{bmatrix}, \cdots \right),$$

$$[b_1^l, b_2^l, ..., b_{k_l}^l], \begin{bmatrix} w_{1,1}^l & w_{2,1}^l & \cdots & w_{k_l,1}^l \\ \vdots & \vdots & \ddots & \vdots \\ w_{1,k_{l-1}}^l & w_{2,k_{l-1}}^l & \cdots & w_{k_l,k_{l-1}}^l \end{bmatrix}\Bigg).$$

4.2.1 Propagacja wprzód

Propagacja wprzód jest podstawową częścią działania sieci neuronowej, w której przy danym wektorze wejścia $X = [x_1, x_2, ..., x_n]$ obliczane jest wyjście przy wektorze Θ – początkowo wektor ten ma postać losowo wybranych wartości, na przykład z rozkładu normalnego. Przykładowo, dla l warstw ukrytych w sieci neuronowej i przy $n = 3$ oraz czterech neuronach w pierwszej warstwie ukrytej, dla każdego k-tego, $k = 1, 2, 3, 4$, neuronu losujemy każdą z wartości wag połączoną z nim: $w_{1,1}^l$, ..., $w_{4,3}^l$, oraz wartość przypisanego mu unikalnego biasu – b^k. Z powyższych wartości układamy równanie macierzowe jak w (4.2) zwracające $(\sum_1^1, ..., \sum_4^1) = (y_1^1, ...y_4^1)$, dzięki któremu obliczyć możemy wyjście dla pierwszej warstwy ukrytej. Na to wyjście możemy nałożyć następnie funkcję aktywacji (przykładowo tangens hiperboliczny), aby aktywować poszczególne neurony. Przykładowy ruch *forward* ukazaliśmy na rysunku 4.1.



Rysunek 4.1: Przykład dwu-warstwowej sieci neuronowej z funkcją aktywacji tanh [4]

W testach rozważać będziemy również dwie inne funkcje aktywacji takie jak sigmoidalną σ, czy ReLu *(ang. rectified linear unit)*:

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \qquad \text{ReLu}(x) = \max(0, x).$$

Funkcja $\sigma(x)$ jest powszechnie używaną funkcją aktywacji, zauważmy, że spełnia ona warunki funkcji sigmoidalnej i dyskryminacyjnej:

$$\lim_{t \to +\infty} \sigma(t) = \lim_{t \to +\infty} \frac{1}{1 + e^{-t}} = \frac{1}{1 + 0} = 1, \qquad \lim_{t \to -\infty} \sigma(t) = \lim_{t \to -\infty} \frac{1}{1 + e^{-t}} = 0.$$

Zatem σ istotnie jest sigmoidalna. Teraz przyjmijmy $\mu \in B(I)$, gdzie I – skończone. Jak pokazaliśmy σ przyjmuje wartości z przedziału (0,1). Ponieważ $\sigma(y^T x+\theta)$ jest ciągła i zawsze dodatnia na całym I, to, jedynym sposobem, aby całka z ciągłej i dodatniej funkcji spełniała warunek:

$$\forall_{y,\theta} \quad \int_I \frac{1}{1+e^{-(y^T x+\theta)}} d\mu(x) = 0,$$

jest, że μ na I jest 0, co implikuje również dyskryminacyjność. Widzimy, że funkcja σ spełnia założenia twierdzenia o Uniwersalnej Aproksymacji, co skłania nas, aby implementacje sieci neuronowej przeprowadzać właśnie na tej funkcji aktywacji.

W kolejnym kroku przeprowadzimy proces aktywowania kolejnych neuronów przy sigmoidalnej funkcji aktywacji, dla uogólnionej architektury l warst ukrytych. Wyjście dla pierwszej warstwy ukrytej (przy ustalonej dla niej liczbie neuronów równej k_1), przy danym wektorze wejściowym X, macierzy wag W oraz wektorze biasów b, możemy obliczyć w następujący sposób:

$$y^1 = x \cdot W^1 + b^1 = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix} \begin{bmatrix} w^1_{1,1} & w^1_{2,1} & \cdots & w^1_{k_1,1} \\ w^1_{1,2} & w^1_{2,2} & \cdots & w^1_{k_1,2} \\ \vdots & \vdots & \vdots & \vdots \\ w^1_{1,n} & w^1_{2,n} & \cdots & w^1_{k_1,n} \end{bmatrix} + \begin{bmatrix} b^1_1 & b^1_2 & \cdots & b^1_{k_1} \end{bmatrix}. \quad (4.2)$$

W ten sposób dostaniemy wyjście z pierwszej warstwy ukrytej, k_1-elementowy wektor y^1, na który możemy nałożyć funkcję aktywacji, otrzymując w ten sposób decyzję o aktywacji poszczególnych neuronów:

$$Z^1 = \sigma(y^1) = \sigma\left(\begin{bmatrix} y^1_1 & y^1_2 & \cdots & y^1_{k_1} \end{bmatrix}\right). \quad (4.3)$$

Możemy postępować analogicznie jak w (4.2)–(4.3), aby stworzyć kolejną warstwę ukrytą, tym razem przyjmując Z^1 jako wejście do drugiej warstwy ukrytej oraz W^2, b^2 jako wagi i biasy dla nowej warstwy. W ten sposób otrzymujemy kolejno: Z^2 – wyjście z drugiej warstwy ukrytej, Z^3 – wyjście z trzeciej warstwy ukrytej. Przypuśćmy, że wektor y^l jest wyjściem z ostatniej warstwy ukrytej, bez nałożonej na niego funkcji aktywacji. Wówczas wektor ten jest predykcją naszej sieci neuronowej oraz zakończyliśmy działanie propagacji wprzód. W przypadku aproksymacji wektora wyjściowego istotnie nie będziemy nakładać funkcji aktywacji na wyjście z sieci, posłużymy się więc w tym celu zwykłą funkcją liniową i otrzymamy predykcję $Y = f([x_1, x_2, ..., x_n]) = [y^l_1, ...y^l_n]$. Na koniec powyższych obliczeń możemy obliczyć jak bardzo nasza sieć neuronowa się myli. Po-

służmy się przypadkiem procesu rozwiązywania równania różniczkowego zwyczajnego pierwszego rzędu oraz jego warunku początkowego:

$$\frac{df(x)}{dx} = f(x), \qquad f(x_0) = y_0.$$

Chcemy, aby sieć neuronowa jak najskuteczniej rozwiązywała poniższy problem minimalizacji:

$$\min_{\theta} \left(\frac{df(x)}{dx} - f(x) \right). \tag{4.4}$$

Zauważmy, że w naszym przypadku, jak i ogólnie w algorytmach uczenia maszynowego minimalizowania błędu dokonujemy po wagach θ. W tym celu wprowadzić musimy tak zwaną **funkcję próby**, która na podstawie Teorii Uniwersalnej Aproksymacji reprezentuje wyrażenie $f(x)$ w terminie omówionej wyżej propagacji wprzód. Funkcja próby jest przesunięciem rozwiązania danego przez sieć neuronową o dane warunki początkowe x_0 i $f(x_0)$. Oznaczmy $Forward(x, \theta) := y^l$, a więc jako wyjście z naszego pełnego perceptronu wielowarstwowego. W naszych testach będziemy rozważać dwie różne funkcje próby wymienione w tabeli 4.1.

Rodzaj funkcji próby	Formuła
Wielomianowa	$g_t(x) = f(x_0) + (x - x_0)Forward(x, \theta)$
Eksponencjalna	$g_t(x) = f(x_0) + (1 - e^{-(x-x_0)})Forward(x, \theta)$

Tabela 4.1: Wybrane funkcje próby dla problemu minimalizacji

Wówczas wyrażenie (4.4) sprowadza nam się do:

$$\min_{\theta} \left(\frac{d}{dx} g_t(x_i, \theta) - g_t(x_i, \theta) \right).$$

W terminach uczenia maszynowego wprowadzimy średni błąd kwadratowy na wyprowadzonej wyżej różnicy i otrzymany w ten sposób wzór nazwiemy **funkcją kosztu**:

$$C(x, \theta) = \frac{1}{N} \sum_{i=1}^{N} \left(\frac{d}{dt} g_t(x_i, \theta) - g_t(x_i, \theta) \right)^2. \tag{4.5}$$

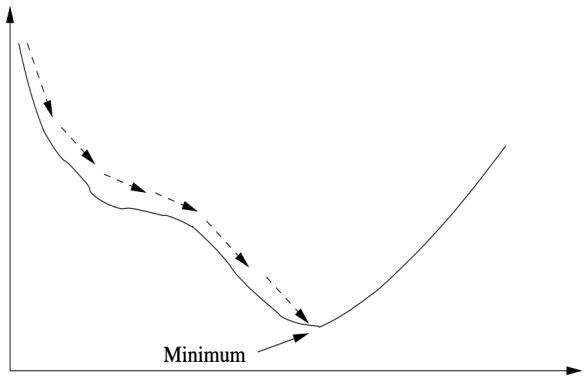
Rozwiązanie problemu minimalizacji dla funkcji (4.5) gwarantuje nam skuteczność aproksymowania rozwiązań równań różniczkowych zwyczajnych dla naszej metody. Optymalizacji funkcji $C(x, \theta)$ dokonywać będziemy za pomocą algorytmu spadku wzdłuż gradientu oraz stochastycznego spadku wzdłuż gradientu.

4.2.2 Optymalizacja metodą spadku wzdłuż gradientu

Optymalizacja metodą spadku wzdłuż gradientu polega na minimalizacji funkcji kosztu poprzez iteracyjne aktualizowanie parametrów modelu w kierunku, w którym funkcja kosztu maleje najszybciej. Algorytm ten wykorzystuje pochodne funkcji kosztu względem parametrów, aby

określić kierunek zmiany parametrów, a następnie aktualizuje parametry zgodnie z tym kierunkiem [6]. Proces ten powtarzamy iteracyjnie aż do osiągnięcia zadowalającego poziomu minimalizacji funkcji kosztu (jak na rysunku 4.2). Metodę tę zastosujemy do minimalizacji funkcji $C(x, \theta)$ po parametrach θ:

$$\frac{\partial C(x,\theta)}{\partial \theta} = \frac{\partial}{\partial \theta}\frac{1}{N}\sum_{i}^{N}\left(\frac{dg_t(x_i,\theta)}{dx} - g_t(x_i,\theta)\right)^2.$$



Rysunek 4.2: Przykład iteryjnej zbieżności algorytmu spadku wzdłuż gradientu do minimum [4]

Wypiszemy wyprowadzenie pochodnej tej funkcji po parametrach $\theta = (b^1, W^1, ..., b^l, W^l)$. W tym celu skorzystamy z reguły łańcuchowej. Pokażemy proces obliczania wartości $C(x, \theta)$ dla wartości wagi w ostatniej warstwie $w^l_{1,1}$, dla obserwacji x_1. Skorzystamy z tego, że funkcja $g_t(x, \theta)$ jest gładka, ponieważ $\forall_x \sigma(x)$ jest gładka.

$$\begin{aligned}
C_l :&= \frac{\partial}{\partial w^l_{1,1}}\left(\frac{dg_t(x_1,\theta)}{dx} - g_t(x_1,\theta)\right)^2 \\
&= 2\left(\frac{dg_t(x_1,\theta)}{dx} - g_t(x_1,\theta)\right)\cdot\frac{\partial}{\partial w^l_{1,1}}\left(\frac{dg_t(x_1,\theta)}{dx} - g_t(x_1,\theta)\right) \\
&= 2\left(\frac{dg_t(x_1,\theta)}{dx} - g_t(x_1,\theta)\right)\cdot\frac{\partial}{\partial w^l_{1,1}}\frac{dg_t(x_1,\theta)}{dx} \quad - 2\left(\frac{dg_t(x_1,\theta)}{dx} - g_t(x_1,\theta)\right)\cdot\frac{\partial}{\partial w^l_{1,1}}g_t(x_1,\theta) \\
&= \underbrace{K\frac{\partial}{\partial w^l_{1,1}}\frac{dg_t(x_1,\theta)}{dx}}_{(*)} - \underbrace{K\frac{\partial}{\partial w^l_{1,1}}g_t(x_1,\theta)}_{(**)},
\end{aligned}$$

gdzie

$$\begin{aligned}
K &= 2\left(\frac{dg_t(x_1,\theta)}{dx} - g_t(x_1,\theta)\right) \\
&= 2\left(\frac{d}{dx}\left(y_0 + (x_1 - x_0)Forward(x_1,\theta)\right) - \left(y_0 + (x_1 - x_0)Forward(x_1,\theta)\right)\right) \\
&= 2\left(\frac{d}{dx}(y_0 + (x_1 - x_0)\sigma(y^l_1)) - (y_0 + (x_1 - x_0)\sigma(y^l_1))\right)
\end{aligned}$$

Dalej możemy wyprowadzić rozwinięcie dla $(**)$ w następujący sposób:

$$\begin{aligned}
K\frac{\partial}{\partial w^l_{1,1}} g_t(x_1,\theta) &= K\frac{\partial(y_0+(x_1-x_0)\sigma(y^l_1))}{\partial w^l_{1,1}} \\
&= K(x_1-x_0)\sigma'(y^l_1)\frac{\partial y^l_1}{\partial w^l_{1,1}} \\
&= K(x_1-x_0)\sigma'(y^l_1)\frac{\partial\left(\sum_j w^l_{1,j}\sigma(y^{l-1}_j)+b_j\right)}{\partial w^l_{1,1}} \\
&= K(x_1-x_0)\sigma'(y^l_1)\frac{\partial\left(w^l_{1,1}\sigma(y^{l-1}_1)+b_1+...+w^l_{1,k_l}\sigma(y^{l-1}_{k_l})+b_{k_l}\right)}{\partial w^l_{1,1}} \\
&= K(x_1-x_0)\sigma'(y^l_1)\sigma(y^{l-1}_1).
\end{aligned}$$

Wówczas dostaniemy:

$$\begin{aligned}
C_l &= K\frac{\partial}{\partial w^l_{1,1}}\frac{d}{dx}((x_1-x_0)\sigma(y^l_1)) - K(x_1-x_0)\sigma'(y^l_1)\sigma(y^{l-1}_1) \\
&= K\frac{\partial}{\partial w^l_{1,1}}\sigma(y^l_1) - K(x_1-x_0)\sigma'(y^l_1)\sigma(y^{l-1}_1) \\
&= K\sigma'(y^l_1)\sigma(y^{l-1}_1) - K(x_1-x_0)\sigma'(y^l_1)\sigma(y^{l-1}_1) \\
&= K\sigma'(y^l_1)\sigma(y^{l-1}_1)(1-(x_1-x_0)).
\end{aligned}$$

Postępując analogicznie dla wszystkich parametrów θ od ostatniej warstwy do początkowej w rekurencyjny sposób otrzymamy wartości wektora:

$$\frac{\partial C}{\partial \theta} = \left[\frac{\partial C}{\partial W^1}, \frac{\partial C}{\partial b^1}, \frac{\partial C}{\partial W^2}, \frac{\partial C}{\partial b^2}, ..., \frac{\partial C}{\partial W^l}, \frac{\partial C}{\partial b^l}\right],$$

gdzie dla p–tej warstwy ukrytej, przy czym $p = l-1, l-2, ..., 1$, zachodzi:

$$\frac{\partial C}{\partial W^p} = delta^p \cdot \left(\sigma(y^{p-1})\right)^T,$$

$$\frac{\partial C}{\partial b^p} = \sum_i^n delta^p.$$

Aby dojść do postaci $delta^p$ – różnicy błędu na p-tej warstwie ukrytej wyjaśnimy pojęcie propagacji wstecznej błędu.

4.2.3 Propagacja wsteczna

Celem ruchu propagacji wstecznej *(ang. Backward propagation)* [4, 6], a więc od wyjścia sieci neuronowej do jej wejścia, jest poprawienie wag oraz biasów wszystkich neuronów znajdujących się w architekturze sieci, tak aby zminimalizować błąd $C(x,\theta)$ za pomocą metody optymalizacyjnej na przykład spadku wzdłuż gradientu. Ważną metodą w propagacji wstecznej jest 2. reguła perceptronu, która mówi, że jeśli neuron nie jest aktywowany, a powinien być, to zmieniamy wagę

tak, żeby zwiększyć jego szansę aktywacji [4, 6]. Pierwszym krokiem jest policzenie różnicy błędu, co wcześniej oznaczyliśmy jako $delta^p$, na p-tej warstwie ukrytej. Dokonujemy tego za pomocą iloczynu dla macierzy wag neuronów w kolejnej warstwie ukrytej (lub wektorze wyjściowym) jak w (4.6), przy ustalonej liczbie $k_1, ..., k_l$ neuronów oraz n elementach wektora wyjściowego. Następnie wyrażenie to mnożymy przez pochodną funkcji aktywacji.

Dla $p = l$, czyli ostatniej warstwy ukrytej mamy:

$$delta^l = \begin{bmatrix} w^l_{1,1} & w^l_{2,1} & \cdots & w^l_{n,1} \\ w^l_{1,2} & w^l_{2,2} & \cdots & w^l_{n,2} \\ \vdots & \vdots & \vdots & \vdots \\ w^l_{1,k_l} & w^l_{2,k_l} & \cdots & w^l_{n,k_l} \end{bmatrix}^T \cdot 2\left(\frac{dg_t(x,\theta)}{dx} - g_t(x,\theta)\right)\sigma'(y^l), \tag{4.6}$$

przy czym dla każdej poprzedniej $p = l-1, l, ..., 1$–szej warstwy postępujemy według zależności:

$$delta^p = \begin{bmatrix} w^{p+1}_{1,1} & w^{p+1}_{2,1} & \cdots & w^{p+1}_{k_{p+1},1} \\ w^{p+1}_{1,2} & w^{p+1}_{2,2} & \cdots & w^{p+1}_{k_{p+1},2} \\ \vdots & \vdots & \vdots & \vdots \\ w^{p+1}_{1,k_p} & w^{p+1}_{2,k_p} & \cdots & w^{p+1}_{k_{p+1},k_p} \end{bmatrix}^T delta^{p+1}\sigma'(y^p). \tag{4.7}$$

Kolejnym krokiem jest obliczenie gradientów p–tej warstwy ukrytej. Poniżej kolejno przedstawione są rezultaty dla wag oraz biasów każdej z warstw. Dla rozpatrywanej p–tej warstwy ukrytej na podstawie zależności wyprowadzonych w poprzedniej sekcji mamy:

$$grad^p_w = \frac{\partial C}{\partial W^p},$$

$$grad^p_b = \frac{\partial C}{\partial b^p}.$$

Otrzymane w ten sposób gradienty na p-tej warstwie ukrytej możemy zapisać jako:

$$\forall_{p=l-1,...,1} \quad grad^p = (grad^p_w, grad^p_b).$$

Następnie musimy zaktualizować wagi dla każdej warstwy ukrytej. W tym celu wprowadzamy krok uczenia, oznaczany jako η i najczęściej przyjmujemy go na poziomie $\eta = 0{,}001$. Dalej dokonujemy aktualizacji wag za pomocą $grad^p$ i η:

$$\forall_{p=l-1,\ldots,1} \quad \Theta^p = (W^p,\, b^p) + \eta\ grad^p = (W^p + \eta grad_w^p, b^p + \eta grad_b^p).$$

W momencie gdy zakończymy na wejściu do pierwszej warstwy ukrytej kończone jest działanie algorytmu propagacji wstecznej oraz od nowa przeprowadzana jest propagacja wprzód, tym razem już dla zaktualizowanych wag i biasów. Kolejne iteracje ruchu *forward-backward* nazywamy epokami *(ang. epoch)*. Zadaniem każdej z epok jest obliczenie powyższych gradientów i zminimalizowanie ostatecznego błędu $C(\theta, x)$. Algorytm uczenia przetwarzamy do momentu osięgnięcia pewnego warunku stopu. Możemy go opisać tak jak w Pseudokodzie 1. Warunkiem stopu w tym pseudokodzie może być na przykład osiągnięcie ustalonej liczby epok lub zamierzonej wartości błędu $C(x, \theta)$.

Wyróżniamy również metodę stochastycznego spadku wzdłuż gradientu *(ang. stochastic gradient descent)*. W przeciwieństwie do podstawowej optymalizacji spadkiem gradientu, gdzie na każdym kroku algorytmu obliczamy gradient na podstawie wszystkich próbek w zbiorze treningowym, w tej metodzie na każdym kroku algorytmu obliczamy gradient na podstawie tylko jednej próbki, jak w Pseudokodzie 2. Ta jedna zmiana czyni algorytm stochastyczny szybszym, a więc bardziej efektywnym dla dużego wektora wejściowego. Kolejną zmianą jest większa podatność na niestabilne ścieżki spadku. W niektórych sytuacjach pozwala to metodzie uciec od lokalnych minimów, lecz czasami nie dać tak dobrego minimum jak w metodzie podstawowego spadku. Dobrą praktyką będzie przeprowadzenie testu na obydwu algorytmach.

Pseudokod 1 Algorytm uczenia sieci metodą spadku wzdłuż gradientu

```
1:  Θ ← InicjujLosowo;
2:  while ¬WarunekStopu do
3:      ΔΘ ← 0;
4:      for all X ∈ WektorWejscia do
5:          Ŷ ← g_t(Θ, X);
6:          grad ← Backward(Ŷ, Y);
7:          ΔΘ ← ΔΘ + ηgrad;
8:      end for
9:      Θ ← Θ + ΔΘ;
10: end while
```

Pseudokod 2 Algorytm uczenia sieci metodą stochastycznego spadku wzdłuż gradientu

```
1:  Θ ← InicjujLosowo;
2:  while ¬WarunekStopu do
3:      for all X ∈ WektorWejscia do
4:          Ŷ ← g_t(Θ, X);
5:          grad ← Backward(Ŷ, Y);
6:          ΔΘ ← 0;
7:          ΔΘ ← ΔΘ + ηgrad;
8:          Θ ← Θ + ΔΘ;
9:      end for
10: end while
```

5. Rezultaty na podstawie prostych implementacji

Aby sprawdzić skuteczność uczenia się sieci neuronowej przebadane zostały w tym celu dwie różne funkcje opisujące równanie różniczkowe zwyczajne. Początkowo rozpatrywana była klasa pierwszego rzędu na odcinku [0,1] oraz dwa wybrane równania dane jak w (5.1) i (5.2), które mają rozwiązania odpowiednio (5.3), (5.4):

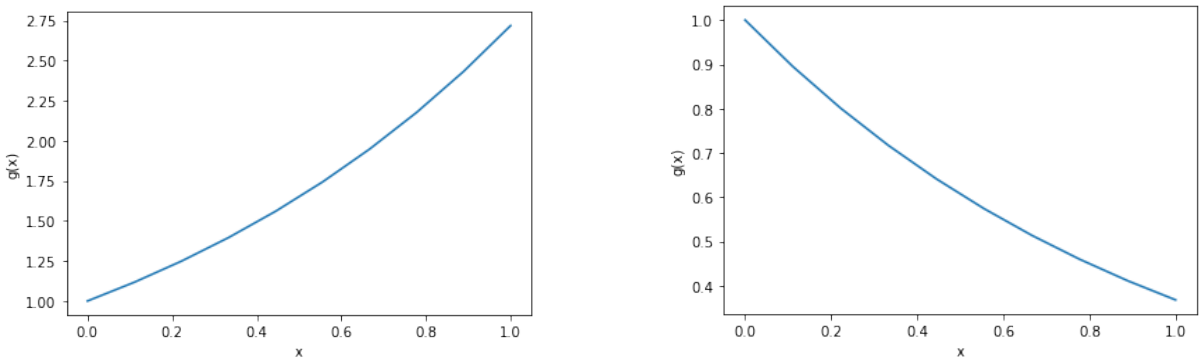
$$\frac{dy}{dx} = y \tag{5.1}$$

$$\frac{dy}{dx} = -y \tag{5.2}$$

$$y(x) = y_0 e^{x} \tag{5.3}$$

$$y(x) = y_0 e^{-x} \tag{5.4}$$

Rozpatrywany był warunek początkowy y_0=1, co daje nam rozwiązania analityczne powyższych równań jak na rysunku 5.1.



(a) Rozwiązanie równania (5.1) $y(x) = e^{x}$

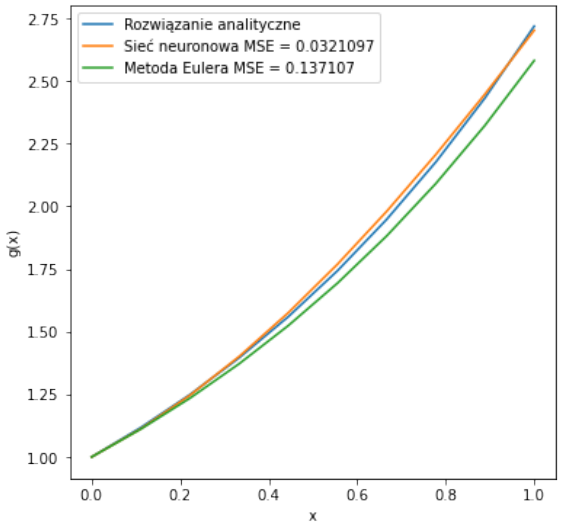
(b) Rozwiązanie równania (5.2) $y(x) = e^{-x}$

Rysunek 5.1: Analityczne rozwiązania wybranych równań różniczkowych na przedziale [0.1]

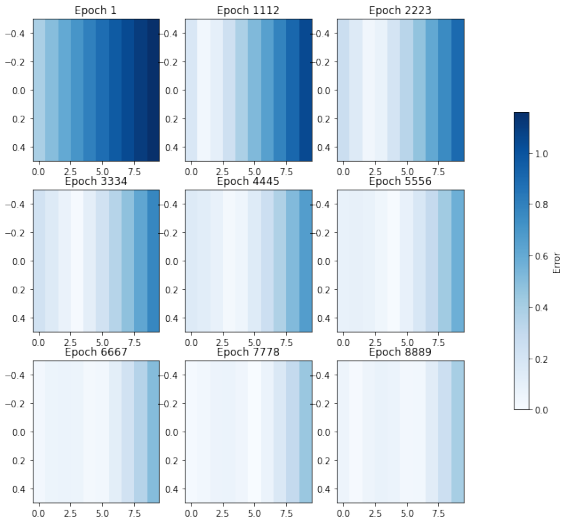
Dla przedstawionych dwóch równań testowaliśmy proces uczenia sieci neuronowej oraz jej rezultaty. Dla porównania dodaliśmy również metodę Eulera realizującą to samo zadanie. Obie metody aproksymowały rozwiązanie na przedziale [0,1] podzielonym na taką samą ilość stu równoodległych punktów. Jako miarę dokładności aproksymacji dla każdej metody wykorzystaliśmy średni błąd kwadratowy *(ang. mean squared error)* $MSE = \frac{1}{N}\sum_{i=1}^{N}(Y_i - \hat{Y}_i)^2$, w którym Y_i

jest i-tym punktem rozwiązania obliczonym analitycznie, zaś $\hat{Y}_i$ wartością obliczoną za pomocą metody.

Dla rozpatrywanych równań mierzyliśmy jak zmienia się błąd sieci neuronowej $C(x, \theta)$ na przestrzeni kolejnych epok (rysunek 5.4) oraz jak wygląda macierz $delta^l$ wag dla poszczegolnych epok (rysunek 5.2, 5.3 (b)). Ponadto przedstawiliśmy również rezultaty aproksymacji rozwiązania (rysunek 5.2, 5.3 (a)).

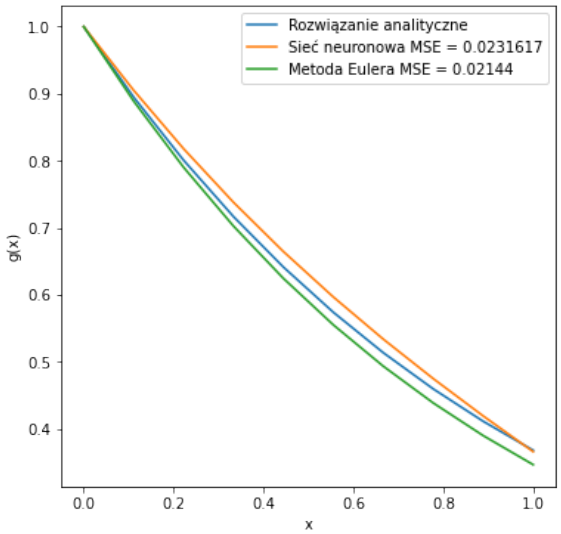


(a) Równanie (5.1) – aproksymowane rozwiązania

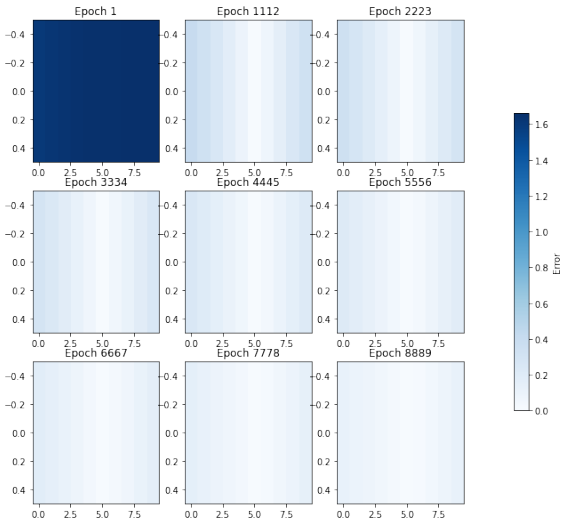


(b) Równanie (5.1) – zbieżność punktu x_i do rozwiązania

Rysunek 5.2: Wyniki przedstawiające proces uczenia dla równania (5.1)

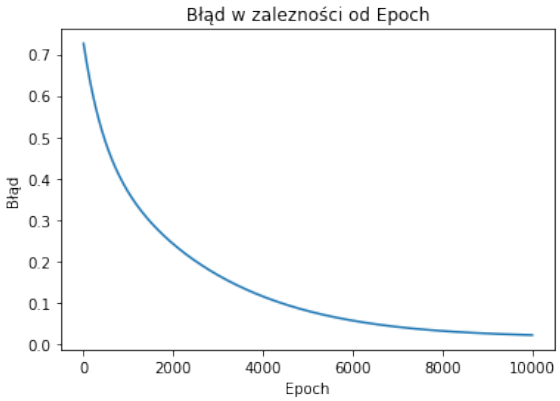


(a) Równanie (5.2) – aproksymowane rozwiązania

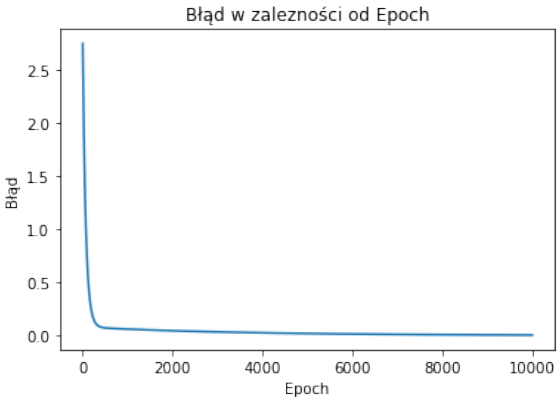


(b) Równanie (5.2) – zbieżność punktu x_i do rozwiązania

Rysunek 5.3: Wyniki przedstawiające proces uczenia dla równania (5.2)



(a) Równanie (5.1) – błąd sieci (wartości funkcji kosztu) w zależności od epoki



(b) Równanie (5.2) – błąd sieci (wartości funkcji kosztu) w zależności od epoki

Rysunek 5.4: Wyniki przedstawiające proces minimalizowania funkcji kosztu dla równania (5.1),(5.2)

Na rysunkach 5.2, 5.3, 5.4 widzimy, że implementacja z części czwartej istotnie pozwala sieci neuronowej realizować proces uczenia, minimalizować funkcję kosztu oraz obliczać rozwiązanie danego równania różniczkowego z pewną dokładnością. Każda z powyższych sieci została wytrenowana poprzez sigmoidalną funkcję aktywacji w jednej warstwie ukrytej oraz liniową w warstwie wyjściowej, z parametrem *learning rate* równym 0,001, liczbą epoch równą 10000 oraz wielomianową funkcją próby. Dla warstwy ukrytej w obu równaniach ustawiliśmy 10 nuronów. W przypadku obu metod podzieliliśmy odcinek [0,1] na 100 części tj. próbka danych treningowych dla sieci zawierała 100 obserwacji. Dokładne wyniki czasu oraz średniego błędu kwadratowego (sieć neuronowa / metoda Eulera) możemy zaobserwować w tabeli 5.1.

Równanie:	(5.1) $\frac{dy}{dx} = y$	(5.2) $\frac{dy}{dx} = -y$
Czas wykonania:	9,6s / 0,5s	9,7s / 0,5s
Dokładność aproksymacji:	0,0321 / 0,137	0,0232 / 0,0214

Tabela 5.1: Zestawienie dokładności aproksymacji (końcowej wartości średniego błędu kwadratowego) i czasu obliczeń (w sekundach) dla metody Eulera i prostej architektury sieci neuronowej

Choć udało nam się osiągnąć efekt uczenia i optymalizowania błędu, to wciąż sieć nie daje zadawalających wyników. Czas jej działania jest znacznie większy od metody Eulera, a daje zbliżony rezultat aproksymacji. Ponadto na wykresach, gdzie możemy obserwować wartość błędu w kolejnych epochach oraz jak punkty x_i zbiegają do rozwiąania widzimy, że proces uczenia przebiega wolno – w szczególności dla rozwiązania (5.1). W celu poprawy rezulatów w dalszych

krokach badać będziemy m.in. wpływ metod normalizacji gradientu, momentów, normalizacji danych i zwiększenia liczby warstw, aby poprawić nasze wyniki. Dalsze testowanie skuteczności sieci neuronowych opierać będzimy na wielu czynnikach, z których wyróżnić możemy następujące.

- Architektura sieci (liczba warstw ukrytych i neuronów w poszczególnych warstwach ukrytych);
- Metoda spadku wzdłuż gradientu, metoda stochastycznego spadku wzdłuż gradientu oraz metoda *batching*;
- Uczenie z zastosowaniem momentu i normalizacji gradientu lub bez użycia tych metod;
- Metoda losowania wag spośród: rozkład normalny, rozkład jednostajny na przedziale [0,1], Xavier oraz He;
- Wybór funkcji próby spośród wieolomianowej i eksponencjalnej;
- Wybór funkcji aktywacji spośród sigmoidalnej, tanh oraz ReLU.

6. Metody ulepszeń sieci neuronowej

6.1. Proces uczenia sieci neuronowej z momentem

Sposobem poprawienia procesu uczenia jest zaimplementowanie metody uczenia z momentem. Sposób ten ma za zadanie przyspieszyć proces uczenia poprzez szybsze zbieganie w kierunku minimum funkcji kosztu. Moment *(tablica Momentum)* zapamiętuje kierunek, w którym w poprzednich krokach były zmiany wag, co pozwala na szybsze poruszanie się wzdłuż gradientu. Ponadto dzięki temu sieć neuronowa może unikać sytuacji gdzie gradient jest bliski zeru i gdzie uczenie bez momentu utyka w lokalnym minimum. W celu zaimplementowania uczenia z momentem definiujemy nowy parametr λ – współczynnik wygaszania momentu. Zapamiętywania kierunków zmian wag w poprzednich krokach dokonujemy za pomocą tablicy Momentum, którą aktualizujemy o nowe zmiany dla każdej nowej wartości gradientu w kolejnych epokach. Przedstawiliśmy procedurę uczenia z momentem za pomocą Pseudokodu 3.

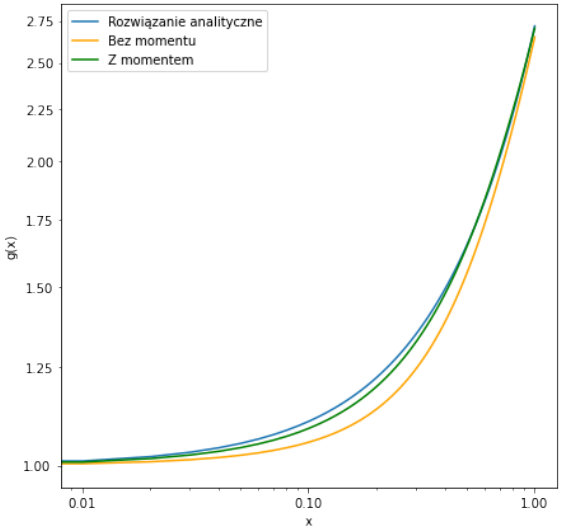
Pseudokod 3 Proces uczenia z momentem

```
1: Θ ← InicjujLosowo;
2: Momentum ← [0, 0, ..., 0];
3: while ¬StopCondition do
4:     grad ← [0, 0, ..., 0];
5:     for all X ∈ WektorWejscia do
6:         Ŷ ← g_t(Θ, X);
7:         grad ← grad − ∇g_t(Θ, X);
8:     end for
9:     Moment ← grad + Moment · λ;
10:    Θ ← Θ + η · Moment;
11: end while
```

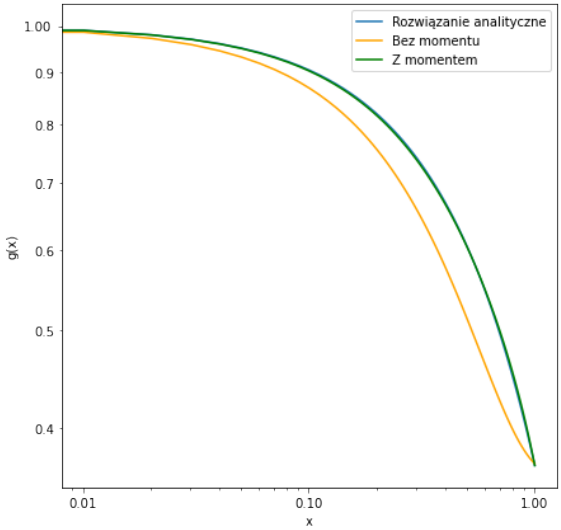
Zaimplementowanie powyższego ulepszenia w sieci neuronowej dało dużo lepsze rezultaty, niż w przypadku niekorzystania z tej motedy co widzimy w tabeli 6.1 i na rysunku 6.1 oraz 6.2. Zmniejszyliśmy również liczbę epok z 10000 do 1000 pozostawiając resztę parametrów bez zmian.

Sposób uczenia	Bez metody momentu	Z metodą momentu
Końcowy $C(x, \theta)$ na odcinku [0,1] (równ. 5.1):	0,05062	0,01437
Końcowy $C(x, \theta)$ na odcinku [0,1] (równ. 5.2):	0,05051	0,00035

Tabela 6.1: Porównanie wyników dla metody uczenia z momentem

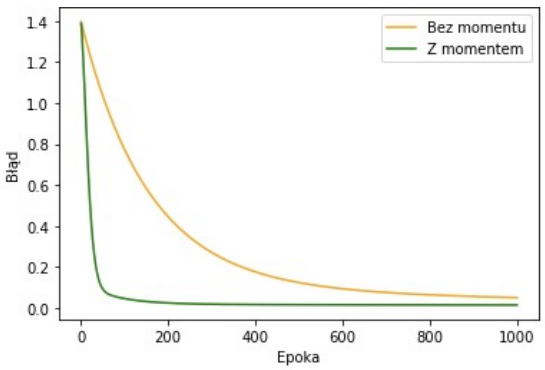


(a) Równanie (5.1) – aproksymacja rozwiązania

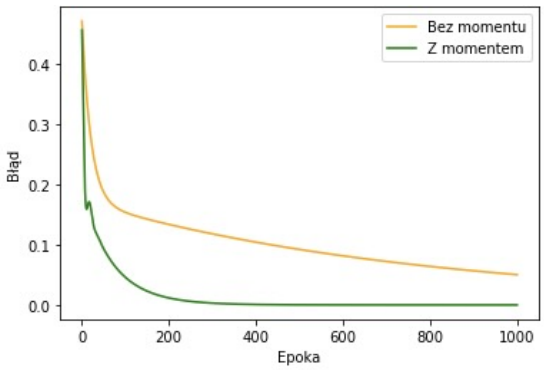


(b) Równanie (5.2) – aproksymacja rozwiązania

Rysunek 6.1: Wyniki przedstawiające proces uczenia sieci neuronowej dla metody z momentem i bez dla równań (5.1),(5.2) – dokładność aproksymacji w skali logarytmicznej



(a) Równanie (5.1) – błąd sieci (wartości funkcji kosztu) w zależności od epoki



(b) Równanie (5.2) – błąd sieci (wartości funkcji kosztu) w zależności od epoki

Rysunek 6.2: Wyniki przedstawiające proces uczenia sieci neuronowej dla metody z momentem i bez dla równań (5.1),(5.2) – wartość funkcji kosztu dla kolejnych epok

Możemy zauważyć, że w przypadku procesu uczenia z momentem sieć neuronowa minimalizuje funkcję kosztu znacznie szybciej (już na poziomie początkowych epok, w pierwszych stu iteracjach, widzimy wyraźny spadek do zera) od podstawowej implementacji. Oznacza to, że dzięki tej metodzie możemy skrócić czas uczenia, co może mieć znaczny wpływ przy większych odcinkach podziału funkcji i większych przedziałach.

6.2. Normalizacja gradientu – RMSprop *(ang. Root Mean Square Propagation)*

Główną zaletą implementowania normalizacju gradientu jest zapobieganie problemowi zanikającego lub „wybuchającego" gradientu. Przykładem metody normalizacji gradientu jest algorytm RMSprop (Pseudokod 4). Utrzymuje on normę gradientu w określonym zakresie, dzięki temu unikamy problemów związanych z bardzo małymi lub bardzo dużymi wartościami gradientu, które mogą utrudnić lub spowolnić proces uczenia. Aby zaimplementować RMSprop wprowadzamy parametr β – współczynnik wygaszania. Skuteczną wartością współczynnika β jest wartość *0,9*. Normalizacji gradientu dokonujemy poprzez kombinację liniową drugiego momentu funkcji g.

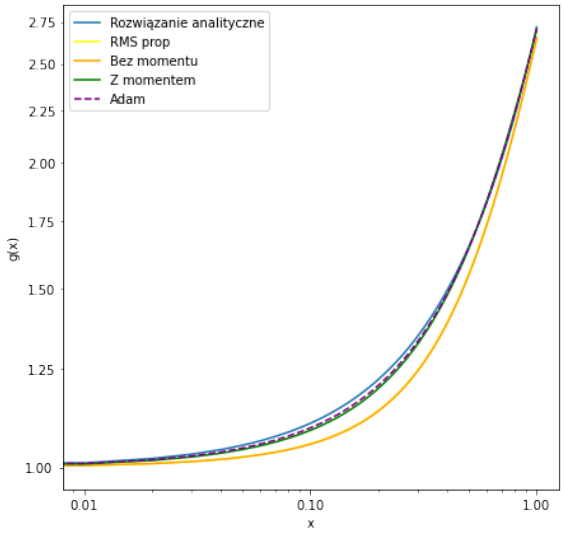
Pseudokod 4 algorytm RMSprop [4]

$\Theta \leftarrow$ InicjujLosowo;
$Eg^2 \leftarrow [0, 0, \ldots, 0]$;
while ¬StopCondition **do**
    $g \leftarrow 0$;
    for all $X \in WektorWejscia$ **do**
        $\hat{Y} \leftarrow g_t(\Theta, X)$;
        $g \leftarrow g + \nabla g_t(\Theta, X)$;
    end for
    $Eg^2 \leftarrow \beta Eg^2 + (1 - \beta)Eg^2$;
    $\Theta \leftarrow \Theta - \eta \cdot \frac{g}{\sqrt{Eg^2 + \epsilon}}$;
end while

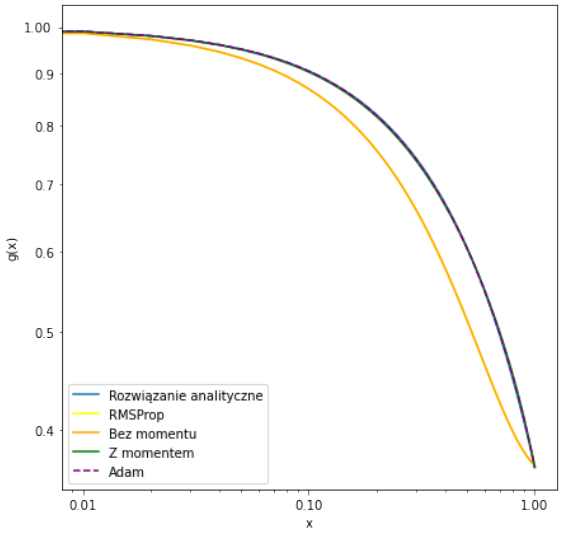
Możemy dalej połączyć implementację RMSprop oraz uczenia z momentem, otrzymując w ten sposób algorytm Adam *(ang. Adaptive moment estimation)*. Dzięki implementacji tego algorytmu w naszej sieci neuronowej unikamy wrażliwych sytuacji, które mogą wpłynąć na proces uczenia: eksplozji wag, utknięcia w minimum lokalnym, spowolnienie procesu uczenia oraz możemy przyspieszyć proces minimalizacji funkcji kosztu. Rezultaty dodania takich usprawnień sieci neuronowej przedstawiliśmy na rysnkach 6.3, 6.4 oraz w tabeli 6.2.

Sposób uczenia	Z momentem	RMSprop	Adam
Końcowy $C(x, \theta)$ na odcinku [0,1] (równ. 5.1):	0,01437	0,05050	$6{,}93 \cdot 10^{-3}$
Końcowy $C(x, \theta)$ na odcinku [0,1] (równ. 5.2):	0,00035	0,05076	$8{,}9 \cdot 10^{-5}$

Tabela 6.2: Porównanie wyników na wybranych metodach optymalizacyjnych

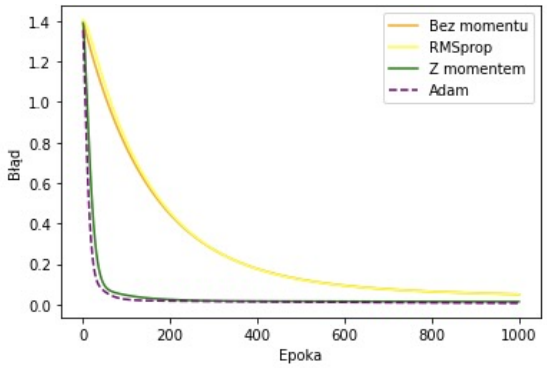


(a) Równanie (5.1) – aproksymacja rozwiązania

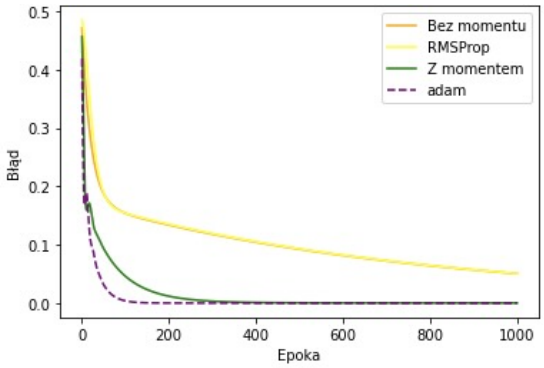


(b) Równanie (5.2) – aproksymacja rozwiązania

Rysunek 6.3: Wyniki przedstawiające proces uczenia sieci neuronowej dla metod Adam, RMSprop i Moment dla równań (5.1),(5.2) - dokładność aproksymacji w skali logarytmicznej



(a) Równanie (5.1) – błąd sieci (wartości funkcji kosztu) w zależności od epoki



(b) Równanie (5.2) – błąd sieci (wartości funkcji kosztu) w zależności od epoki

Rysunek 6.4: Wyniki przedstawiające proces uczenia sieci neuronowej dla metod Adam, RMSprop i Moment dla równań dla równań (5.1),(5.2) - wartość funkcji kosztu dla kolejnych epok

6.3. Metoda podziału zbioru uczącego *(ang. batching method)*

Kolejną implementacją przydatną do testowania rezulatów sieci neuronowej jest metoda podziału zbioru uczącego *(ang. batching)*. Polega ona na stopniowym uczeniu sieci poprzez dawkowanie mniejszej ilości danych w przeciwieństwie do trenowania wszystkich wartości z przedziału naraz. Dzięki temu, w każdej z epok, możemy uczyć sieć neuronową poprzez podział danych na na p częsci. Wówczas konieczne jest, aby w każdej z p z podziałów indeksy danych były wymieszane między sobą. Następnie trenujemy sieć neuronową nadając jej stopniowo w treningu 1-szą,2-gą,...p-tą część danych, gdzie w każdej iteracji zbiór wejściowy przyjmuje postacie X_k:

$$X_k = X^{(k)}, \quad k \in \{1, 2, \ldots, p\}.$$

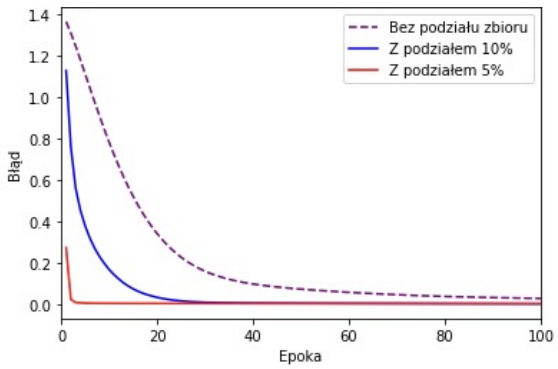
Takie podejście może być szczególnie pomocne, ponieważ możemy ustabilizować proces uczenia. Uśrednianie gradientów dla wielu przykładów w batchu pomaga w zmiejszaniu zmienności gradientów i może prowadzić do bardziej jednolitego procesu uczenia, co z kolei może pomóc w uniknięciu oscylacji funkcji kosztu. Dodatkowo zapobiegamy przetrenowania sieci, ponieważ model ma szansę zobaczyć różne przykłady z różnych części zestawu danych w każdym kroku uczenia, co może pomóc sieci w uchwyceniu różnorodności danych. W tym ekperymencie przedstawione zostały trzy sposoby podziału danych, przy podziale odcinka na 100 równych części:

- bez *batchingu* – 100% danych w jednym *batchu* (trenowanie sieci neuronowej pełnym zakresem danych naraz);
- podział na 10 *batchów* – 10% danych w jednym *batchu* (10 różnych zbiorów, z których każdy zawiera 10% różnych, wymieszanych między sobą danych z pełnego odcinka);
- podział na 5 *batchów* – 5% danych w jednym *batchu* (20 różnych zbiorów, z których każdy zawiera 5% różnych, wymieszanych między sobą danych z pełnego odcinka).

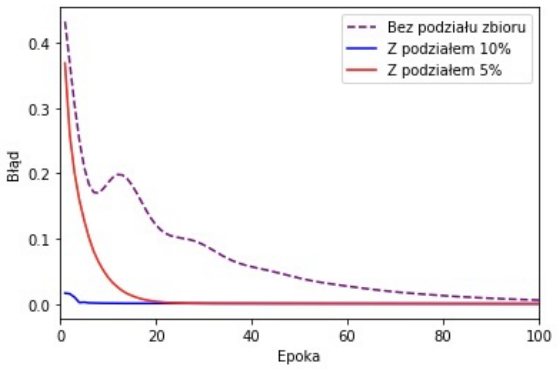
Na podstawie tabeli 6.3 i wykresu 6.5 można zaobserwować stabilizację procesu uczenia. W przypadku uczenia sieci neuronowej całym zakresem danych naraz w równaniu (5.2) sieć natrafiła na pewien szum, co udało się usunąć ucząć sieć zbiorami podzielonymi na 5% i 10% całości. Finalnie podziały zbioru uczącego na 10% i 5% cześci z odcinka w jednym *batchu* pozwoliły sieci neuronowej szybciej zbiec oraz zmniejszyć ostateczną wartość funkcji kosztu. W każdym z testów użyliśmy metody Adam z poprzedniej sekcji oraz liczby epok równej 1000, podczas gdy na rysunku 6.5 pokazaliśmy jedynie 100 pierwszych epok dla lepszej czytelności.

Zbiór	Bez batchingu	Batching 10%	Batching 5%
$C(x,\theta)$ na odcinku [0,1] (równ. 5.1):	$6{,}93 \cdot 10^{-3}$	$7{,}77 \cdot 10^{-6}$	$1{,}44 \cdot 10^{-5}$
$C(x,\theta)$ na odcinku [0,1] (równ. 5.2):	$8{,}9 \cdot 10^{-5}$	$1{,}57 \cdot 10^{-5}$	$2{,}57 \cdot 10^{-6}$

Tabela 6.3: Porównanie szybkości minimalizacji funkcji kosztu dla metody *batching*



(a) Równanie (5.1) – błąd sieci (wartości funkcji kosztu) w zależności od epoki



(b) Równanie (5.2) – błąd sieci (wartości funkcji kosztu) w zależności od epoki

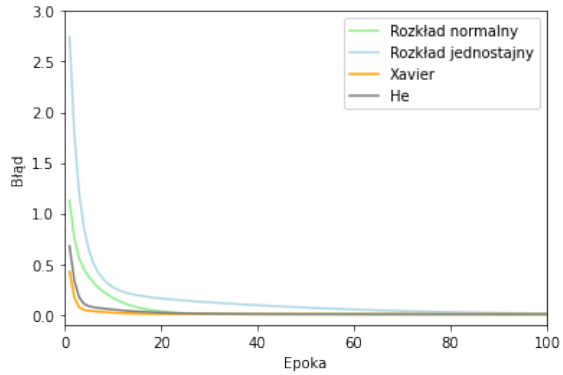
Rysunek 6.5: Porównanie wyników metody *batching* dla równań (5.1),(5.2) – wartość funkcji kosztu dla kolejnych epok

6.4. Sposób inicjowania wag

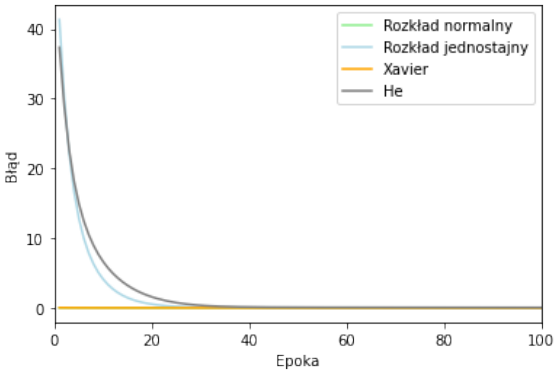
Testować będziemy również sposoby początkowego inicjowania wag dla każdej warstwy ukrytej W^l. Metoda inicjowania wag może wpływać na szybkość zbieżności oraz czas uczenia, a także na końcową wartość $C(x,\theta)$. Wyróżnimy cztery sposoby inicjowania wag:

- Rozkład normalny na przedziale [0,1]: $W_{ij} \sim \mathcal{N}(0,1)$;
- Rozkład jednostajny na przedziale [0,1]: $W_{ij} \sim U(0,1)$;
- Inicjacja Xavier: $W_{ij} \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n+m}}\right)$;
- Inicjacja He: $W_{ij} \sim \mathcal{N}\left(0, \sqrt{\frac{6}{n+m}}\right)$.

Z poprzednich sekcji pracy zauważyliśmy, że metodami, które najefektywniej wpłynęły na proces uczenia sieci neuronowej, obu z rozpatrywanych równań (5.1), (5.2), były algorytm Adam oraz podział zbioru uczącego na 10%/5% danych metodą *batching*, zatem dalej będziemy rozpatrywać tylko sieci o takiej architekturze. Jako ziarno generatora niezmiennie ustawiliśmy 15.



(a) Równanie (5.1) – błąd sieci (wartości funkcji kosztu) w zależności od epoki



(b) Równanie (5.2) – błąd sieci (wartości funkcji kosztu) w zależności od epoki

Rysunek 6.6: Porównanie różnych sposobów inicjowania wag

Na rysunku 6.6 widzimy, że dobór sposobu inicjowania wag również wpływa na proces uczenia. Przykładowo dla zwykłego rozkładu jednostajnego proces ten przebiega najgorzej, a dla metod takich jak He lub Xavier parametry sieci wpływają na znacznie mniejszą wartość funkcji kosztu już od pierwszych iteracji. Może to być kolejna praktyczna metoda do obniżenia liczby epok.

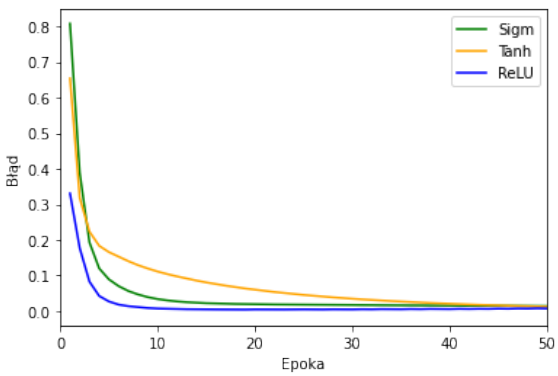
6.5. Różne funkcje aktywacji i próby

Wspomnieliśmy, że testować będziemy również różne funkcje aktywacji spośród tangens hiperboliczny, ReLU oraz sigmoidalną. Wówczas uwzględniać będziemy, we wzorze na $delta^l$ różne pochodne tych funkcji, które po wyprowadzeniu przyjmują postać:

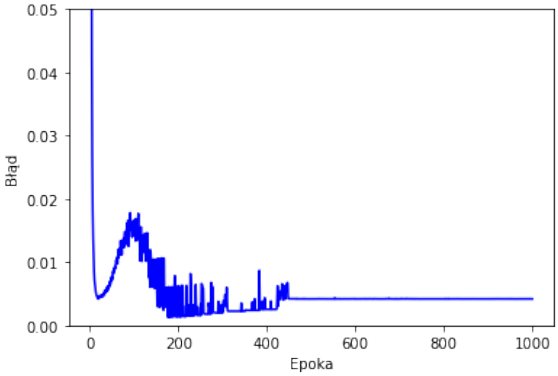
- Dla funkcji sigmoidalnej: $\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$;
- Dla funkcji tanh: $\sigma'(x) = 1 - \tanh^2(x)$;
- Dla funkcji ReLU: $\sigma'(x) = \begin{cases} 1 & \text{jeśli } x > 0, \\ 0 & \text{jeśli } x \leqslant 0. \end{cases}$

Na rysnku 6.7 możemy zauważyć jak każda z nich poradziła sobie z minimalizacją funkcji kosztu dla równania (5.1). Funkcja tangens hiporboliczny okazała się być najwolniejsza podczas gdy sigmoidalna i funkcja ReLU były porównywalnie podobne na przełomie epok od 0 do 50. Mimo iż funkcja ReLU była szybsza od sigmoidalnej, w szczególności na początku działania procesu uczenia, to na rysunku z prawej strony widzimy, że funkcja ta od pewnego momentu zaczyna się destabilizować i przybierać pewien szum. Z powodu, iż funkcja sigmoidalna w każdym z po-

przednich testów stabilnie minimalizowała funkcję kosztu i jest podstawą Teorii Uniwersalnej Aproksymacji uznaliśmy ją jako lepszy wybór do finalnych testów.



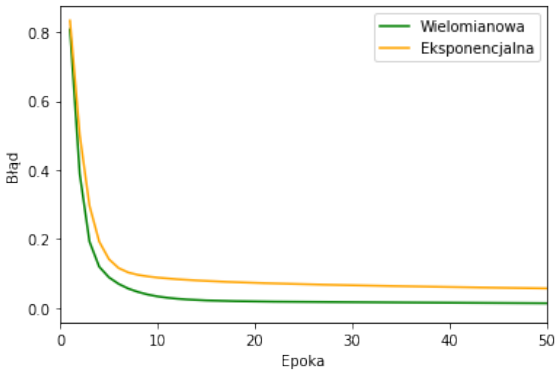
(a) Porównanie trzech funkcji aktywacji na jednym wykresie na przełomie epok 0-50



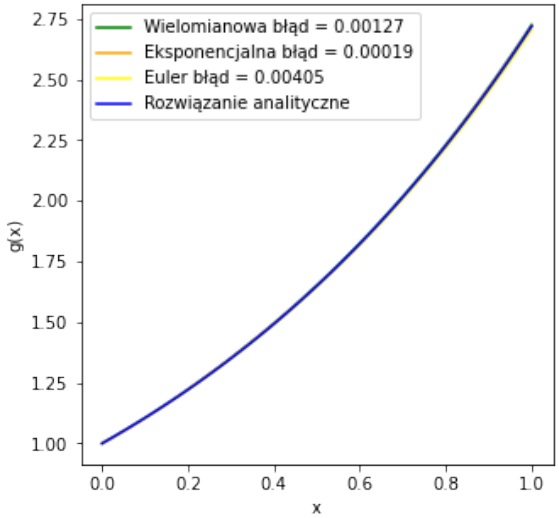
(b) Funkcja aktywacji ReLU na przełomie epok 0-1000 na wybranym zakresie błędu 0-0.05

Rysunek 6.7: Porównanie różnych funkcji aktywacji

Następnie przetestowaliśmy jaki wpływ na wynik ma zmiana funkcji próby z wielomianowej na eskponencjalną. Choć funkcja wielomianowa szybciej pomaga zbiec funkcji kosztu do minimum, to przy dokładności aproksymacji funkcja eksponencjalna dała lepszy wynik, co możemy zobaczyć na rysunku 6.8.



(a) Porównanie funkcji wielomianowej i eskponencjalnej



(b) Dokładnosć aproksymacji (błąd MSE)

Rysunek 6.8: Porównanie funkcji próby wielomianowej i eskponencjalnej

7. Rozpatrywanie wybranych układów równań różniczkowych

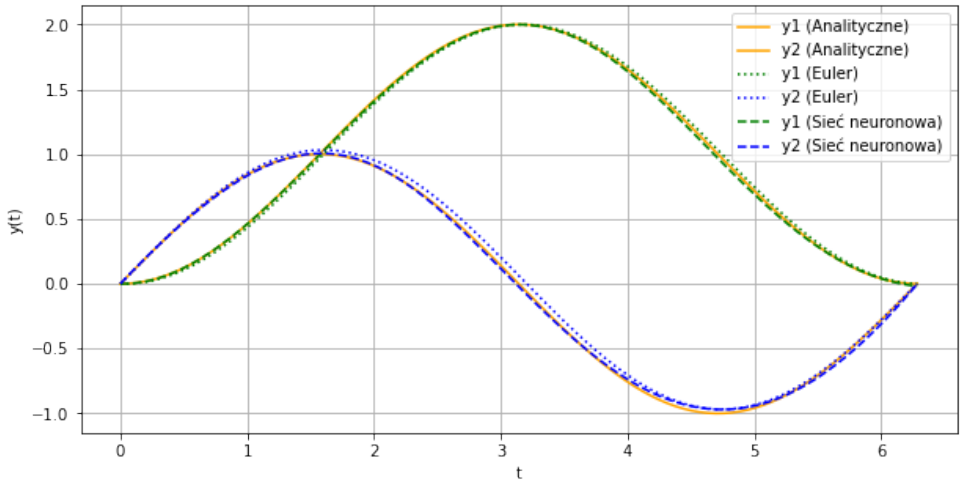
Po dotychczasowych eksperymentach widzimy, że sieci neuronowe skutecznie potrafią aproksymować wyniki naszych wybranych równań różniczkowych pierwszego rzędu. Oprócz podstawowych równań przetestujemy sieć neuronową na następującym prostym układzie równań:

$$\begin{cases} \frac{dy_1}{dt} = \sin(t), \\ \frac{dy_2}{dt} = \cos(t), \end{cases} \tag{7.1}$$

gdzie $y_1(t)$ i $y_2(t)$ są funkcjami czasu t. W takim układzie, rozwiązania tych równań będą miały postać:

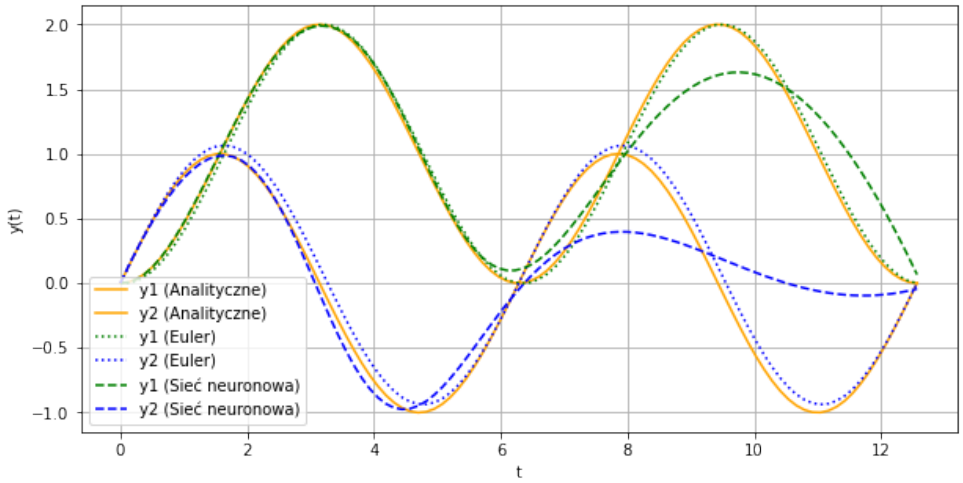
$$y_1(t) = -\cos(t) + C_1, \qquad y_2(t) = \sin(t) + C_2,$$

gdzie C_1 i C_2 są stałymi zależnymi od warunków początkowych. Jako warunki początkowe przyjmiemy $y_1(0) = 0$ oraz $y_2(0) = 0$. Rozwiązanie powyższego układu przeprowadziliśmy z wykorzystaniem algorytmu Adam, podziału zbioru na 5 *batchów* zawierających po 20% danych oraz przy 50 neuronach w jednej warstwie ukrytej. Do wytrenowania sieci potrzebne było 1000 epok. Metoda Eulera została policzona z krokiem podziału odcinka na 100 części, a sieć neruonowa z próbką 100 danych z badanego zakresu. Na rysunku 7.1 przedstawiliśmy dokładność aproksymacji rozwiązania dla sieci neuronowej w porównaniu z metodą Eulera.

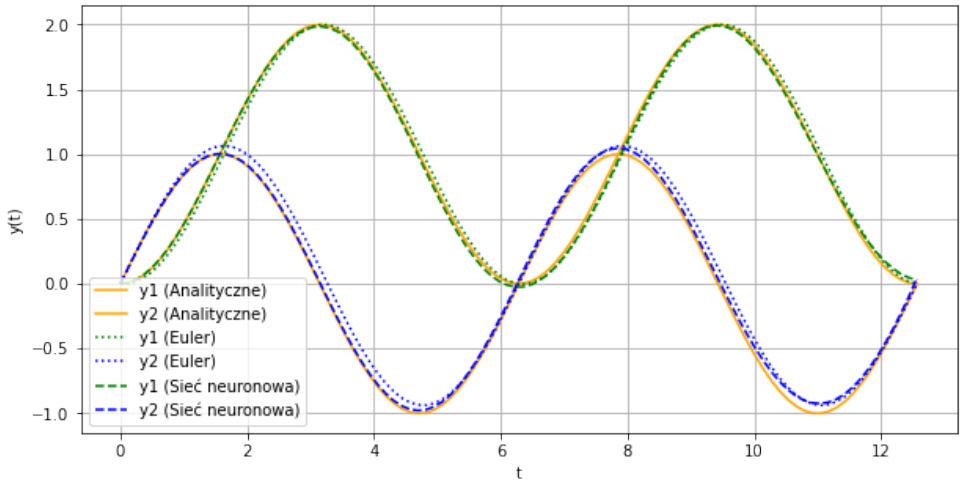


Rysunek 7.1: Aproksymacja rozwiązań metodą Eulera i siecią neruonową na odcinku $[0, 2\pi]$

Następnie sprawdziliśmy działanie sieci neuronowej na odcinku $[0, 4\pi]$ z liczbą epok odpowiednio 1000 oraz 10000. Poniżej rysunki 7.2 i 7.3 oraz tabela 7.1 przedstawiają rezultaty.



Rysunek 7.2: Aproksymacja rozwiązań z siecią neruonową 1000 epok dla odcinka $[0, 4\pi]$



Rysunek 7.3: Aproksymacja rozwiązań z siecią neuronową 10000 epok dla odcinka $[0, 4\pi]$

Wybrana metoda	Wynik dla rozwiązania y_1	Wynik dla rozwiązania y_2
Metoda Eulera dla $[0, 2\pi]$	0,0321	0,0634
Sieć Neuronowa dla $[0, 2\pi]$ (1 tys. epok)	**0,0316**	**0,0339**
Metoda Eulera dla $[0, 4\pi]$	0,0648	0,1269
Sieć Neuronowa dla $[0, 4\pi]$ (1 tys. epok)	0,4955	0,9448
Sieć Neuronowa dla $[0, 4\pi]$ (10 tys. epok)	**0,0477**	**0,0785**

Tabela 7.1: Wyniki średniego błędu kwadratowego sieci neuronowej dla układu równań 7.1

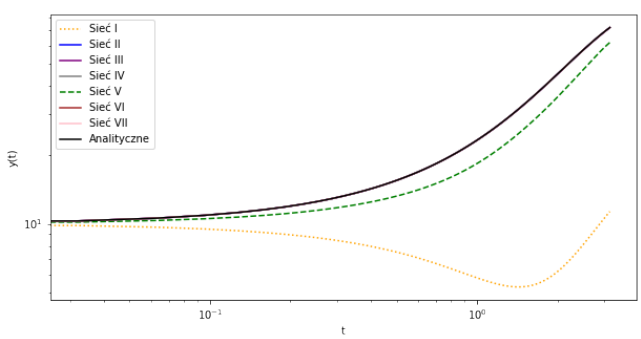
8. Rezultaty i wizualizacje najlepszych metod

Do ostatecznego zestawienia podjęliśmy 3 równania i jeden układ równań tj. równanie (5.1) na odcinku [0,1], równanie (5.2) na odcinku [0,2], układ równań (7.1) na odcinku $[0, 2\pi]$ i równanie nieliniowe postaci (8.1) na odcinku [0,1] a potem $[0, \pi]$.

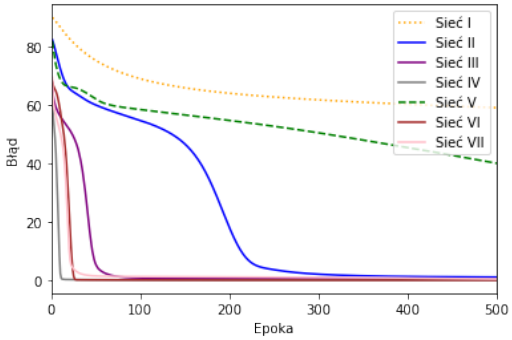
$$y' = y \cdot \left(1 - \frac{y}{100}\right), \qquad y(0) = 10. \tag{8.1}$$

Przetestowaliśmy 7 różnych architektur sieci neuronowych w porównaniu z metodą Rungego-Kutty 4. rzędu oraz metodą Eulera. W przypadku każdego badania odcinek podzieliliśmy na 100 części, przy czym w metodzie Rungego-Kutty krok podziału zaokrąglaliśmy do piątego miejsca po przecinku. Finalne rezultaty widzimy na rysunku 8.2. Sieci neuronowe jakie wybraliśmy do testowania na podstawie poprzednich rozdziałów i umieściliśmy je w tabeli to odpowednio:

- Sieć neuronowa I: jedna warstwa ukryta [10], Adam, bez metody *batching*, wielomianowa funkcja próby, inicjowanie z rozkładu jednostajnego normalnego, metoda stochastycznego spadku wzdłuż gradientu, 1000 epok;
- Sieć neuronowa II: jedna warstwa ukryta [50], Adam, bez metody *batching*, wielomianowa funkcja próby, inicjowanie z rozkładu jednostajnego normalnego, metoda stochastycznego spadku wzdłuż gradientu, 10000 epok;
- Sieć neuronowa III: jedna warstwa ukryta [50], Adam, *batching 10%*, wielomianowa funkcja próby, inicjowanie z rozkładu jednostajnego normalnego, 1000 epok;
- Sieć neuronowa IV: dwie warstwy ukryte [50, 50], Adam, *batching 10%*, wielomianowa funkcja próby, inicjowanie z rozkładu jednostajnego normalnego, 10000 epok;
- Sieć neuronowa V: trzy warstwy ukryte [10, 10, 10], Adam, *batching 10%*, eksponencjalna funkcja próby, inicjowanie z rozkładu Xavier, 1000 epok;
- Sieć neuronowa VI: trzy warstwy ukryte [50, 50, 50], Adam, *batching 20%*, eksponencjalna funkcja próby, inicjowanie z rozkładu He, 1000 epok;
- Sieć neuronowa VII: jedna warstwa ukryta [100], Adam, *batching 10%*, eksponencjalna funkcja próby, inicjowanie z rozkładu He, 10000 epok.



(a) Wyniki aproksymacji w skali logarytmicznej



(b) Porównanie procesu uczenia

Rysunek 8.1: Rezultaty testowanych sieci neuronowych dla równania (8.1), na odcinku $[0, \pi]$

Metoda:	Równ. (5.1)	Równ. (5.2)	Układ (7.1) y1	Układ (7.1) y2	Równ. (8.1), [0,1]	Równ. (8.1), [0,π]
Euler	0.013603	0.001865	0.064822	0.126944	0.061190	0.405645
Runge-Kutty	0.000272	0.000038	0.000432	0.009561	0.001781	0.006598
Sieć neuronowa I	0.013422	0.003876	1.216930	1.357860	0.261772	60.68285
Sieć neuronowa II	0.0001311	0.000206	0.017044	0.039913	0.000470	0.006875
Sieć neuronowa III	0.003365	0.000193	0.624004	0.626647	0.001643	0.233559
Sieć neuronowa IV	0.000065	0.000303	0.075849	0.048979	0.016996	0.006901
Sieć neuronowa V	0.000811	0.001379	2.014780	1.133140	0.005990	10.41479
Sieć neuronowa VI	0.003000	0.002473	0.094092	0.224929	0.007376	0.020767
Sieć neuronowa VII	0.000176	0.000047	0.023170	0.184630	0.005841	0.002735

Metoda:	Równ. (5.1)	Równ. (5.2)	Układ (7.1) y1	Układ (7.1) y2	Równ. (8.1), [0,1]	Równ. (8.1), [0,π]
Euler	0.013603	0.001865	0.064822	0.126944	0.061190	0.405645
Runge-Kutty	0.000272	0.000038	0.000432	0.009561	0.001781	0.006598
Sieć neuronowa I	0.013422	0.003876	1.216930	1.357860	0.261772	60.682850
Sieć neuronowa II	0.000131	0.000206	0.017044	0.039913	0.000470	0.006875
Sieć neuronowa III	0.003365	0.000193	0.624004	0.626647	0.001643	0.233559
Sieć neuronowa IV	0.000065	0.000303	0.075849	0.048979	0.016996	0.006901
Sieć neuronowa V	0.000811	0.001379	2.014780	1.133140	0.005990	10.414790
Sieć neuronowa VI	0.003000	0.002473	0.094092	0.224929	0.007376	0.020767
Sieć neuronowa VII	0.000176	0.000047	0.023170	0.184630	0.005841	0.002735

1. miejsce 2. miejsce 3. miejsce 4. miejsce 9. miejsce

Rysunek 8.2: Rezultaty badań i porównanie metod zestawione w tabeli oraz w mapie ciepła (miara średniego błędu kwadratowego pomiędzy aproksymacją i rozwiązaniem analitycznym).

Na rysunku 8.1 możemy zaobserwować, że sieci neuronowe I oraz V nie poradziły sobie z równaniem nieliniowym i ich aproksymacje są dalekie od rozwiązania analitycznego. W obu przypadkach dysponowały one jedynie liczbą 10 neuronów, w jednej bądź trzech warstwach, co okazało się być zbyt małą liczbą do zamodelowania bardziej złożonego problemu przy liczbie epok 1000. Również różnicę widzimy gdy porównamy między sobą procesy uczenia wszystkich sieci. Sieci I oraz V minimalizują funkcję $C(x, \theta)$ bardzo powoli, co przekłada się na ich wyniki. Z kolei architektury z większą liczbą neuronów modelują równanie nieliniowe bardzo skutecznie, a nawet dają lepsze rezultaty od zaimplementowanych metod numerycznych. Najlepsze do rozwiązania równania nieliniowego, również lepsze od algorytmu Rungego-Kutty, okazały się być sieci II oraz VII, a więc te z jedną warstwą ukrytą z dużą liczbą neuronów. Ciekawą obserwacją jest proces minimalizacji funkcji kosztu dla sieci II, który choć na początku przebiega powoli znajduje finalnie najlepsze rozwiązanie ze wszystkich. Oznacza to zatem, że bardzo strome spadki funkcji kosztu niekoniecznie zawsze oznaczają najlepsze dobory wag w połączeniach neuronów, a co za tym idzie końcowe aproksymacje rozwiązań.

Spoglądając na rysunek 8.2 możemy zauważyć, że dostaliśmy bardzo zróżnicowane wyniki. Przykładowo prawie wszystkie sieci neuronowe, z wyjątkiem sieci I okazały się być dobrymi metodami aproksymacji równań (5.1) i (5.2) przewyższając w tych przypadkach numeryczną metodę Eulera i doganiając swoimi wynikami metodę Rungego-Kutty. Metody numeryczne, z drugiej strony, lepiej poradziły sobie z przybliżaniem rozwiązania naszego przykładu prostego układu równań. Sieci neuronowe dały w tym przypadku znacznie słabsze wyniki, a jedynie ponownie sieć II z 50 neuronami osiągnęła dobry wynik dla obu rozwiązań y_1 i y_2. Ostatecznie pod względem 1. miejsca zaszedł remis – sieci neuronowe (II, IV, VII) i metoda Rungego-Kutty czwartego rzędu były najlepsze w dokładnie trzech przypadkach. „Uniwersalnymi" sieciami neuronowymi okazały się być II oraz VII otrzymując za każdym razem miejsca na lub blisko podium, jednak najlepiej jest dobierać odpowiednie parametry i architektury indywidualnie pod każde zagadnienie, bo jak widzimy drobne zmiany potrafią bardzo przybliżyć lub bardzo oddalić nas od odpowiedniego rozwiązania.

9. Podsumowanie

Rezultaty zaimplementowanych sieci neuronowych udowodniły skuteczność sieci neuronowych jako uniwersalnych aproksymatorów. Możemy zaobserwować, że eksperymenty na różnych typach równań dały nam zróżnicowane wyniki i pokazały, że sieci potrafią być często równie skuteczne jak metoda Eulera lub Rungego-Kutty. Ponadto warto zauważyć, że najlepszymi architekturami okazały się być te o tylko jednej warstwie ukrytej z dużą liczbą neuronów w niej – sieć neuronowa II oraz VII – co potwierdza teorytyczne podstawy matematyczne z Teorii Uniwersalnej Aproksymacji. Najsłabszymi okazały się być sieci o małej liczbie neuronów w jednej lub kilku warstwach ukrytych. Wnioskujemy również, że stosowanie jednej warstwy ukrytej oraz ekperymentowanie z większymi liczbami neuronów niż 100 mogłoby przynieść jeszcze lepsze wyniki, lecz należy również brać pod uwagę odpowiedni dobór parametrów do każdego rozpatrywanego równania. Tradycyjne metody numeryczne, takie jak metoda Eulera czy Rungego-Kutty, mimo swojej skuteczności, mają ograniczenia związane z złożonością obliczeniową i trudnością w stosowaniu ich do złożonych problemów o wysokim wymiarze. Sieci neuronowe, z drugiej strony, oferują elastyczność i zdolność do modelowania skomplikowanych zależności nieliniowych, co czyni je efektywnym i przyszłościowym narzędziem do aproksymacji rozwiązań równań różniczkowych zwyczajnych.

A. Implementacja sieci neuronowej

```
import numpy as np
import matplotlib.pyplot as plt

class MultiLayers(object):
    def __init__(self, input_size, layer_sizes, output_size,
                 weight_init="normal", activation="sigm", exit="linear",
                     measure="ode",
                 momentum = False, RMSprop=False, normalization="none",
                 ode_function = None, g0 = None):
        ## ______ Initialization ______ ##
        # 0) layers and architecture init
        np.random.seed(50)
        layers = [input_size] + layer_sizes + [output_size]
        IN = layers[:-1]
        OUT = layers[1:]
        self.ode_function = ode_function
        if momentum==True: self.momentum = True
        else: self.momentum = False
        if RMSprop==True: self.RMSprop = True
        else: self.RMSprop = False
        if normalization=="none": self.normalization = None
        if normalization=="std": self.normalization = "std"
        if normalization=="minmax": self.normalization = "minmax"
        # 1) __ wieghts init
        if weight_init == "normal":
            def initializer(a,b): return npr.randn(a,b)
        elif weight_init == "he":
            def initializer(a,b): return np.random.normal(0,np.sqrt(2/(a+b)),
                size=(a,b))
        elif weight_init=="xavier":
            def initializer(a,b): return np.random.normal(0,np.sqrt(6/2*(a+b)),
                size=(a,b))
        elif weight_init=="uniform":
            def initializer(a,b): return np.random.uniform(0,1, size=(a,b))
```

```
        # 2) __ Loss function
        self.measure = "ode"
        if g0!=None: self.g0 = g0
        # 3) __ Activation function
        if activation=="relu":
            self.activation = self.relu
            self.derivative = self.relu_derivative
        elif activation=="tanh":
            self.activation = self.tanh
            self.derivative = self.tanh_derivative
        elif activation=="linear":
            self.activation = self.linear
            self.derivative = self.linear_derivative
        elif activation=="sigm":
            self.activation = self.sigmoid
            self.derivative = self.sigmoid_derivative
        # 4) __ exit activation
        if self.normalization==None: self.exit = self.linear
        if self.normalization=="std": self.exit = self.denormalize_std
        if self.normalization=="minmax": self.exit = self.denormalize_minmax

        # 5) ______ weights and biases initialation ______
        self.weights = [initializer(input,output) for (input,output) in
            zip(IN,OUT)]
        self.biases = [np.zeros(output) for output in OUT]
        # 6) _______ arrays for neurons calculating _______
        self.summed = [None]*len(self.weights) # _______ calculating linear
            combinations on a single neuron
        self.activated = [None]*len(self.weights) # _______ calculated
            activation on linear combinations
        # 7) _______ Momentum - velocity _________
        self.V_biases = [np.zeros(b.shape) for b in self.biases]
        self.V_weights = [np.zeros(w.shape) for w in self.weights]
        # 8) _______ Means for RMSprop  _________
        self.M_biases = [np.zeros(b.shape) for b in self.biases]
        self.M_weigths = [np.zeros(w.shape) for w in self.weights]

        # 9) __ arrays for errors
        self.errors_array = []
        self.error_matrix=[]
        self.epoch_n=[]
        self.scores = []
```

```
71
72      #####
73      ## ______ Activation Formulas ______ ##
74      def sigmoid(self, x):
75          return 1 / (1 + np.exp(-x))
76      def relu(self, x):
77          return np.maximum(0, x)
78      def tanh(self, x):
79          return np.tanh(x)
80      def linear(self, x):
81          return x
82      # 1) __ activation - derivative formulas
83      def relu_derivative(self,x):
84          return np.where(x > 0, 1, 0)
85      def tanh_derivative(self,x):
86          return 1 - np.tanh(x)**2
87      def sigmoid_derivative(self,x):
88          return self.sigmoid(x) * (1 - self.sigmoid(x))
89      def linear_derivative(self,x):
90          return 1
91      ######
92
93  ## ______ Fit data  ______ ##
94  ##
95  ##
96      def forward(self, x, params):
97  # 1) _______ Forward propagation _______
98          if self.normalization=="std": layer=self.normalize_std(x)
99          if self.normalization=="minmax": layer=self.normalize_minmax(x)
100         self.weights,self.biases = params[0],params[1]
101         layer = x
102         n=len(self.weights)
103         self.summed[0] = layer    # ____ first layer
104
105         for i in range(n-1): # ____ go over all layers except first/last
106             self.activated[i] = layer
107             layer = layer * self.weights[i] + self.biases[i]
108             self.summed[i+1] = layer
109             layer = self.activation(layer)
110
111         self.activated[-1]=layer    # ____ last layer
112         output_layer = np.dot(layer,self.weights[-1]) + self.biases[-1]
```

```
113         output = self.exit(output_layer)
114         return output
115 
116     def backward(self,y,y_predicted, optimization_method):
117 # 2) _______ backward algorithm with gradients _______ #
118         gradient_biases, gradient_weights = self.gradients(y, y_predicted)
119         gradient_biases, gradient_weights =
                optimization_method(gradient_biases, gradient_weights)
120         self.weights = [w- self.learn*g for (w,g) in zip(self.weights,
                gradient_weights)]
121         self.biases = [b- self.learn*g for (b,g) in
                zip(self.biases,gradient_biases)]
122 
123 # 3) _______ Train data _______ #
124     def train(self, x_train, y_train, epochs, batch_size=None, batching=False
            ,learning_rate=0.001):
125         n=len(x_train)
126         self.epochs= epochs
127         if batch_size==None: batch_size=n
128         self.x_train = x_train
129         # ____ setting network optimization
130         self.learn=learning_rate
131         if self.normalization=="std": self.set_normals(x_train,y_train)
132         if self.momentum==True and self.RMSprop==True: optimization=self.Adam
133         if self.momentum==True and self.RMSprop==False:
                optimization=self.Momentum_fun
134         if self.momentum==False and self.RMSprop==True:
                optimization=self.RMSprop_fun
135         if self.momentum==False and self.RMSprop==False:
                optimization=self.no_opt
136 
137         for epoch in range(epochs):
138             if epoch==0: print("Initial error = ", self.ode_loss(x_train))
139             indeces = list(range(n))
140             if batching: np.random.shuffle(indeces) # shuffling indeces
141 
142             # __________ getting new predictions in new epoch for all branches
                    __________
143             for k in range(0,n,batch_size):
144                 part_in_use = indeces[k:k+batch_size]
145                 x,y = x_train[part_in_use], y_train[part_in_use]
146                 #~~~~~
```

```
147                 y_predicted = self.forward(x, [self.weights,self.biases])
148                 self.backward(y,y_predicted, optimization_method = optimization)
149                 #~~~~~
150 
151             # __________ saving outputs for showing errors __________
152             if self.measure == "ode": self.scores.append(self.ode_loss(x_train))
153             if epoch in range(1, epochs+1, int(epochs/9)) or epoch==epochs:
154                 self.epoch_n.append(epoch)
155                 if self.measure == "ode": print_error = self.ode_loss(x_train)
156                 self.errors_array.append(print_error)
157                 self.error_matrix.append(self.output_delta)
158         if self.measure == "ode": print(f'Epoch {epoch+1}, Training MSE Score
                for ODE: {print_error}\n')
159         return print_error
160 
161 # 4) _____ Gradient implementation ______ ##
162     def gradients(self, y, y_predicted):
163         n = len(self.weights)
164         gradient_biases = [np.zeros(b.shape) for b in self.biases]
165         gradient_weights = [np.zeros(w.shape) for w in self.weights]
166         if self.measure == "ode":
167             self.output_delta = self.ode_loss_derivative(self.x_train)
168             for i in range(1, n+1):
169                 gradient_biases[-i] = np.sum(self.output_delta, axis=0)
170                 gradient_weights = [np.dot(self.activated[-i].T,
                        self.output_delta)]
171                 self.output_delta = np.dot(self.output_delta,
                        self.weights[-i].T) * self.derivative(self.summed[-i])
172         return gradient_biases,gradient_weights
173 
174 ## ______ Improvements - direction funs ______ ##
175     # 0) ___ normalization of data
176     def set_normals(self,x,y):
177         self.meanx = np.mean(x)
178         self.stdx = np.std(x)
179         self.meany = np.mean(y)
180         self.stdy = np.std(y)
181     def normalize_std(self,x):
182         return (x-self.meanx)/self.stdx
183     def denormalize_std(self,y_pred):
184         return y_pred*self.stdy + self.meany
185     def normalize_minmax(self,x):
```

```
186         return (x-min(x))/(max(x)-min(x))
187     def denormalize_minmax(self,y_pred):
188         return y_pred*(max(y_pred)-min(y_pred)) + min(y_pred)
189     ## 1) ___ improvements - Momentum
190     def Momentum_fun(self,G_biases,G_weights, beta = 0.9):
191         self.V_biases = [beta*vb+gb for (vb,gb) in zip(self.V_biases, G_biases)]
192         self.V_weights = [beta*vw+gw for (vw,gw) in zip(self.V_weights,
                G_weights)]
193         return self.V_biases, self.V_weights
194     ## 2) ___ improvements - RMSprop
195     def RMSprop_fun(self,G_biases,G_weights,beta = 0.9):
196         self.M_biases = [beta*mb + (1-beta)*np.square(gb) for (mb,gb) in
                zip(self.M_biases,G_biases)]
197         self.M_weigths = [beta*mw + (1-beta)*np.square(gw) for (mw,gw) in
                zip(self.M_weigths,G_weights)]
198         # actualization of g function
199         eps=1e-15
200         G_biases = [np.divide(gb, np.sqrt(mb)+eps) for (mb,gb) in
                zip(self.M_biases, G_biases)]
201         G_weights = [np.divide(gw, np.sqrt(mw)+eps) for (mw,gw) in
                zip(self.M_weigths,G_weights)]
202         return G_biases, G_weights
203     ## 3) ___ Adam - combined RMSprop, momemntum
204     def Adam(self, G_biases,G_weights):
205         beta1 = 0.9
206         beta2= 0.999
207         eps=1e-15
208         self.V_biases = [beta1*vb + (1-beta1)*gb for (vb,gb) in
                zip(self.V_biases,G_biases)]
209         self.V_weights = [beta1*vw + (1-beta1)*gw for (vw,gw) in
                zip(self.V_weights,G_weights)]
210         self.M_biases = [beta2*mb + (1-beta2)*np.square(gb) for (mb,gb) in
                zip(self.M_biases,G_biases)]
211         self.M_weigths = [beta2*mw + (1-beta2)*np.square(gw) for (mw,gw) in
                zip(self.M_weigths,G_weights)]
212         G_weights = [np.divide(vw/(1-beta1),np.sqrt(mw/(1-beta2))+eps) for
                (mw,vw) in zip(self.M_weigths,self.V_weights)]
213         G_biases = [np.divide(vb/(1-beta1),np.sqrt(mb/(1-beta2))+eps) for
                (mb,vb) in zip(self.M_biases,self.V_biases)]
214         return G_biases,G_weights
215     def no_opt(self, a,b):
216         return a,b
```

```
217 ## ______ Visualizations ______ ##
218     def show_error(self):
219         plt.plot([i for i in range(1, self.epochs, int(self.epochs/9))],
                self.errors_array)
220         plt.xlabel('Epoka')
221         plt.ylabel('Blad')
222         plt.title('Blad w zaleznosci od Epoch')
223         plt.show()
224     def show_matrix_error(self):
225         fig, axs = plt.subplots(3,3, figsize=(10, 10))
226         k=0
227         for i in range(3):
228             for j in range(3):
229                 ax = axs[i, j]
230                 error_values = self.error_matrix[k]
231                 im = ax.imshow(error_values.reshape(1, -1), cmap='Blues',
                        aspect='auto', vmin=0, vmax=0.2)
232                 ax.set_title(f'Epoka {self.epoch_n[k]+1}')
233                 k+=1
234         cbar = fig.colorbar(im, ax=axs, orientation='vertical', fraction=0.03,
                pad=0.1)
235         cbar.set_label('Error')
236         plt.show()
237
238 ## ______ ODE loss function ______ ##
239
240     def my_grad(self, f, argnum=0):
241         def gradient(*args):
242             def fn(x):
243                 args_ = args[:argnum] + (x,) + args[argnum + 1:]
244                 return f(*args_)
245             eps = 1e-6
246             x = args[argnum]
247             grad_x = []
248
249             for i in range(len(x)):
250                 x_plus = x.copy()
251                 x_plus[i] += eps
252                 x_minus = x.copy()
253                 x_minus[i] -= eps
254                 grad_x.append((fn(x_plus) - fn(x_minus)) / (2 * eps))
255
```

```
256             result_reshaped = [x for row in np.array(grad_x) for x in row if x
                    != 0]
257             return result_reshaped
258         return gradient
259
260     def ode_loss(self, x):
261         params = [self.weights, self.biases]
262         def g_trial(x,params):
263             neuralnetwork = self.forward(x, params)
264             return self.g0 + x*neuralnetwork
265         g_t = g_trial(x,params)
266         d_g_t = self.my_grad(g_trial,0)
267         d_g_t = d_g_t(x,params)
268         predicted_function = self.ode_function(g_t)
269         error = np.square(d_g_t-predicted_function)
270         return np.sum(error) / np.size(error)
271
272     def ode_loss_derivative(self, x):
273         params = [self.weights, self.biases]
274
275         def C(x, params):
276             def neuralnetwork(params):
277                 return self.forward(x, params)
278             def g_trial(x,params):
279                 return self.g0 + x*neuralnetwork(params)
280             g_t = g_trial(x,params)
281             d_g_t = self.my_grad(g_trial,0)(x,params)
282             predicted_function = self.ode_function(g_t)
283             return (2*(d_g_t-predicted_function))
284         return C(x,params)
285
286     def ode_solution(self,x):
287         return self.g0 + x*self.forward(x, [self.weights, self.biases])
```

Bibliografia

[1] Coryn A.L. Bailer-Jones, R. Gupta i Harinder P. Singh. *An Introduction to Artificial Neural Networks*. 2001.

[2] J. Banasiak i K. Szymańska-Dębowska. *Układy dynamiczne w modelowaniu procesów przyrodniczych, społecznych, technologicznych*. 2023.

[3] G. Cybenko. "Approximation by Superpositions of a Sigmoidal Function". W: *Mathematics of Control, Signals, and Systems* (1989).

[4] A. P. Engelbrecht. *Computational Intelligence: An Introduction*. 2007.

[5] Z. Fortuna, B. Macukow i J. Wąsowski. *Metody numeryczne*. 2024.

[6] J. Karwowski. *Wykłady z metod inteligencji komputerowych w analizie danych*.

[7] M. Kulczycki i K. Wójcik. *Matematyczne modele rozprzestrzeniania się chorób zakaźnych*. Uniwersytet Jagielloński w Krakowie, 2014.

[8] G. W. Leibniz. "Nova Methodus pro Maximis et Minimis". W: *Acta Eruditorum* (1684).

[9] M. Matyka. *Symulacje komputerowe w fizyce*. 2020.

[10] A. Palczewski. *Równania różniczkowe zwyczajne*. 2017.

[11] G. Świątek. *Wykłady z równań różniczkowych zwyczajnych*.

Wykaz symboli i skrótów

$A < B$	przestrzeń A jest podprzestrzenią przestrzeni B
*	operator gwiazdka
bias	wyraz wolny
batching	metoda tworzenia mniejszych podziałów zbioru
forward	propagacja wprzód
backward	propagacja w tył

Spis rysunków

Spis tabel

Spis załączników