Politechnika Wrocławska

**Wydział Matematyki**
Kierunek studiów: Matematyka stosowana
Specjalność: –

Praca dyplomowa – inżynierska

# SIECI NEURONOWE W ROZWIĄZYWANIU RÓWNAŃ RÓŻNICZKOWYCH ZWYCZAJNYCH

Szymon Stano

słowa kluczowe:
sieci neuronowe, perceptron wielowarstwowy, funkcja straty, automatyczne różniczkowanie, równania różniczkowe zwyczajne

krótkie streszczenie:
W pracy zaprezentowano zastosowanie sieci neuronowych do aproksymacji rozwiązań równań różniczkowych zwyczajnych. Omówiony został perceptron wielowarstwowy, konstrukcja funkcji straty oraz niezbędna metodologia. Przeprowadzone zostały analizy dla wybranych równań, skoncentrowane na uzyskanej dokładności i efektywności obliczeniowej oraz czynnikach wpływających na otrzymane wyniki. Niniejsza praca porównuje nowoczesne podejście z tradycyjnymi metodami numerycznymi, ukazując potencjał i ograniczenia sieci neuronowych.

| | | | |
|---|---|---|---|
| Opiekun pracy dyplomowej | dr hab. inż. Łukasz Płociniczak | ..... | ..... |
| | Tytuł/stopień naukowy/imię i nazwisko | ocena | podpis |

*Do celów archiwalnych pracę dyplomową zakwalifikowano do:**

*a) kategorii A (akta wieczyste)*

*b) kategorii BE 50 (po 50 latach podlegające ekspertyzie)*

** niepotrzebne skreślić*

pieczątka wydziałowa

Wrocław, rok 2025

Wrocław University
of Science and Technology

**Faculty of Pure and Applied Mathematics**
Field of study: Applied Mathematics
Specialty: –

Engineering Thesis

# NEURAL NETWORKS IN SOLVING ORDINARY DIFFERENTIAL EQUATIONS

Szymon Stano

keywords:
neural networks, multilayer perceptron, loss function, automatic differentiation, ordinary differential equations

short summary:
This thesis presents the application of neural networks for approximating solutions to ordinary differential equations. The study discusses the multilayer perceptron, the construction of the loss function, and the necessary methodology. Analyses were conducted for selected equations, focusing on the obtained accuracy, computational efficiency, and factors influencing the results. This work compares the modern approach with traditional numerical methods, highlighting the potential and limitations of neural networks.

| | | | |
|---|---|---|---|
| Supervisor | dr hab. inż. Łukasz Płociniczak | ..... | ..... |
| | Title/degree/name and surname | grade | signature |

*For the purposes of archival thesis qualified to:**

*a) category A (perpetual files)*

*b) category BE 50 (subject to expertise after 50 years)*

** delete as appropriate*

stamp of the faculty

Wrocław, 2025

# Spis treści

# Wstęp

Rozwiązywanie równań różniczkowych zwyczajnych jest istotnym elementem wielu dziedzin nauki i inżynierii. Tradycyjne metody numeryczne, choć skuteczne, nierzadko okazują się niewystarczające w przypadku złożonych problemów o wysokiej nieliniowości lub dużej liczbie zmiennych. W odpowiedzi na te wyzwania, rozwój technologii uczenia maszynowego otworzył nowe możliwości w danej dziedzinie z wykorzystaniem sztucznych sieci neuronowych.

Obiecującym podejściem stały się *Physics-Informed Neural Networks* (PINNs) po raz pierwszy zaproponowane w 2017 roku przez Raissi, Perdikaris i Karniadakis. Umożliwiają one efektywne rozwiązywanie równań różniczkowych, zarówno zwyczajnych, jak i cząstkowych, poprzez włączenie informacji o fizycznych prawach w ich strukturę. Rozważane przeze mnie sieci neuronowe będą również zaliczać się do tej kategorii. Wartym uwagi jest fakt, iż w ostatnich latach PINNs zyskały dużą uwagę w środowisku naukowym, a badania nad nimi wciąż trwają.

W niniejszej pracy przybliżę zasadę działania perceptronu wielowarstwowego jako podstawowej architektury sieci neuronowych, szczególnie przydatnej w problemach aproksymacji funkcji. Przedstawię również metodologię rozwiązywania równań różniczkowych dla zaproponowanego modelu, opierającą się na odpowiedniej konstrukcji funkcji straty oraz wykorzystaniu niezwykle efektywnego i ważnego dla sieci neuronowych algorytmu automatycznego różniczkowania. Następnie zaimplementuję wybrane architektury do rozważanej problematyki poszczególnych równań. Podstawową różnicą pomiędzy nimi będą funkcje aktywacyjne – tangens hiperboliczny i funkcja ReLU – oraz harmonogramy współczynnika uczenia – Cosine Decay i ReduceLROnPlateau. Przetestuję ich skuteczność dla każdej z konfiguracji, a następnie porównam uzyskane wyniki z rezultatami tradycyjnych metod numerycznych. Ostatecznie przeanalizuję również zdolność sieci neuronowych do rozwiązywania układów równań różniczkowych, a także tempo wzrostu czasu obliczeniowego w miarę zwiększania liczby równań.

# Rozdział 1

# Wstęp do sieci neuronowych

W niniejszym rozdziale omówię teoretyczne podstawy algorytmów uczenia głębokiego, jakimi są sieci neuronowe. W swojej pracy skupiłem się na sieci typu *feed-forward*, a dokładniej na modelu perceptronu wielowarstwowego z racji jego fundamentalnego znaczenia, uniwersalności zastosowań oraz przejrzystej konstrukcji, dzięki którym możliwe było szczegółowe zrozumienie matematycznych procesów zachodzących w trakcie uczenia się algorytmu. Sieć ta charakteryzuje się budową warstwową oraz jednostronnym przepływem danych, co odróżnia ją przykładowo od sieci rekurencyjnych. Przejdźmy zatem do omówienia poszczególnych zagadnień.

## 1.1 Perceptron wielowarstwowy

Podstawowym zadaniem **sieci neuronowej** [11] (ang. neural network, NN) jest aproksymacja pewnej funkcji $f^*$. Może ona przybierać różne formy. Przykładowo, w przypadku klasyfikacji, polegającej na przypisaniu danych wejściowych do jednej z predefiniowanych klas, $y = f^*(x)$ odwzorowywać będzie dane wejściowe $x$ na kategorię $y$. W przypadku regresji, służącej do przewidywania ciągłych wartości na podstawie danych wejściowych, danymi wyjściowymi będą wartości liczbowe. Sieć neuronowa będzie więc dążyć do jak najlepszego przybliżania tych funkcji. Jej najbardziej podstawowa reprezentacja posiada podobny wygląd, powiększony o jeszcze jeden niezbędny argument, którym jest wektor jej parametrów $\boldsymbol{\theta}$:

$$\mathbf{y} = f(\mathbf{x}, \boldsymbol{\theta}), \tag{1.1}$$

gdzie $\mathbf{x}$ i $\mathbf{y}$ są kolejno danymi wejściowymi i wyjściowymi sieci, a funkcja $f$ jest działaniem, jakie sieć neuronowa nad nimi wykonuje. Oczywiście dokładna postać $f^*$ nie jest znana, a początkowo $f$ może znacznie się od niej różnić. Szczególnie uwzględniając fakt, że parametry $\boldsymbol{\theta}$ są inicjalizowane w sposób losowy. Jednakże, za pomocą **algorytmów optymalizacyjnych** jesteśmy w stanie iteracyjnie zmieniać ich wartości, tak aby różnice między tymi funkcjami znacząco się zminimalizowały. W istocie, proces ten stanowi fundament większości modeli uczenia maszynowego.

**Perceptron wielowarstwowy** [11] (ang. *multilayer perceptron*, MLP) posiada charakterystyczną architekturę, zainspirowaną przez, a także w pewien sposób przypominającą połączenia neuronowe w mózgu, która zarazem pozwala na zaimplementowanie skutecznych

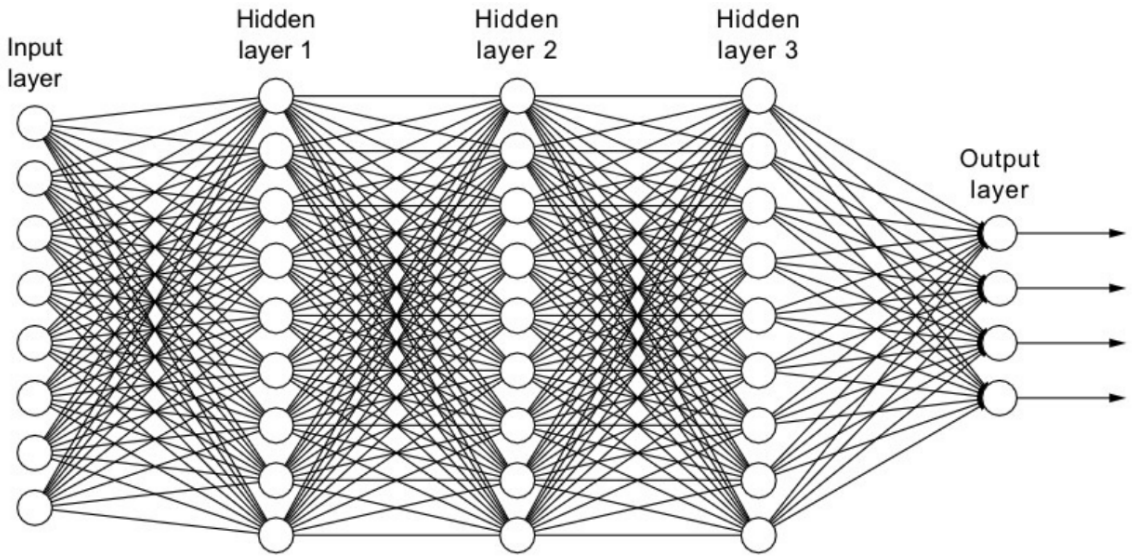
algorytmów uczenia się. Składa się ona z trzech głównych elementów:

1. **Warstwy wejściowej**, odpowiedzialnej za dostarczanie danych do naszego modelu.

2. **Warstw ukrytych**, w których dokonuje się transformacja danych za pomocą funkcji aktywacyjnych, wag oraz wektorów przesunięć (ang. *bias*).

3. **Warstwy wyjściowej**, generującej odwzorowanie danych wejściowych.

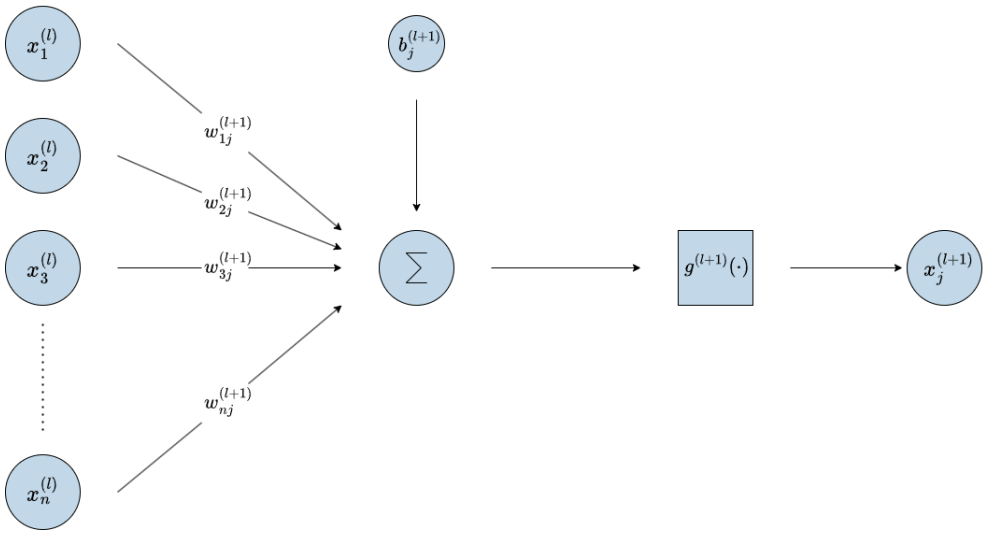
Jej wizualizację możemy zaobserwować na Rysunku 1.1.




Rysunek 1.1: Przykładowa architektura sieci neuronowej z trzema warstwami ukrytymi [30].

## 1.2 Przepływ danych

W trakcie przetwarzania przez sieć danych wejściowych, wykonuje ona nad nimi szereg ściśle określonych operacji [11]. Przyjrzyjmy się, jak wygląda proces dostarczania informacji do pojedynczego neuronu. Każda wartość $x_i^{(l)}$ przechowywana przez neuron $i$ z poprzedniej warstwy $l$, mnożona jest przez odpowiadającą mu wagę $w_{ij}^{(l+1)}$ dla przejścia do neuronu $j$ z następnej warstwy $l+1$. Następnie, wartości te są sumowane, jak i również dodawany jest wektor przesunięcia $b_j^{(l+1)}$. Ostatecznie, na wynik ten nakładana jest funkcja aktywacyjna $g^{(l+1)}$ przypisana do danej warstwy ukrytej. Całość możemy zaobserwować na Rysunku 1.2.




Rysunek 1.2: Przepływ danych dla pojedynczego neuronu.

Proces ten powtarzany jest dla wszystkich neuronów w każdej z kolejnych warstw, aż do warstwy wyjściowej. Możemy również zobrazować, jak operacje te będą wyglądać w szerszej perspektywie, przyglądając się tym razem warstwom. Dla pierwszej z nich mamy

$$\mathbf{h}^{(1)} = g^{(1)}(\mathbf{W}^{(1)T}\mathbf{x} + \mathbf{b}^{(1)}),$$

natomiast dla każdej kolejnej

$$\mathbf{h}^{(l+1)} = g^{(l+1)}(\mathbf{W}^{(l+1)T}\mathbf{h}^{(l)} + \mathbf{b}^{(l+1)}),$$

gdzie $\mathbf{x}$ jest wektorem danych wejściowych, $\mathbf{W}^{(k)}$ są macierzami wag, a $\mathbf{b}^{(k)}$ są wektorami przesunięć dla $k$-tej warstwy.

Łącząc powyższe wzory, otrzymamy alternatywną postać równania (1.1), prezentującą się w sposób

$$\mathbf{y} = g^{(L)}\left(\mathbf{W}^{(L)}g^{(L-1)}\left(\mathbf{W}^{(L-1)}g^{(L-2)}\left(\ldots\mathbf{W}^{(2)}g^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x}+\mathbf{b}^{(1)}\right)+\mathbf{b}^{(2)}\ldots\right)+\mathbf{b}^{(L-1)}\right)+\mathbf{b}^{(L)}\right).$$

## 1.3 Parametry oraz hiperparametry

Dwoma istotnymi elementami w sieciach neuronowych są ich **parametry** oraz **hiperparametry**. Główną różnicą pomiędzy nimi jest to, że hiperparametry definiujemy przed rozpoczęciem treningu i nie są one zmieniane w jego trakcie poprzez algorytm optymalizacyjny, a ich dobór ma wpływ na sposób uczenia oraz architekturę modelu. Mogą one różnić się w zależności od zadania i jego złożoności, jakie dana sieć ma wykonywać. Przy ich wyborze pomocnym jest proces **optymalizacji hiperparametrów**, który dokładniej omówiony jest w sekcji 1.12. Parametry z kolei inicjalizowane są w sposób losowy, a ich wartości aktualizowane są w trakcie uczenia się algorytmu, tak aby spełnić określone przez nas wymagania. W rozważanej przeze mnie architekturze sieci typu MLP wyróżniamy:

**Parametry:**

- wagi,
- wektory przesunięć.

**Hiperparametry:**

- liczba warstw (głębokość sieci),
- liczba neuronów (szerokość sieci),
- funkcje aktywacyjne,
- algorytm optymalizacyjny,
- współczynnik uczenia,
- liczba epok,
- harmonogram współczynnika uczenia,
- ... (oraz wiele innych, w zależności od architektury sieci).

Każdy z nich postaram się dokładnie opisać w niniejszym rozdziale.

## 1.4 Funkcje aktywacyjne

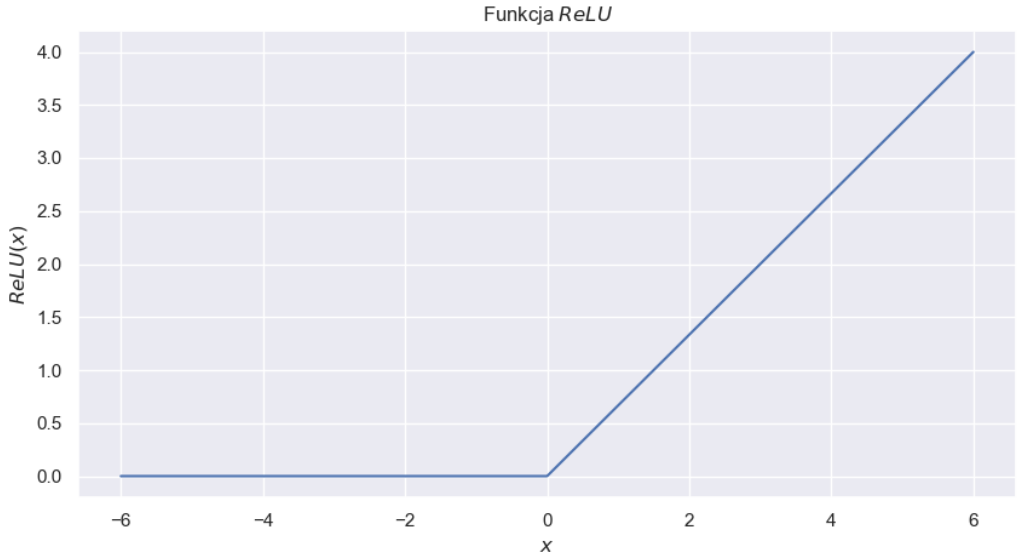
**Funkcje aktywacyjne** [11, 17] są istotnym elementem sieci neuronowych, wprowadzając do naszych modeli nieliniowość, która umożliwia im uczenie się zaawansowanych zależności i wzorców. Bez funkcji aktywacyjnych sieci neuronowe byłyby jedynie złożeniem liniowych przekształceń, co oznacza, że ich zdolność do modelowania relacji oraz struktur w danych byłaby znacznie ograniczona. Ich odpowiedni wybór wpływa na efektywność uczenia, szybkość konwergencji oraz zdolność do uogólniania modelu. Istnieją różne rodzaje funkcji aktywacyjnych, każda z nich posiada swoje wady i zalety, które pokrótce przedstawię poniżej.

### 1.4.1 ReLU

Jedną z najbardziej popularnych i najczęściej polecanych funkcji aktywacyjnych jest funkcja **ReLU** (ang. *rectified linear unit*). Charakteryzuje się ona prostotą oraz małym nakładem kosztów obliczeniowych, co w znacznym stopniu może przyczyniać się do zmniejszenia czasu potrzebnego na trening modelu. Niestety, szczególnie w kontekście rozważanego przeze mnie tematu, nie jest ona różniczkowalna w 0, jak również już jej pierwsza pochodna zeruje się dla $x < 0$. Poniżej przedstawioną mamy jej reprezentację matematyczną:

$$\mathrm{ReLU}(x) = \max(0, x),$$

oraz wykres na Rysunku 1.3:



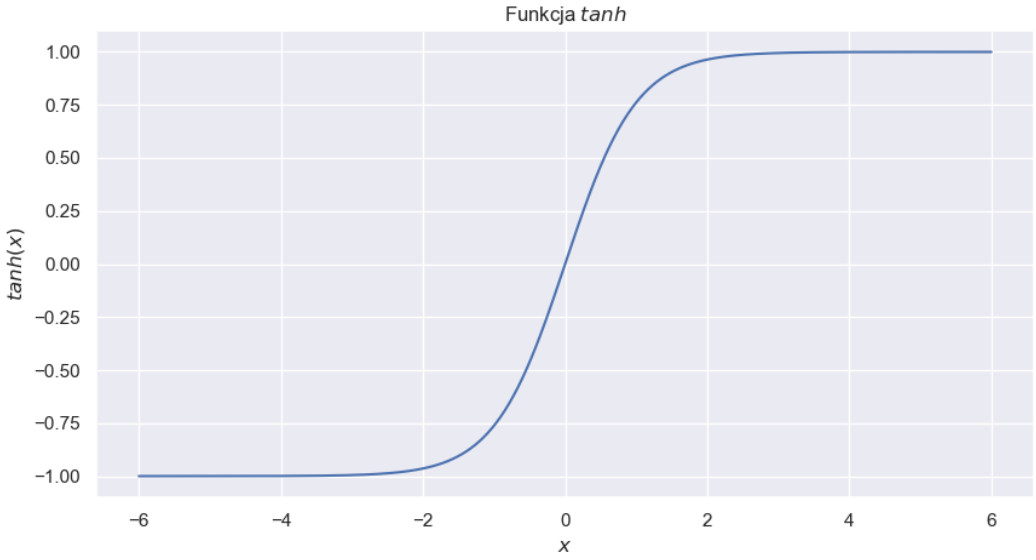

Rysunek 1.3: Funkcja aktywacyjna ReLU

### 1.4.2 Tangens hiperboliczny

Kolejną, również popularną funkcją aktywacyjną, jest **tangens hiperboliczny**. Jej ważnym elementem, szczególnie w problemie aproksymacji rozwiązań równań różniczkowych, jest fakt, że jest nieskończenie wiele razy różniczkowalna w całej swojej dziedzinie. Bardzo dobrze sprawdza się ona w zadaniach wymagających dokładnego modelowania funkcji ciągłych oraz ich pochodnych, co możemy zaobserwować w pracach naukowych poruszających podobną tematykę [6, 13]. Jego postać możemy przedstawić za pomocą:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

Wykres natomiast przedstawiony jest na Rysunku 1.4.




Rysunek 1.4: Funkcja aktywacyjna *tanh*

## 1.5 Funkcja straty

W uczeniu maszynowym **funkcja straty** [11] (ang. *loss function*) jest podstawowym wyznacznikiem tego, jak nasz model radzi sobie z powierzonym mu zadaniem w trakcie „uczenia się". Może przyjmować różne formy, w zależności od problematyki, a co za tym idzie, typu uczenia. Tak więc wyróżniamy:

**Ucznie nadzorowane** w którego procesie dysponujemy danymi treningowymi, zawierającymi odpowiednie etykiety, na podstawie których sieć zbierać będzie niezbędne informacje. W tym przypadku funkcja straty będzie mierzyć różnice pomiędzy rzeczywistymi wartościami danych a ich predykcjami. Do tego typu uczenia zaliczają się takie problemy jak regresja oraz klasyfikacja, a funkcję straty możemy przykładowo przedstawić w sposób:

$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} L(f(\mathbf{x}_i, \boldsymbol{\theta}), y_i), \tag{1.2}$$

gdzie:

- $\boldsymbol{\theta}$ - wektor parametrów modelu
- $\mathbf{x}_i$ - wektor danych wejściowych dla elementu $i$ ze zbioru treningowego
- $f$ - wynik sieci neuronowej
- $y_i$ - etykieta dla elementu $i$ ze zbioru treningowego
- $L$ - wybrana funkcja miary błędu.

**Uczenie nienadzorowane** w przypadku którego model nie ma dostępu do danych zawierających etykiety lub docelowe wartości wyjściowe. W przeciwieństwie do uczenia nadzorowanego, gdzie dane treningowe są wzbogacone o etykiety wskazujące poprawne wyniki, w uczeniu nienadzorowanym sieć neuronowa samodzielnie analizuje strukturę danych, starając się odnaleźć ukryte wzorce, grupy lub zależności. Jej matematyczna postać będzie więc pozbawiona elementu etykiet $y_i$ względem równania (1.2). Do tego rodzaju zaliczamy takie problemy jak klasteryzacja oraz detekcja anomalii.

Proces „uczenia się" sieci neuronowej polega na minimalizacji funkcji straty, którą możemy osiągnąć za pomocą aktualizacji parametrów przy użyciu *algorytmów optymalizacyjnych*. Przykładowo, im mniejsza jej wartość, tym trafniejsze są predykcje w problemie klasyfikacji. Funkcja straty stanowi więc fundament dla nauki maszynowej, ponieważ pozwala na kontrolowanie procesu uczenia, kierując modele w stronę minimalizacji błędów i uzyskiwania jak najlepszych wyników.

## 1.6 Algorytmy optymalizacyjne

Algorytmy optymalizacyjne odgrywają kluczową rolę w procesie uczenia maszynowego, w szczególności w trenowaniu sieci neuronowych. Stanowią one mechanizm iteracyjnego

dopasowywania parametrów modelu (wag i wektorów przesunięć) w taki sposób, aby minimalizować funkcję celu, zwykle zdefiniowaną jako funkcja straty. Optymalizacja tej funkcji jest sednem „uczenia się" sieci neuronowej.

Proces ten można opisać jako poszukiwanie minimum funkcji $J(\boldsymbol{\theta})$ w przestrzeni parametrów $\mathbb{R}^n$. Większość algorytmów wykorzystuje liczenie gradientów, na podstawie których jesteśmy w stanie wyznaczyć sposób zmiany $\boldsymbol{\theta}$, tak aby zmniejszyć wartość funkcji straty. W najprostszym przypadku algorytm **gradientu prostego** [23] (ang. *Gradient descent*, *Batch gradient descent*, GD) aktualizuje parametry zgodnie z regułą:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla_{\boldsymbol{\theta}_t} J(\boldsymbol{\theta}_t), \tag{1.3}$$

gdzie $\eta$ jest hiperparametrem oznaczającym **współczynnik uczenia** (1.7). W powyższej metodzie gradienty liczone są dla całego zbioru danych, co często może okazać się procesem czasochłonnym, szczególnie w przypadku znaczących ilości danych.

### 1.6.1 Adam

W celu zwiększenia efektywności optymalizacji w dużych przestrzeniach parametrów oraz przyspieszenia zbieżności, opracowano szereg bardziej zaawansowanych algorytmów niż klasyczny gradient prosty. Jednym z najczęściej stosowanych w praktyce jest algorytm **Adam** [15, 11] (ang. *Adaptive Moment Estimation*).

Algorytm Adam bazuje na adaptacyjnych estymatorach momentów pierwszego i drugiego rzędu dla gradientu funkcji straty. Pozwala na dynamiczne dostosowywanie współczynników uczenia dla różnych parametrów w trakcie optymalizacji. Jest on efektywny obliczeniowo i ma niewielkie wymagania pamięciowe. Adam połączył zalety często używanych przed nim metod takich jak AdaGrad [8] oraz RMSProp [27].

#### Reguła aktualizacji parametrów

Reguła iteracyjnej aktualizacji parametrów dla algorytmu Adam wygląda następująco:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}, \tag{1.4}$$

gdzie:

- $\boldsymbol{\theta}_t$ to wektor parametrów modelu w $t$-tej iteracji,
- $\eta$ to stała współczynnika uczenia,
- $\hat{m}_t$ to szacowany pierwszy moment z wprowadzoną korektą,
- $\hat{v}_t$ to szacowany drugi moment z wprowadzoną korektą,
- $\epsilon$ to mała liczba (zwykle $10^{-8}$), która zapobiega dzieleniu przez zero.

#### Pierwszy moment ($m_t$)

Pierwszy moment gradientu jest estymowany za pomocą wykładniczej średniej ruchomej:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t),$$

gdzie $m_{t-1}$ to poprzedni szacunkowy pierwszy moment, a $\beta_1$ jest współczynnikiem kontrolującym wykładnicze tempo zaniku w tej średniej, zazwyczaj ustawiany na wartość bliską 1, najczęściej 0.9.

#### Drugi moment ($v_t$)

Estymator drugiego momentu, adekwatnie do pierwszego, również wyrażony jest za pomocą wykładniczej średniej ruchomej:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t))^2,$$

gdzie $v_{t-1}$ to poprzedni szacunkowy drugi moment. Współczynnik $\beta_2$ również ustawiany jest na wartość bliską 1. W tym przypadku zalecanym wyborem jest 0.999.

#### Korekta zbyt małych momentów ($\hat{m}_t$, $\hat{v}_t$)

Ze względu na inicjalne wartości $m_0 = 0$ i $v_0 = 0$, pierwsze iteracje mogą prowadzić do zbyt małych wartości momentów. W związku z tym stosuje się korektę:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}.$$

Dzięki tym korektom, Adam staje się bardziej odporny na początkowe błędy oszacowań momentów, co skutkuje lepszą stabilnością na początku treningu.

## 1.7 Współczynnik uczenia

**Współczynnik uczenia** [31] (ang. *learning rate*, LR), oznaczany często poprzez $\mu$, odpowiada za stopień aktualizacji parametrów względem wyznaczonego kierunku przez algorytm optymalizacyjny (równania (1.3), (1.4)). Bezpośrednio wpływa na czas treningu, lecz jego dobór może nieść za sobą różnorodne konsekwencje. Większa wartość może prowadzić do szybszego osiągnięcia przez algorytm minimum globalnego, lecz zbyt duża może utrudniać zbieżność, skutkować oscylacjami lub osiągnięciem minimum lokalnego. Mniejsza z kolei prowadzi do stabilniejszej i dokładniejszej nauki, kosztem czasu kompilacji. Jest to kolejny hiperparametr, który warto poddać optymalizacji (1.12) przed rozpoczęciem treningu.

## 1.8 Harmonogram współczynnika uczenia

Wraz z rozwojem uczenia maszynowego podjęto liczne próby optymalizacji współczynnika uczenia oraz zrozumienia jego wpływu na proces treningu. W rezultacie opracowano **harmonogramy współczynnika uczenia** [31] (ang. *learning rate schedulers*), których głównym celem jest stopniowe zmniejszanie wartości tego parametru w trakcie kolejnych iteracji, zgodnie z określonymi zasadami. Taka strategia umożliwia szybszą zbieżność w początkowej fazie uczenia oraz poprawę dokładności modelu w fazie końcowej.

W niniejszej pracy postanowiłem skupić się na dwóch wariantach i porównać ich wydajność:

### 1.8.1 Cosine Decay

Harmonogram ten opiera się na funkcji kosinusowej i zmniejsza wartość współczynnika uczenia się zgodnie z następującą regułą [25, 18]:

$$\eta(t) = \eta_{\text{initial}} \left( \alpha + (1 - \alpha) \frac{1}{2} \left( 1 + \cos \left( \frac{t}{T} \pi \right) \right) \right),$$

gdzie $t$ oznacza aktualną iterację, $T$ całkowitą liczbę iteracji, $\eta_{\text{initial}}$ początkową wartość współczynnika uczenia, a $\alpha$ hiperparametr określający procent $\eta_{\text{initial}}$ do jakiego będzie dążyć współczynnik uczenia pod koniec treningu ($\alpha \in [0, 1]$).

Główną zaletą tej metody jest uzyskanie wysokiej szybkości zbieżności w początkowej fazie procesu treningowego, a także stabilności w końcowej fazie.

### 1.8.2 ReduceLROnPlateau

Ten harmonogram dynamicznie dostosowuje wartość współczynnika uczenia się w zależności od zmian monitorowanego czynnika, przykładowo funkcji straty [26]. Jeśli wartość straty nie poprawia się przez określoną przez nas liczbę epok, współczynnik uczenia się zostaje zmniejszony o zdefiniowany faktor. W przeciwnym razie pozostaje on bez zmian. Mechanizm ten pozwala na bardziej precyzyjne dostosowanie tempa uczenia w trakcie treningu.

## 1.9 Liczba epok

Epoka w trenowaniu sieci neuronowej odnosi się do jednokrotnego przejścia przez cały zbiór danych wejściowych bądź treningowych, podczas którego model aktualizuje swoje parametry. Ich liczba może być utożsamiana z długością uczenia się danego modelu.

## 1.10 Automatyczne różniczkowanie

**Automatyczne różniczkowanie** (ang. *Automatic Differentiation*, AD) [5, 28] to technika obliczania pochodnych funkcji zdefiniowanych przez programy komputerowe, łącząca dokładność metod symbolicznych z efektywnością metod numerycznych. W kontekście uczenia maszynowego AD umożliwia efektywne i dokładne obliczanie gradientów funkcji straty względem parametrów modelu, co jest kluczowe dla procesów optymalizacyjnych.

### Podstawy matematyczne AD

Automatyczne różniczkowanie opiera się na regule łańcucha rachunku różniczkowego, która dla funkcji złożonej $y = f(g(x))$ wyraża pochodną jako:

$$\frac{dy}{dx} = \frac{df}{dg} \frac{dg}{dx}.$$

W praktyce wszystkie obliczenia numeryczne są złożeniem skończonej liczby elementarnych funkcji, dla których pochodne są znane. AD analizuje dany program, a następnie, stosując regułę łańcucha, oblicza pochodne złożonych wyrażeń, propagując ich wartości przez kolejne jego operacje.

## 1.11 Wsteczna propagacja błędu

**Wsteczna propagacja błędu** (ang. *backpropagation*) [24, 11] to algorytm służący do obliczania gradientów funkcji straty w uczeniu głębokim, choć nie ogranicza się on tylko do tego zastosowania. W kontekście sieci neuronowych jest on zoptymalizowaną implementacją **automatycznego różniczkowania w trybie odwrotnym** (ang. *Reverse Mode AD*) dla ich charakterystycznej, hierarchicznej struktury [5]. Algorytm składa się z dwóch etapów:

1. **Forward pass**: Obliczenie wartości wyjściowych sieci $\hat{\mathbf{y}}$ na podstawie danych wejściowych $\mathbf{x}$ i aktualnych parametrów $\boldsymbol{\theta}$.

2. **Backward pass**: Propagacja gradientów funkcji straty od wyjścia sieci przez kolejne warstwy w kierunku wejścia, obliczając pochodne względem wag i wektorów przesunięć w każdej warstwie.

Wsteczna propagacja błędu jest fundamentem trenowania sieci neuronowych i umożliwia dostosowywanie ich parametrów w celu minimalizacji funkcji straty. Jest to kluczowa metoda w algorytmach optymalizacyjnych takich jak Adam czy RMSProp, które korzystają z obliczonych gradientów w celu aktualizacji wag.

## 1.12 Optymalizacja hiperparametrów

Ostatnim elementem, który stosuje się przed zaprojektowaniem sieci i inicjalizacją treningu, jest **Optymalizacja Hiperparametrów** (ang. *Hyperparameter Optimization*, HPO) [9]. Jak wskazuje sama nazwa, proces ten pozwala odpowiednio dobrać wartości hiperparametrów, aby sieć neuronowa mogła jak najlepiej poradzić sobie z powierzonym jej zadaniem.

Oczywiście można ten krok pominąć i ręcznie wybrać odpowiednie wartości na podstawie wiedzy eksperckiej. Jednak w wielu zagadnieniach, szczególnie w złożonych problemach, takich jak rozwiązywanie równań różniczkowych, trafny dobór hiperparametrów bywa trudny do przewidzenia. Na przykład, możemy intuicyjnie oczekiwać, że w tym przypadku funkcja aktywacyjna tangens hiperboliczny będzie bardziej odpowiednia niż ReLU, mając na uwadze gładkość rozwiązań. Niemniej jednak, precyzyjne ustalenie takich parametrów jak współczynnik uczenia się czy liczba warstw ukrytych wymaga często bardziej systematycznego podejścia.

Istnieje wiele metod optymalizacji hiperparametrów, różniących się stopniem złożoności i wymagań obliczeniowych. Do najczęściej stosowanych należą:

- **Grid Search** – systematyczne przeszukiwanie przestrzeni hiperparametrów poprzez testowanie wszystkich możliwych kombinacji w z góry określonej siatce wartości. Metoda ta gwarantuje, że żadna kombinacja nie zostanie pominięta, ale jest bardzo kosztowna obliczeniowo przy dużej liczbie hiperparametrów.

- **Random Search** – prostsza i efektywniejsza niż Grid Search w wielu przypadkach. Polega na losowym wybieraniu wartości hiperparametrów z określonych przedziałów. Losowość pozwala na szerokie przeszukanie przestrzeni bez potrzeby testowania wszystkich kombinacji, co znacznie zmniejsza koszty obliczeniowe przy zachowaniu wysokiego prawdopodobieństwa znalezienia dobrego rozwiązania.

- **Bayesian Optimization** – podejście probabilistyczne, które modeluje zależność między hiperparametrami a wynikami za pomocą modeli bayesowskich, takich jak procesy gaussowskie. Dzięki temu, proces optymalizacji może iteracyjnie wybierać kolejne zestawy hiperparametrów do oceny, bazując na przewidywanej wydajności modelu. Choć metoda ta jest bardziej efektywna niż grid search i random search, szczególnie w przypadku wysokowymiarowych przestrzeni hiperparametrów, jej głównym ograniczeniem jest większe zapotrzebowanie na zasoby obliczeniowe.

W mojej pracy zdecydowałem się na zastosowanie **Bayesian Optimization** do optymalizacji hiperparametrów, ze względu na jego efektywność w przeszukiwaniu wysokowymiarowych przestrzeni hiperparametrów. Dzięki probabilistycznemu podejściu, metoda ta minimalizuje liczbę prób, przyspieszając proces optymalizacji i zwiększając precyzję w doborze najlepszych wartości hiperparametrów.

# Rozdział 2

# Metodologia rozwiązania

W tym momencie, po zapoznaniu się z uczeniem głębokim, możemy przejść do kluczowego elementu rozważanego zagadnienia, jakim jest metoda rozwiązywania równań różniczkowych.

Sieci neuronowe są często kojarzone z problemami uczenia nadzorowanego, w których dysponujemy zestawem danych treningowych. Podejście to nie wydaje się jednak odpowiednie w naszym przypadku – trudno bowiem uzasadnić naukę rozwiązania równań różniczkowych w oparciu o zbiory zawierające już gotowe rozwiązania, gdyż kłóciłoby się to z istotą samego problemu.

Jak możemy intuicyjnie zauważyć, nasze zagadnienie należy do obszaru **uczenia nienadzorowanego**. Jako dane wejściowe sieci neuronowej dostarczać będziemy zadany przez nas przedział zmiennych niezależnych - czasu lub przestrzeni - na którym będziemy chcieli, żeby wybrane równanie różniczkowe zostało rozwiązane. Jak wiemy z Rozdziału 1.2, zaprezentowany perceptron wielowarstwowy jest niczym innym niż funkcją danych wejściowych oraz jego parametrów. Będziemy chcieli więc, żeby funkcja ta stała się rozwiązaniem naszego równania, którego uzyskanie będzie możliwe poprzez modyfikację parametrów. Tym sposobem problem rozwiązania równania różniczkowego sprowadzi się do zagadnienia optymalizacyjnego. Kluczową rolę w tym procesie odgrywać będzie specyficzna konstrukcja **funkcji straty**, która zaprezentowana zostanie w niniejszym rozdziale.

## 2.1 Konstrukcja funkcji straty

Metoda opierająca się na odpowiedniej konstrukcji funkcji straty pojawia się w wielu pracach naukowych o podobnej tematyce, również dla równań różniczkowych cząstkowych, a nawet problemów zawierających dane eksperymentalne [4, 14, 10, 16, 22]. Zbiór tych zagadnień znany jest jako *Physics-Informed Neural Networks* (PINNs) i w ostatnim czasie jest prężnie rozwijającą się dziedziną z zakresu uczenia głębokiego. Formy funkcji straty w tych problemach potrafią nieznacznie się różnić, ale logika stojąca za ich ideą jest identyczna. W każdej z tych metod celem jest skonstruowanie funkcji straty, która będzie uwzględniała zarówno **błąd pomiędzy przewidywaniami modelu a danymi** (np. warunki początkowe lub brzegowe, dane eksperymentalne), jak i **błąd residualny**, który opisuje różnicę między lewą a prawą stroną równania różniczkowego. Dzięki temu sieć neuronowa jest zmuszona do minimalizacji błędu w dwóch aspektach – dopasowania do danych oraz zgodności z równaniem różniczkowym. Proces optymalizacji umożliwia stopniowe dostosowywanie parametrów modelu, tak by uzyskać rozwiązanie, które spełnia zarówno dane, jak i równanie w sensie całkowitym.

Zacznijmy więc od zadanego równania różniczkowego zwyczajnego. Na rzecz naszego problemu, istotne jest jego przedstawienie w formie residualnej:

$$F(t, y(t), y'(t), ..., y^{(k)}(t)) = 0,$$

z zadanymi warunkami początkowymi:

$$[y(t_0), y'(t_0), ..., y^{(k)}(t_0)] = [y_0^0, y_0^1, ..., y_0^k].$$

W identyczny sposób można również uwzględnić warunki brzegowe, jeśli problem tego wymaga.

Przyjmijmy teraz, że wynik sieci neuronowej dla danych wejściowych $t$ przedstawiony będzie za pomocą $\hat{y}_\theta(t)$. To właśnie tą funkcję będziemy chcieli w możliwie najlepszym stopniu przybliżyć do dokładnego rozwiązania równania różniczkowego:

$$\hat{y}_\theta(t) \sim y(t).$$

Zadane równanie różniczkowe będziemy rozwiązywać na określonym przedziale, od $t_0$ do $t_{n-1}$, dzieląc go na n punktów. Wartości w punktach $t_i$ stanowić będą zestaw danych wejściowych dla naszej sieci neuronowej, której zadaniem będzie znalezienie funkcji $\hat{y}_\theta(t)$ najlepiej odwzorowującej rozwiązanie równania na tym przedziale.

Z racji charakterystycznej architektury naszego modelu, za pomocą **automatycznego różniczkowania** (1.10) jesteśmy w efektywny sposób w stanie liczyć kolejne pochodne tego wyniku względem danych wejściowych [5, 21], a więc wyrażenia: $\hat{y}'_\theta(t_i), ..., \hat{y}^{(k)}_\theta(t_i)$.

Posiadając wszystkie te elementy, jesteśmy w stanie przystąpić do konstrukcji funkcji straty. W swojej pracy postanowiłem zdefiniować błąd residualny w sensie najmniejszych kwadratów, a więc:

$$J(\theta)_{\text{ODE}} = \frac{1}{n} \sum_{i=0}^{n-1} \Big( F\Big(t_i, \hat{y}_\theta(t_i), \hat{y}'_\theta(t_i), ..., \hat{y}^{(k)}_\theta(t_i)\Big)\Big)^2,$$

oraz adekwatnie, błąd odpowiadający za warunki początkowe:

$$J(\theta)_{\text{IC}} = \frac{1}{k} \sum_{i=1}^{k} \Big( \hat{y}^{(i)}_\theta(t_0) - y_0^i \Big)^2.$$

Połączenie tych elementów, wraz z przeskalowaniem o ustalone wagi $\omega$, da nam końcową postać, którą wykorzystywać będziemy za każdym razem dla danej problematyki:

$$J(\theta) = \omega_{\text{ODE}} J(\theta)_{\text{ODE}} + \omega_{\text{IC}} J(\theta)_{\text{IC}}.$$

## 2.2 Etapy rozwiązania

Znając odpowiednią konstrukcję funkcji straty, jesteśmy w stanie przystąpić do sformułowania kroków potrzebnych do rozwiązania równania różniczkowego poprzez sieć neuronową.

Prezentują się one w sposób następujący:

1. Ustalenie przedziału oraz zagęszczenia danych wejściowych.
2. Konstrukcja funkcji straty dla zadanego równania.
3. Dobór hiperparametrów (lub przeprowadzenie ich optymalizacji)
4. Inicjalizacja sieci neuronowej.
5. Trening modelu - każda iteracja treningowa (epoka) będzie składać się z kolejnych elementów:
   - Obliczenie niezbędnych pochodnych względem wyjścia sieci za pomocą AD.
   - Dostarczenie powyższych wyników do funkcj straty.
   - Aktualizacja parametrów przy użyciu algorytmu optymalizacyjnego.
6. Predykcja wyniku.

Powyższe kroki są również stosowane podczas procesu Optymalizacji Hiperparametrów (1.12), lecz oczywiście przeprowadzane są dla znacznie mniejszej ilości epok niż w finalnym rozwiązaniu.

# Rozdział 3

# Przegląd i analiza przykładowych równań

W tym rozdziale przejdziemy do docelowego elementu tej pracy - praktycznego rozwiązywania równań różniczkowych. Wszystkie programy, symulacje i modele zostały zaimplementowane w języku *Python*, za pomocą bibliotek:

- **TensorFlow** [1] – Biblioteka do budowania i trenowania modeli sieci neuronowych.
- **Optuna** [2] – Biblioteka do optymalizacji hiperparametrów.
- **Numpy** [12] – Biblioteka do obliczeń numerycznych.
- **SciPy** [29] – Biblioteka do zaawansowanych obliczeń numerycznych i naukowych.

## Dobór hiperparametrów

Każdy z rozważanych przeze mnie przypadków, których elementami było przeprowadzenie optymalizacji hiperparametrów, a następnie inicjalizacja i trening docelowego modelu, posiadał grupę wspólnych hiperparametrów; są one przedstawione w Tabeli 3.1.

Tabela 3.1: Wspólne hiperparametry

| Przedział czasowy | Liczba danych wejściowych - $n$ | Algorytm optymalizacyjny | Liczba epok |
|---|---|---|---|
| $[0, 10]$ | 1000 | *Adam* | 2000 |

*Źródło: opracowanie własne*

Punkty w przedziale czasowym zostały rozmieszczone w jednakowych odstępach, a z racji $n = 1000$, sieci neuronowe będą posiadały warstwy wejściowe i wyjściowe o tej samej liczbie neuronów. Modele sieci konstruowane były za pomocą biblioteki *TensorFlow*, natomiast optymalizacja pozostałych hiperparametrów odbywała się za pomocą biblioteki *Optuna* i zaimplementowanej w niej metody *Bayesian Optimization*. Każda z takich optymalizacji przeprowadzała 50 prób, wszystkie o długości 50 epok. Przedziały, z których możliwy był wybór danego hiperparametru, przedstawione są w Tabeli 3.2

Pozostałe hiperparametry: funkcje aktywacyjne oraz harmonogramy współczynnika uczenia zostały poddane głębszej analizie, przedstawionej w dalszej części pracy.

Tabela 3.2: Przedziały optymalizowanych hiperparametrów

| Liczba warstw ukrytych | Liczba neuronów w każdej z warstw ukrytych | Współczynnik uczenia |
|---|---|---|
| $[1, 15]$ | $[32, 256]$ | $[10^{-4}, 10^{-2}]$ |

*Źródło: opracowanie własne*

### Hiperparametry harmonogramów współczynnika uczenia

Dodatkowo, dwa rozważane przeze mnie harmonogramy współczynnika uczenia posiadały niezbędne parametry, których wartości ustaliłem jako:

- **Cosine Decay**: $\alpha = 0.1$ - odpowiadający za proporcję początkowej wartości współczynnika uczenia do której zbiegała jego wartość pod koniec treningu.
- **ReduceLROnPlateau**: *patience* $= 10$, *reduce factor* $= 0.8$, *min lr*$= 5 \cdot 10^{-5}$, które odpowiadały kolejno za; liczbę epok bez zmiany, po której współczynnik uczenia był redukowany; współczynnik redukcji; minimalną wartość aktualizowanego współczynnika uczenia.

## 3.1 Równanie jednorodne pierwszego rzędu

Na początek, w celu przeprowadzenia dokładnej analizy, przyjrzymy się prostemu równaniu jednorodnemu:

$$\begin{cases} y' + y = 0, \\ y(0) = 1, \end{cases}$$

o trywialnym rozwiązaniu:

$$y = e^{-t}.$$

Funkcję straty możemy więc skonstruować w następujący sposób:

$$J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=0}^{n-1} \left( \hat{y}'_\theta(t_i) + \hat{y}_\theta(t_i) \right)^2 + \left( \hat{y}_\theta(0) - 1 \right)^2.$$

W tym momencie możemy przystąpić do docelowego trenowania sieci neuronowych. Początkową analizę przeprowadzimy względem harmonogramów współczynnika uczenia.

### 3.1.1 Trening sieci neuronowych

#### Cosine Decay

Dla dwóch wybranych funkcji aktywacyjnych: ReLU i tangensa hiperbolicznego, optymalizacja hiperparametrów zakończyła się z wynikami zawartymi w Tabeli 3.3. Znacznie różniącym się hiperparametrem okazuje się być liczba neuronów - niemalże trzy razy więcej w przypadku funkcji aktywacyjnej ReLU. Pozostałe wartości są sobie równe, bądź do siebie zbliżone.

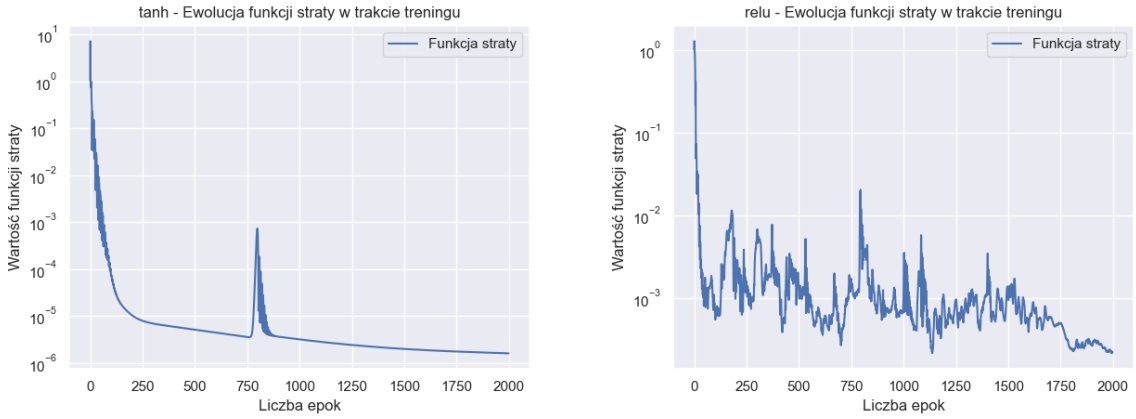
Historia wartości funkcji strat względem kolejnych epok przedstawiona jest na Rysunku 3.1, natomiast historia wartości współczynnika uczenia przedstawiona jest na Rysunku 3.2.

Tabela 3.3: Otrzymane hiperparametry - Cosine Decay

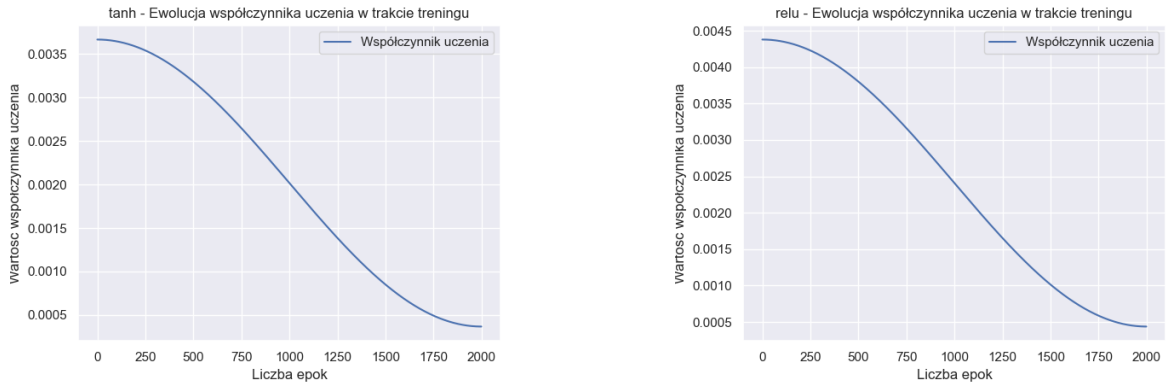
| Funkcja aktywacyjna | Liczba warstw ukrytych | Liczba neuronów w każdej z warstw ukrytych | Współczynnik uczenia |
|---|---|---|---|
| tanh | 9 | 84 | $\approx 0.0037$ |
| ReLU | 9 | 216 | $\approx 0.0044$ |

*Źródło: opracowanie własne*

Jak możemy zauważyć, proces treningu dla sieci z funkcją ReLU był znacznie bardziej niestabilny, pomimo podobnej dynamiki zmiany współczynnika uczenia.




Rysunek 3.1: Historia wartości funkcji straty - Cosine Decay

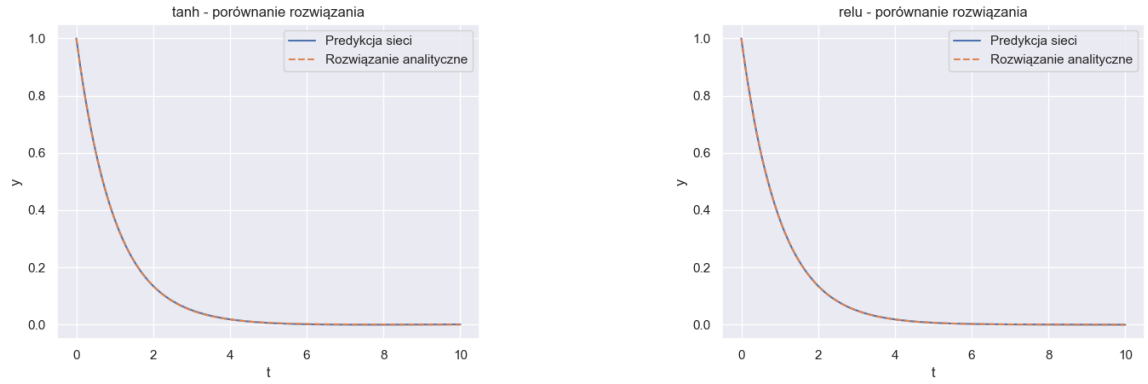



Rysunek 3.2: Historia wartości współczynnika uczenia - Cosine Decay

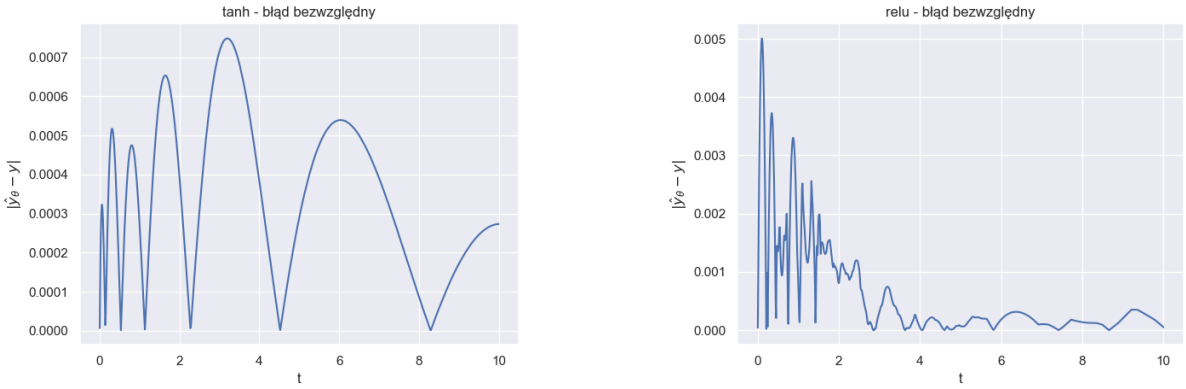
Na końcu zaprezentowano porównanie wyników z rozwiązaniami analitycznymi (Rysunek 3.3) oraz błędów bezwzględnych (Rysunek 3.4). Tylko te drugie umożliwiają uwidocznienie subtelnych różnic między rozwiązaniami, które są odpowiednio rzędów $10^{-4}$ oraz $10^{-3}$. Interesującym aspektem jest fakt, że predykcje sieci częściej i bardziej gwałtownie oscylują wokół dokładnego rozwiązania dla argumentów zbliżających się do zera. Może wynikać to z faktu, że funkcja $e^{-t}$, wraz ze wzrostem argumentów, redukuje się do wartości

zbliżonych zeru, a jej dalsza ewolucja jest niemalże liniowa. Prawdopodobnym jest, że wagi oraz wektory przesunięć również wyzerowywują się dla rosnących argumentów, ograniczając swoją zmienność.

Ponadto, wartym zauważenia jest również zachowanie maksimum lokalnych błędów bezwzględnych dla modelu z funkcją aktywacyjną ReLU. Przypominają one funkcję wykładniczą - identyczną z tą, która jest rozwiązaniem danego równania.




Rysunek 3.3: Porównanie predykcji sieci z rozwiązaniem analitycznym - Cosine Decay




Rysunek 3.4: Błąd bezwzględny rozwiązania - Cosine Decay

### ReduceLROnPlateau

Adekwatnie do Cosine Decay przeprowadzone zostało uczenie dla modeli z harmonogramem ReduceLROnPlateau. Wyniki optymalizacji hiperparametrów znajdują się w Tabeli 3.4, natomiast treningu na Rysunkach 3.5, 3.6, 3.7, 3.8. Proces uczenia dla sieci z funkcją aktywacyjną ReLU znów wydaje się być bardziej niestabilny względem modelu z funkcją tangensa hiperbolicznego, jednakże w porównaniu do poprzedniej sieci z tą samą funkcją aktywacyjną (Rysunek 3.1) możemy zaobserwować znaczną poprawę, nie tylko pod względem ograniczonych oscylacji, ale również mniejszej wartości funkcji straty po zakończeniu treningu.

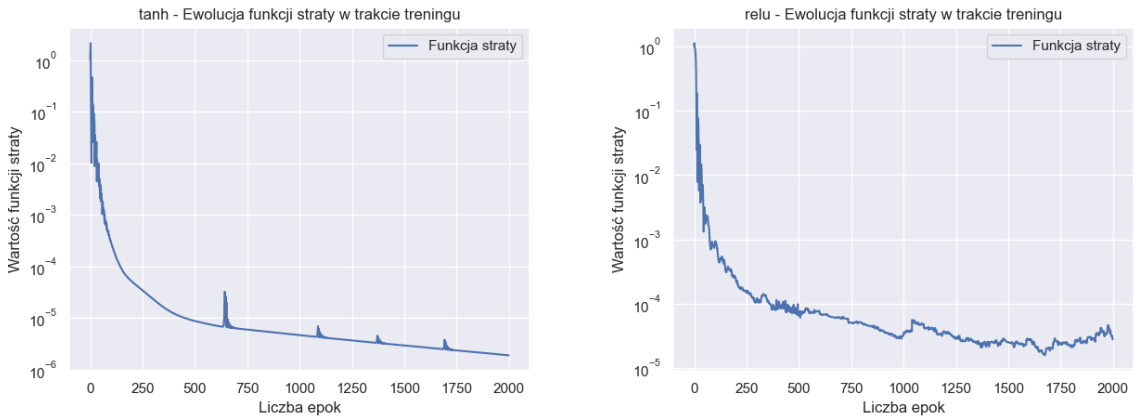
Podczas treningu sieci z funkcją aktywacyjną tangensa hiperbolicznego wystąpiło kilka przypadków zaburzenia stabilności nauki i wzrostu wartości funkcji straty. Zachowania te zostały jednak szybko zniwelowane poprzez interwencję harmonogramu ReduceLROnPlateau i następujące zmniejszenie współczynnika uczenia.

Podobne zdarzenia miały miejsce dla drugiego modelu, jednakże duża chaotyczność spowodowała szybką redukcję współczynnika uczenia do minimalnej ustalonej wartości.

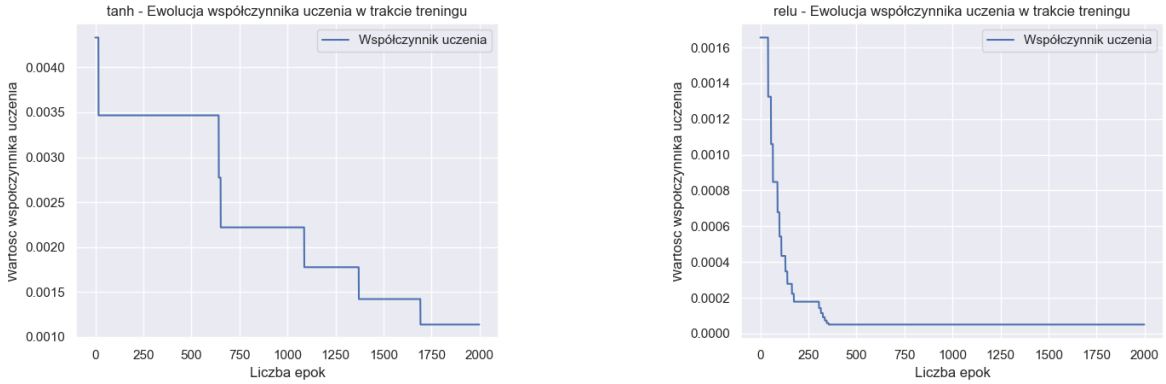
Tabela 3.4: Otrzymane hiperparametry - ReduceLROnPlateau

| Funkcja aktywacyjna | Liczba warstw ukrytych | Liczba neuronów w każdej z warstw ukrytych | Współczynnik uczenia |
|---|---|---|---|
| tanh | 15 | 42 | $\approx 0.0044$ |
| ReLU | 9 | 245 | $\approx 0.0017$ |

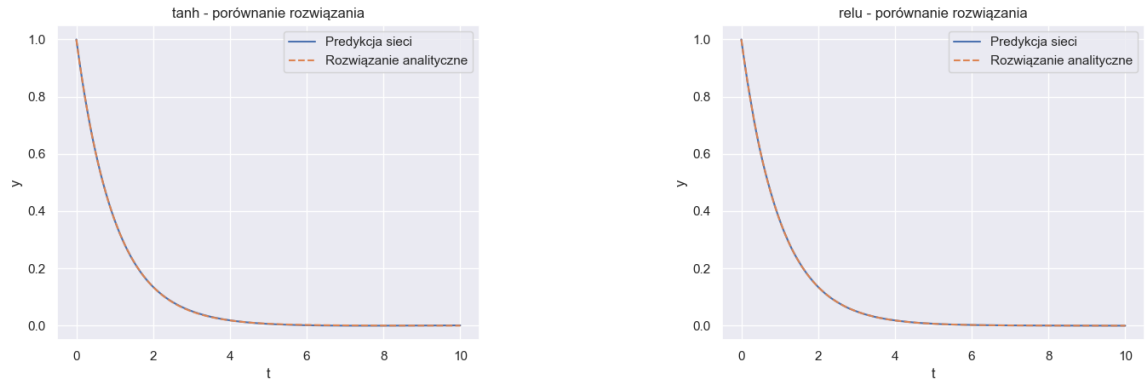
*Źródło: opracowanie własne*



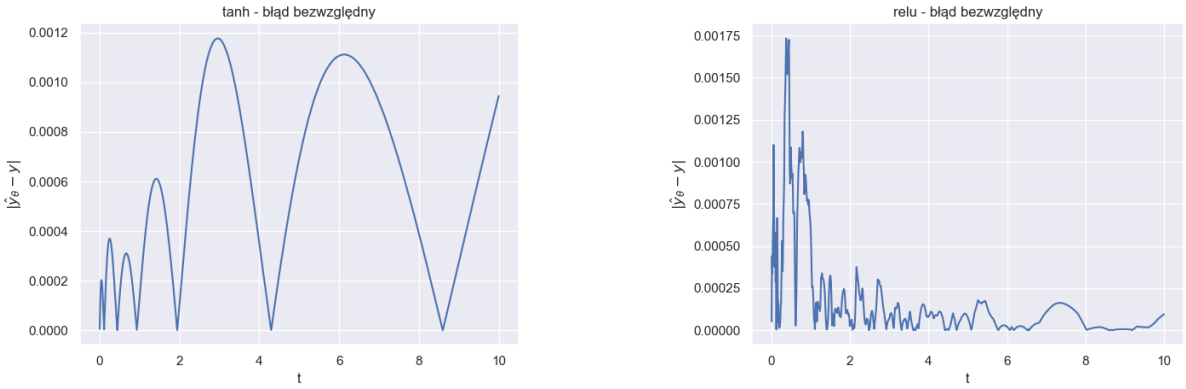

Rysunek 3.5: Historia wartości funkcji straty - ReduceLROnPlateau




Rysunek 3.6: Historia wartości współczynnika uczenia - ReduceLROnPlateau




Rysunek 3.7: Porównanie predykcji sieci z rozwiązaniem analitycznym - ReduceLROnPlateau




Rysunek 3.8: Błąd bezwzględny rozwiązania - ReduceLROnPlateau

### 3.1.2 Ewaluacja modeli

W tej sekcji przeprowadzimy porównanie przedstawionych wcześniej modeli na podstawie wybranych metryk oceny jakości. Jedną z kluczowych miar skuteczności jest dokładność rozwiązania, która została oceniona za pomocą błędu średniokwadratowego (ang. *Mean Squared Error*, MSE) oraz średniego błędu bezwzględnego (ang. *Mean Absolute Error*, MAE). Dodatkowo, w analizie uwzględniono inne istotne wskaźniki, takie jak czas treningu oraz całkowita liczba neuronów ukrytych - wartość proporcjonalna do zużycia pamięci. Zaokrąglone wyniki dla sieci neuronowych zostały przedstawione w Tabeli 3.5.

Najlepsze wyniki pod względem dokładności, zarówno w metryce MSE, jak i MAE, osiągnął model wykorzystujący harmonogram uczenia **ReduceLROnPlateau** oraz funkcję aktywacyjną **ReLU**. Otrzymane wartości błędów były niemal dwukrotnie mniejsze w porównaniu z drugim najlepszym modelem, wykorzystującym harmonogram **Cosine Decay** i funkcję aktywacyjną **tangensa hiperbolicznego**.

Jednakże dalsza analiza wykazała, że czas treningu modelu z ReduceLROnPlateau był około czterokrotnie dłuższy, a liczba neuronów ukrytych wynosiła niemal trzykrotnie więcej niż w modelu z Cosine Decay. Osiągnięcie wyższej dokładności wiązało się więc z istotnym wzrostem kosztów obliczeniowych, zarówno pod względem czasu treningu, jak i zużycia zasobów pamięciowych.

Tabela 3.5: Sieci neuronowe - porównanie wyników

| Harmonogram współczynnika uczenia | Funkcja aktywacyjna | Błąd średnio-kwadratowy | Średni błąd bezwzględny | Czas treningu | Liczba neuronów ukrytych |
|---|---|---|---|---|---|
| CD | tanh | $1.58 \cdot 10^{-7}$ | $3.43 \cdot 10^{-4}$ | 27.26s | 756 |
| CD | ReLU | $9.55 \cdot 10^{-7}$ | $5.49 \cdot 10^{-4}$ | 64.85s | 1944 |
| RLROP | tanh | $4.92 \cdot 10^{-7}$ | $6.04 \cdot 10^{-4}$ | 27.18s | 630 |
| RLROP | ReLU | $8.86 \cdot 10^{-8}$ | $1.47 \cdot 10^{-4}$ | 103.67s | 2205 |

CD - Cosine Decay, RLROP - ReduceLROnPlateau
*Źródło: opracowanie własne*

Pojawia się zatem pytanie, czy taka różnica w dokładności uzasadnia dodatkowy nakład zasobów. Istnieje możliwość, że w tym samym czasie model z Cosine Decay mógłby zostać przetrenowany na zdecydowanie większej ilości epok, co potencjalnie pozwoliłoby mu osiągnąć porównywalną lub nawet wyższą dokładność.

### 3.1.3 Pochodne względem danych wejściowych

Mając na uwadze charakterystykę problemu, jakim jest rozwiązywanie równań różniczkowych, szczególnie istotne może okazać się zbadanie kolejnych pochodnych wyników modeli względem danych wejściowych. Analiza ta zostanie przeprowadzona dla dwóch najlepszych sieci neuronowych omówionych w danym podrozdziale. Kluczowym elementem tej sekcji będzie porównanie wpływu zastosowanych funkcji aktywacyjnych na obliczane wyrażenia.

Wiemy, że rozwiązaniem równania jest funkcja $e^{-t}$. Wartości bezwzględne jej kolejnych pochodnych zachowują tę samą postać. Dlatego w analizie uwzględnione zostaną wyrażenia $\hat{y}_\theta$, $|\hat{y'}_\theta|$, $|\hat{y''}_\theta|$, ..., co pozwoli na ich wizualne porównanie i ocenę zgodności z oczekiwanym rozwiązaniem.

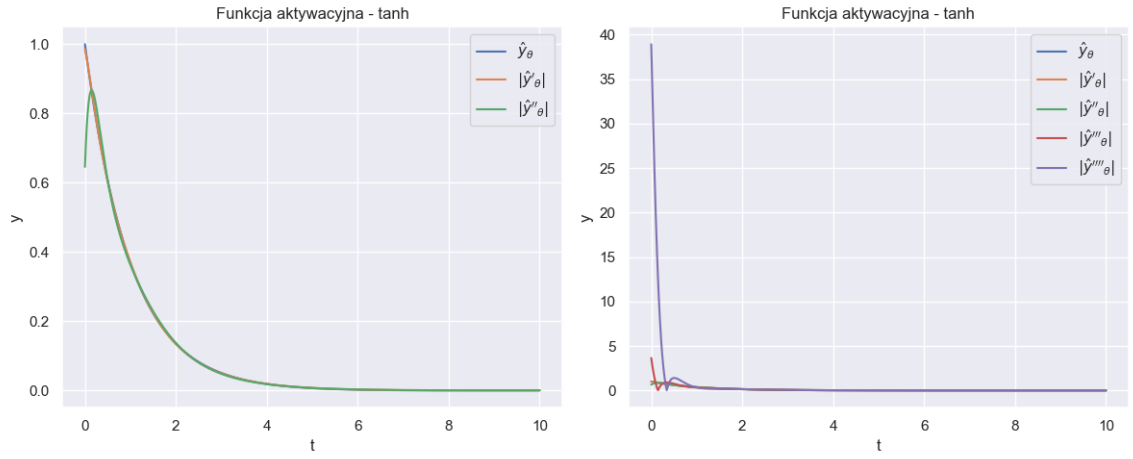
#### Tangens hiperboliczny

Wyrażenia zawierające kolejne pochodne modelu z funkcją aktywacyjną tangensa hiperbolicznego zostały przedstawione na Rysunku 3.9. Pochodne te zachowują ciągłość, a wartości pierwszych z nich są zbliżone do ich analitycznych odpowiedników. Jednak począwszy od drugiej pochodnej, można zaobserwować błędy numeryczne w pobliżu argumentu równego zero. Błędy te nasilają się w kolejnych wynikach, prowadząc do ich znaczącej akumulacji.

#### Funkcja ReLU

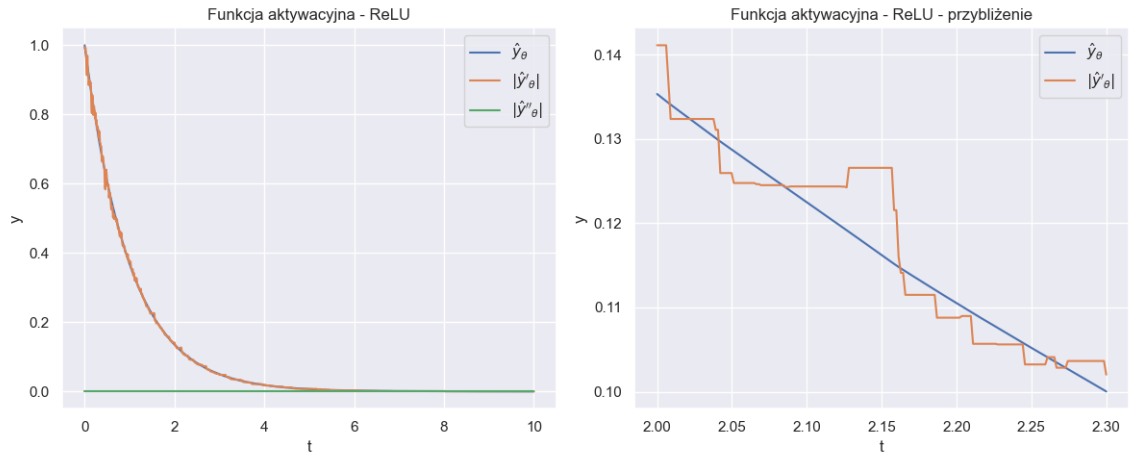
Podobne wykresy dla modelu z funkcją aktywacyjną ReLU przedstawiono na Rysunku 3.10. Od razu można zauważyć znaczące różnice, zgodne z wcześniejszymi oczekiwaniami. Pierwsza pochodna wyraźnie nie jest funkcją gładką i jest stała na pewnych pododcinkach, co odpowiada właściwościom pierwszej pochodnej funkcji ReLU. Druga pochodna również zachowuje się zgodnie z przewidywaniami, przyjmując wartość zerową dla wszystkich argumentów.

Co więcej, wykres ten częściowo wyjaśnia, dlaczego modele z ReLU wymagały znacznie większej liczby neuronów. Było to konieczne, aby jak najlepiej przybliżyć wyniki modelu do funkcji gładkiej.




Rysunek 3.9: Pochodne modelu względem danych wejściowch - tanh

Analiza zachowania pochodnych wskazuje, że model z funkcją aktywacyjną ReLU nie nadaje się do rozwiązywania równań różniczkowych wyższych rzędów niż pierwszy.




Rysunek 3.10: Pochodne modelu względem danych wejściowch - ReLU

## 3.2 Równanie niejednorodne pierwszego rzędu

Rozważmy teraz równanie różniczkowe niejednorodne pierwszego rzędu:

$$\begin{cases} y' - \sin(t) + \frac{y}{8} = 0, \\ y(0) = 3. \end{cases} \tag{3.1}$$

Zacznijmy od rozwiązania analitycznego. Przekształćmy równanie do postaci:

$$y' + \frac{y}{8} = \sin(x).$$

Możemy zastosować metodę czynnika całkującego, który wynosić będzie:

$$\mu(x) = e^{\int \frac{1}{8} dx} = e^{\frac{x}{8}}.$$

Wymnażając, oraz stosując kolejne przekształcenia, otrzymujemy:

$$e^{\frac{x}{8}}y' + e^{\frac{x}{8}}\frac{y}{8} = e^{\frac{x}{8}}\sin(x),$$

$$\frac{d}{dx}\left(e^{\frac{x}{8}}y\right) = e^{\frac{x}{8}}\sin(x),$$

$$e^{\frac{x}{8}}y = \int e^{\frac{x}{8}}\sin(x)\,dx + C. \tag{3.2}$$

Rozwiążmy teraz całkę znajdującą się po prawej stronie równania. Stosując całkowanie przez części, otrzymujemy:

$$\int e^{\frac{x}{8}}\sin(x)dx = 8e^{\frac{x}{8}}\sin(x) - 8\int e^{\frac{x}{8}}\cos(x)dx.$$

Adekwatnie rozwiązujemy kolejną pojawiającą się całkę

$$\int e^{\frac{x}{8}}\cos(x)dx = 8e^{\frac{x}{8}}\cos(x) + 8\int e^{\frac{x}{8}}\sin(x)dx,$$

a następnie podstawiając, mamy:

$$\int e^{\frac{x}{8}}\sin(x)dx = 8e^{\frac{x}{8}}\sin(x) - 8\Big(8e^{\frac{x}{8}}\cos(x) + 8\int e^{\frac{x}{8}}\sin(x)dx\Big).$$

Wyrażenie to po przekształceniach zredukuje się do postaci

$$\int e^{\frac{x}{8}}\sin(x)dx = \frac{8}{65}e^{\frac{x}{8}}\sin(x) - \frac{64}{65}e^{\frac{x}{8}}\cos(x). \tag{3.3}$$

Podstawiając (3.3) do równania (3.2) otrzymujemy rozwiązanie ogólne:

$$y(x) = \frac{8}{65}\sin(x) - \frac{64}{65}\cos(x) + Ce^{-\frac{x}{8}}$$

Które po uwzględnieniu warunku początkowego $y(0) = 3$ , przyjmuje ostateczną postać

$$y(x) = \frac{8}{65}\sin(x) - \frac{64}{65}\cos(x) + 3\frac{64}{65}e^{-\frac{x}{8}}.$$

Przejdźmy teraz do konstrukcji funkcji straty. Na podstawie równania (3.1) otrzymujemy:

$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=0}^{n-1}\Big(\hat{y}'_\theta(t_i) - \sin(t_i) + \frac{\hat{y}_\theta(t_i)}{8}\Big)^2 + \Big(\hat{y}_\theta(0) - 3\Big)^2.$$

Podobnie jak w poprzednim podroździale, przeanalizujemy cztery różne architektury sieci neuronowych, aby porównać skuteczność w aproksymacji rozwiązania równania.

### 3.2.1 Trening sieci neuronowych

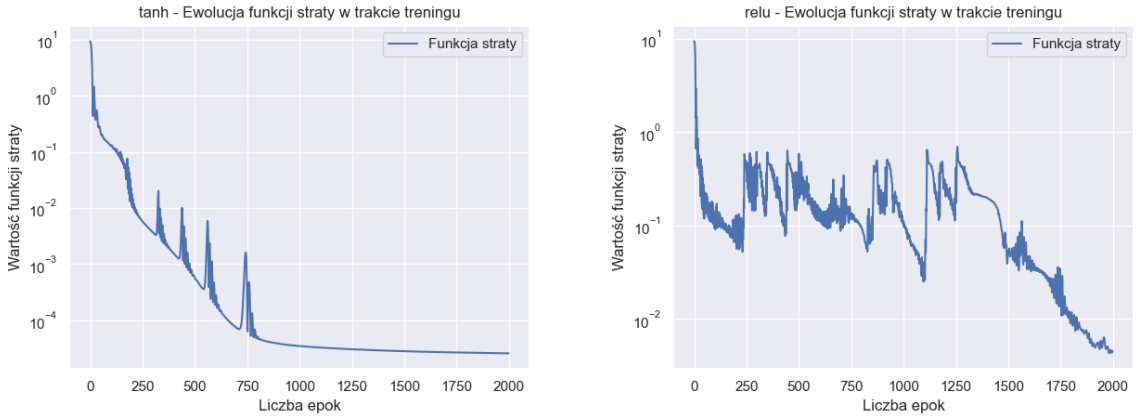
#### Cosine Decay

Wyniki optymalizacji hiperparametrów dla harmonogramu Cosine Decay oraz tych samych funkcji aktywacyjnych, co w poprzednich przykładach, przedstawiono w Tabeli 3.6.

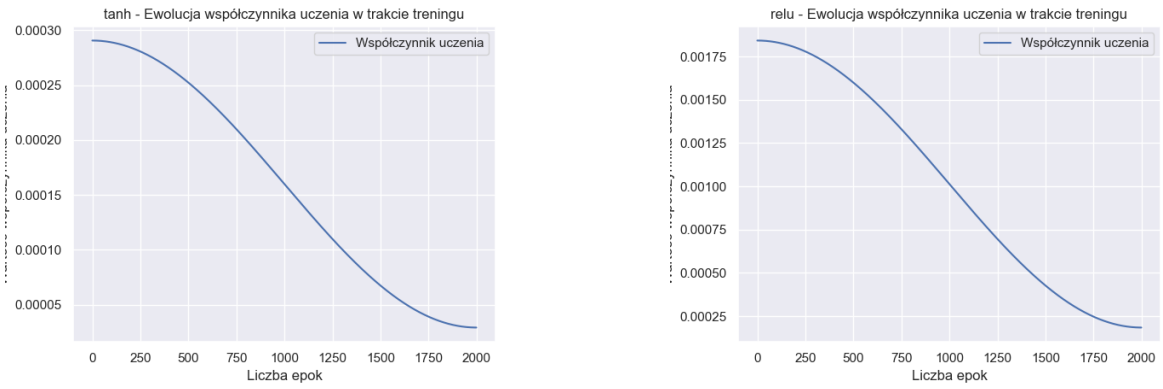
Tabela 3.6: Otrzymane hiperparametry - Cosine Decay

| Funkcja aktywacyjna | Liczba warstw ukrytych | Liczba neuronów w każdej z warstw ukrytych | Współczynnik uczenia |
|---|---|---|---|
| tanh | 14 | 96 | $\approx 0.0003$ |
| ReLU | 14 | 185 | $\approx 0.0018$ |

*Źródło: opracowanie własne*




Rysunek 3.11: Historia wartości funkcji straty - Cosine Decay

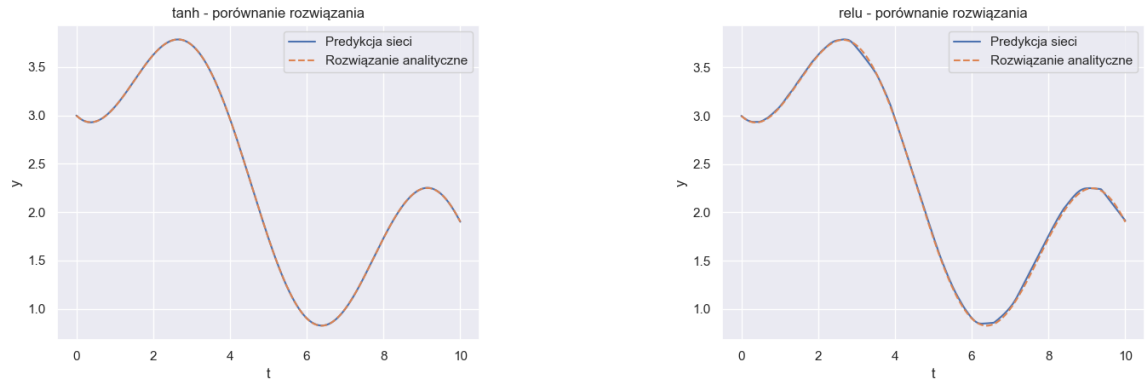



Rysunek 3.12: Historia wartości współczynnika uczenia - Cosine Decay

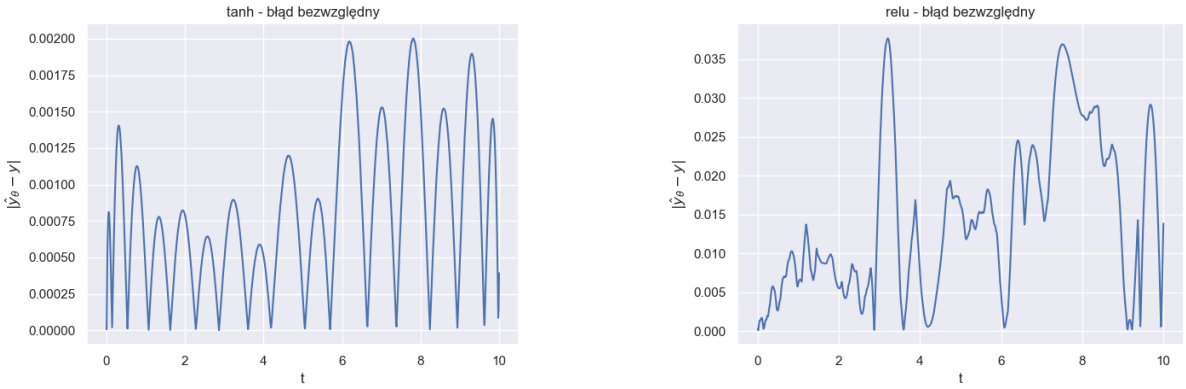
Ponownie liczba neuronów jest większa dla modelu z funkcją ReLU, tym razem jednak współczynnik uczenia znacznie się różni.

Kolejne wykresy ilustrujące przebieg treningu poszczególnych modeli przedstawiono na Rysunkach 3.11, 3.12, 3.13 oraz 3.14.

W tym przypadku model z funkcją aktywacyjną ReLU osiągnął znacznie gorsze wyniki, co zauważalne jest już na wykresie porównującym go z rozwiązaniem analitycznym, znajdującym się na Rysunku 3.13.




Rysunek 3.13: Porównanie predykcji sieci z rozwiązaniem analitycznym - Cosine Decay




Rysunek 3.14: Błąd bezwzględny rozwiązania - Cosine Decay

### ReduceLROnPlateau

Optymalizacja hiperparametrów oraz trening docelowych modeli zostały również przeprowadzone dla sieci neuronowych z harmonogramem ReduceLROnPlateau. Ich wyniki zostały przedstawione w Tabeli 3.7 oraz na Rysunkach 3.15, 3.16, 3.17 i 3.18.

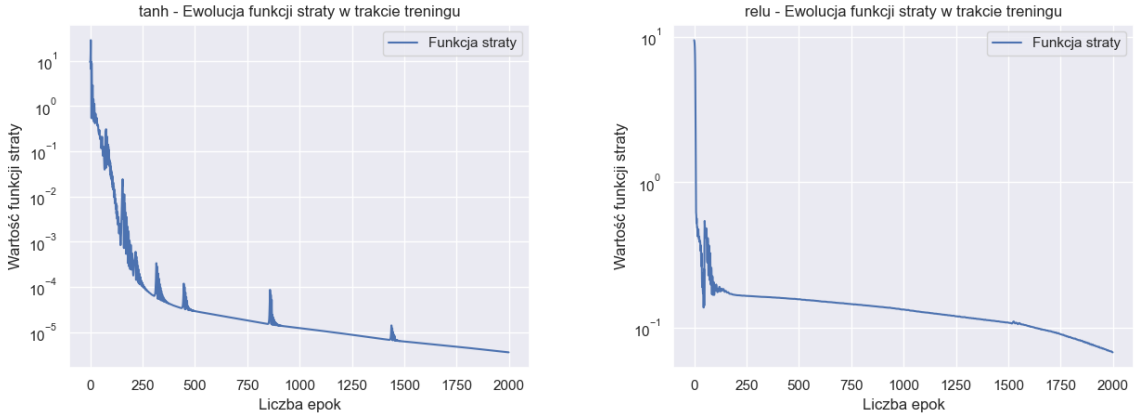
Tabela 3.7: Otrzymane hiperparametry - ReduceLROnPlateau

| Funkcja aktywacyjna | Liczba warstw ukrytych | Liczba neuronów w każdej z warstw ukrytych | Współczynnik uczenia |
|---|---|---|---|
| tanh | 5 | 199 | $\approx 0.0099$ |
| ReLU | 11 | 93 | $\approx 0.0043$ |

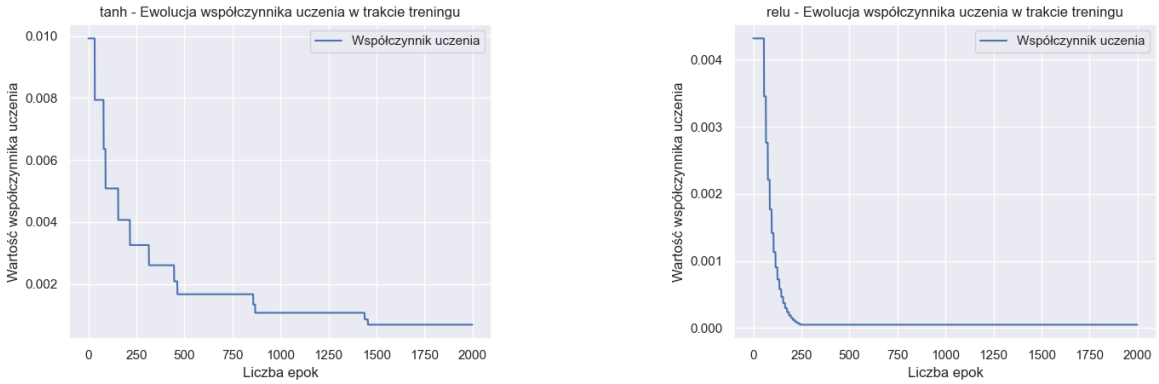
*Źródło: opracowanie własne*

Możemy zaobserwować, że wartości współczynników uczenia są znacznie wyższe w porównaniu z sieciami neuronowymi wykorzystującymi harmonogram Cosine Decay. Jest to związane z faktem, że algorytm ReduceLROnPlateau w dynamiczny i bardziej zdecydowany sposób dostosowuje ten hiperparametr. Umożliwia to szybszą zbieżność w początkowej fazie treningu, dzięki przyjęciu wyższych wartości współczynnika uczenia, a także dynamiczne

obniżanie jego wartości w celu zapewnienia stabilności w późniejszych etapach. Opisane zjawisko możemy zauważyć na Rysunku 3.16.




Rysunek 3.15: Historia wartości funkcji straty - ReduceLROnPlateau



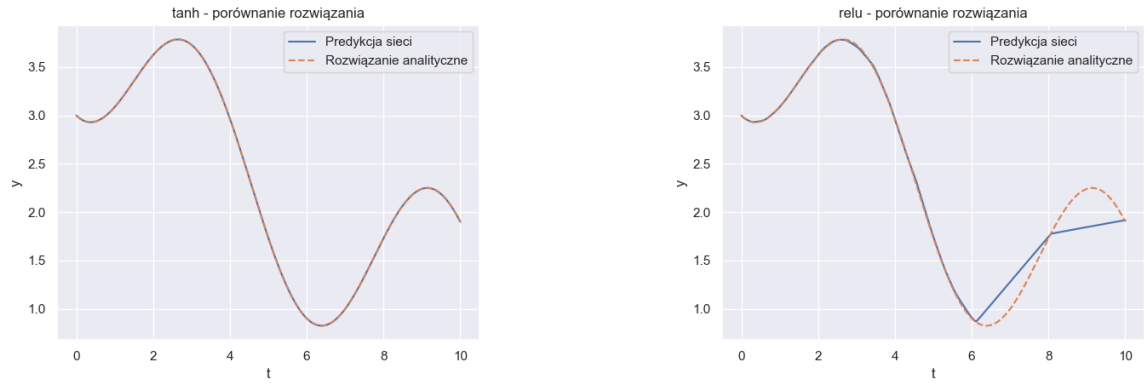

Rysunek 3.16: Historia wartości współczynnika uczenia - ReduceLROnPlateau

Model z funkcją aktywacyjną ReLU ponownie osiągnął znacznie gorsze wyniki. Różnica między predykcją sieci neuronowej a rozwiązaniem analitycznym jest szczególnie wyraźna dla danych wejściowych w przedziale $[6, 10]$ na Rysunku 3.17.

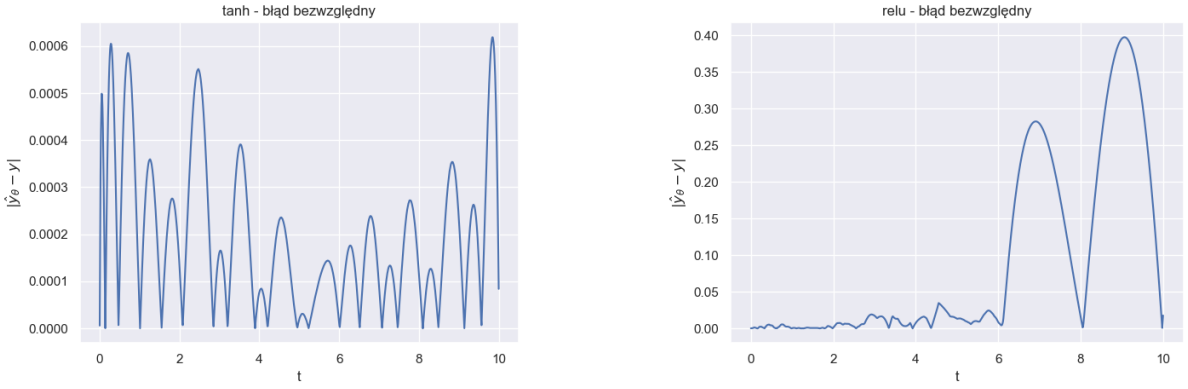
### 3.2.2 Ewaluacja modeli

Dane przedstawione w Tabeli 3.8 potwierdzają wcześniejsze obserwacje, że funkcja aktywacyjna tangensa hiperbolicznego przewyższa ReLU w danym problemie.

Na podstawie uzyskanych jak dotąd wyników dalsza analiza w niniejszej pracy zostanie skoncentrowana wyłącznie na funkcji aktywacyjnej tangensa hiperbolicznego, jako bardziej efektywnej i stabilnej w kontekście rozwiązywania równań różniczkowych.




Rysunek 3.17: Porównanie predykcji sieci z rozwiązaniem analitycznym - ReduceLROnPlateau




Rysunek 3.18: Błąd bezwzględny rozwiązania - ReduceLROnPlateau

Tabela 3.8: Sieci neuronowe - porównanie wyników

| Harmonogram współczynnika uczenia | Funkcja aktywacyjna | Błąd średnio-kwadratowy | Średni błąd bezwzględny | Czas treningu | Liczba neuronów ukrytych |
|---|---|---|---|---|---|
| CD | tanh | $9.35 \cdot 10^{-7}$ | $8.15 \cdot 10^{-4}$ | 61.59s | 1344 |
| CD | ReLU | $3.09 \cdot 10^{-4}$ | $1.44 \cdot 10^{-2}$ | 101.87s | 2590 |
| RLROP | tanh | $6.07 \cdot 10^{-8}$ | $1.94 \cdot 10^{-4}$ | 41.32s | 995 |
| RLROP | ReLU | $2.34 \cdot 10^{-2}$ | $8.96 \cdot 10^{-2}$ | 44.06s | 1023 |

CD - Cosine Decay, RLROP - ReduceLROnPlateau

*Źródło: opracowanie własne*

## 3.3 Równanie jednorodne drugiego rzędu

Jako ostatni przykład w tym rozdziale, poddamy analizie równanie:

$$\begin{cases} y'' + y = 0, \\ y(0) = 1, \\ y'(0) = 1. \end{cases}$$

Jest to równanie jednorodne drugiego rzędu o stałych współczynnikach. Możemy więc rozpatrzyć jego wielomian charakterystyczny:

$$r^2 + 1 = 0.$$

Pierwiastkami tego wielomianu są liczby urojone $r_{1,2} = \pm i$, a więc rozwiązaniem ogólnym naszego równania różniczkowego będzie suma funkcji sinus i cosinus:

$$y = C_1 \sin(t) + C_2 \cos(t).$$

Uwzględniając warunki początkowe $y(0) = 1$, $y'(0) = 1$, otrzymujemy rozwiązanie szczególne:

$$y = \sin(t) + \cos(t).$$

Adekwatnie, odpowiednio skonstruowana funkcja straty przyjmie postać:

$$J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=0}^{n-1} \left( \hat{y}''_\theta(t_i) + \hat{y}_\theta(t_i) \right)^2 + \left( \hat{y}_\theta(0) - 1 \right)^2 + \left( \hat{y}'_\theta(0) - 1 \right)^2.$$

Uwzględniając wnioski z wcześniejszych podrozdziałów, przykład ten rozwiążemy wyłącznie dla funkcji aktywacyjnej **tangensa hiperbolicznego**, mając na uwadze nie tylko uzyskaną dokładność, lecz także niezerową wartość drugiej pochodnej, istotnej w danym równaniu.

### 3.3.1 Trening sieci neuronowych

Hiperparametry otrzymane w procesie optymalizacji przedstawione zostały w Tabeli 3.9. Ponownie, harmonogram ReduceLROnPlateau pozwolił na rozpoczęcie treningu z wyższą wartością współczynnika uczenia.

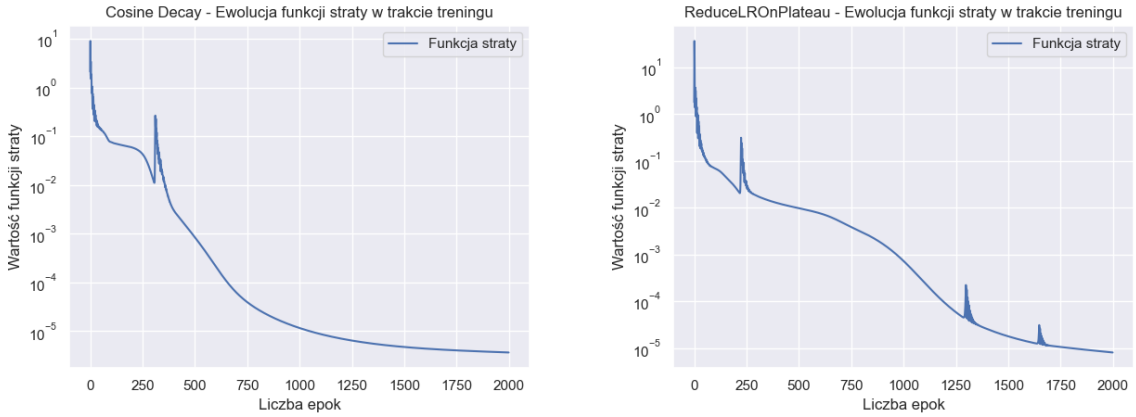
Tabela 3.9: Otrzymane hiperparametry - tangens hiperboliczny

| Harmonogram współczynnika uczenia | Liczba warstw ukrytych | Liczba neuronów w każdej z warstw ukrytych | Współczynnik uczenia |
|---|---|---|---|
| CD | 7 | 120 | $\approx 0.0026$ |
| RLROP | 10 | 106 | $\approx 0.0045$ |

CD - Cosine Decay, RLROP - ReduceLROnPlateau

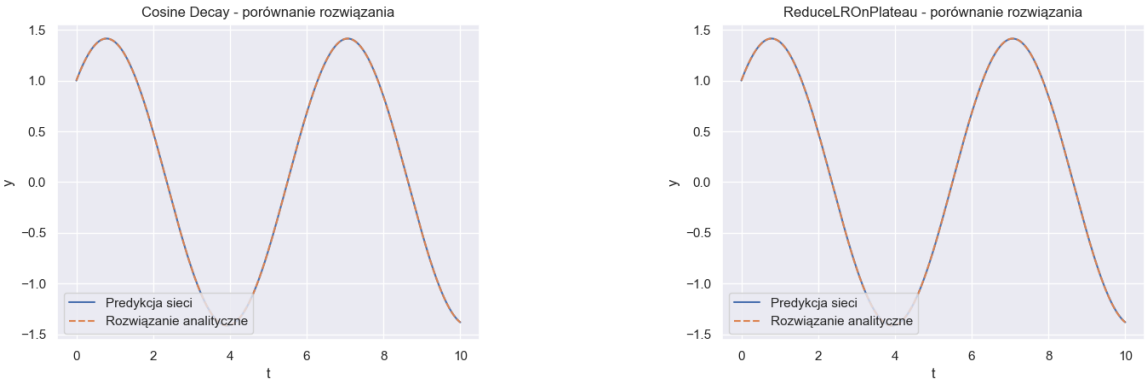
*Źródło: opracowanie własne*

Wykresy związane z procesem treningu sieci neuronowych zostały przedstawione na Rysunkach 3.19, 3.20, 3.21 oraz 3.22.



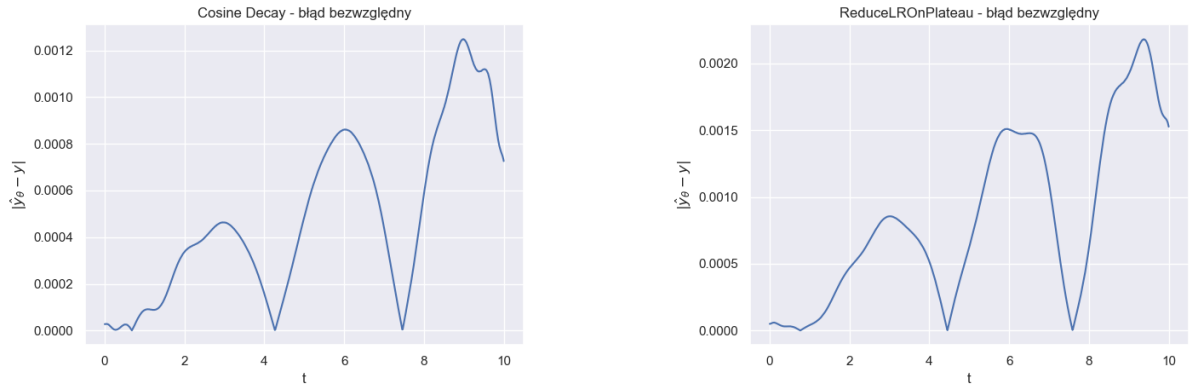

Rysunek 3.19: Historia wartości funkcji straty - tangens hiperboliczny


Rysunek 3.20: Historia wartości współczynnika uczenia - tangens hiperboliczny




Rysunek 3.21: Porównanie predykcji sieci z rozwiązaniem analitycznym - tangens hiperboliczny

Analiza Rysunku 3.22 wskazuje, że maksymalny błąd bezwzględny wzrasta wraz z wielkością danych wejściowych. Jednocześnie można zauważyć, że wartości tego błędu są najmniejsze w obszarach, gdzie rozwiązanie równania charakteryzuje się najmniejszą




Rysunek 3.22: Błąd bezwzględny rozwiązania - tangens hiperboliczny

zmiennością, czyli w pobliżu wierzchołków fali. Natomiast największe wartości błędu występują tam, gdzie rozwiązanie jest najbardziej zmienne, a zatem w obszarach między wierzchołkami, gdzie wartość bezwzględna pochodnej rozwiązania osiąga swoje maksimum.

Podobne zachowania można zaobserwować na Rysunkach 3.4 oraz 3.14 dla funkcji aktywacyjnej ReLU, gdzie także występuje wyraźna zależność pomiędzy zmiennością rozwiązania a wartością błędu bezwzględnego. Należy jednak zauważyć, że analiza pozostałych wykresów dotyczących omawianych zmiennych wskazuje na brak jednoznacznej powtarzalności tej zależności, co nie wyklucza, że może ona wynikać z przypadkowych czynników, a nie z ogólnej charakterystyki badanego zjawiska.

### 3.3.2 Ewaluacja modeli

Końcowe wyniki uzyskanej dokładności rozwiązań umieszczone zostały w Tabeli 3.10. Tym razem harmonogram Cosine Decay okazał się być lepszy w każdej z rozważanych kategorii.

Tabela 3.10: Sieci neuronowe - porównanie wyników

| Harmonogram współczynnika uczenia | Funkcja aktywacyjna | Błąd średnio-kwadratowy | Średni błąd bezwzględny | Czas treningu | Liczba neuronów ukrytych |
|---|---|---|---|---|---|
| CD | tanh | $3.64 \cdot 10^{-7}$ | $4.86 \cdot 10^{-4}$ | 46.49s | 840 |
| RLROP | tanh | $1.12 \cdot 10^{-6}$ | $8.42 \cdot 10^{-4}$ | 59.97s | 1060 |

CD - Cosine Decay, RLROP - ReduceLROnPlateau

*Źródło: opracowanie własne*

## 3.4 Porównanie z metodami numerycznymi

Jako ostatni element tego rozdziału przeprowadzimy porównanie wydajności sieci neuronowych z metodami numerycznymi. Analiza zostanie przeprowadzona dla modelu opisanego w Podrozdziale 3.1, wykorzystującego harmonogram Cosine Decay oraz funkcję aktywacyjną tangensa hiperbolicznego. W ramach badania przeprowadzimy symulację 100 różnych

inicjalizacji i treningów danej sieci neuronowej. Wyniki tych eksperymentów zostaną uśrednione i porównane z wynikami uzyskanymi przy pomocy metody Rungego-Kutty czwartego rzędu (RK4) [3] dla 10 000 prób.

Metoda RK4 jest popularną metodą numeryczną stosowaną do rozwiązywania równań różniczkowych zwyczajnych zadanych postacią

$$\begin{cases} \frac{dy}{dt} = f(y,t), \\ y(t_0) = y_0. \end{cases}$$

Wyróżnia się ona dobrą równowagą pomiędzy dokładnością a złożonością obliczeniową, zapewniając czwarty rząd dokładności. Jej algorytm polega na iteracyjnym obliczaniu przybliżonego rozwiązania, przy użyciu następujących wzorów:

$$\begin{aligned} y_{n+1} &= y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4), \\ k_1 &= f(t_n, y_n), \\ k_2 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right), \\ k_3 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right), \\ k_4 &= f(t_n + h, y_n + hk_3), \end{aligned}$$

gdzie $h$ jest krokiem czasowym, $t_n$ to punkt czasowy, $y_n$ to przybliżona wartość rozwiązania w punkcie, a następujący punkt czasowy obliczany jest jako $t_{n+1} = t_n + h$.

Symulacje przeprowadzimy na tej samej siatce czasowej, używanej przez nas do treningu sieci neuronowych. Był to przedział $[0, 10]$ z tysiącem równo odległych od siebie punktów, a więc przyjętym przez nas krokiem czasowym będzie $h = 0.01$.

## Wyniki

Przybliżone wyniki porównania przedstawione zostały w Tabeli 3.11. Wskazują one na znaczące różnice w czasie wykonania oraz dokładności pomiędzy metodą RK4 a siecią neuronową.

Tabela 3.11: Porównanie z metodą RK4

| Metoda | Średni czas wykonania | Odchylenie standardowe czasu wykonania | Średnia wartość błędu średnio-kwadratowego | Odchylenie standardowe błędu średniokwadratowego |
|---|---|---|---|---|
| Sieć neuronowa | 29.31s | 1.35s | $1.85 \cdot 10^{-7}$ | $1.79 \cdot 10^{-7}$ |
| RK4 | 0.0025s | 0.0016s | $1.76 \cdot 10^{-22}$ | $4.70 \cdot 10^{-38}$ |

*Źródło: opracowanie własne*

Średni czas wykonania okazał się być ponad dziesięć tysięcy razy krótszy dla metody Rungego-Kutty niż sieci neuronowej. Jest to ogromna różnica, jednoznacznie przemawiająca na korzyść metody numerycznej. Odchylenia standardowe czasu wykonania dla obu metod zachowują proporcje podobne do średnich wartości, co wskazuje na spójność wyników w tym kontekście.

Jeszcze większe różnice, niż w przypadku czasu wykonania, można zaobserwować w odniesieniu do błędu średniokwadratowego. Szczególną uwagę zwraca wyjątkowo wysoka wartość odchylenia standardowego dla sieci neuronowej, co świadczy o znacznej zmienności wyników. Najprawdopodobniej jest to konsekwencja losowej inicjalizacji parametrów podczas tworzenia modelu, która może wpływać na późniejszą zbieżność.

W przeciwieństwie do sieci neuronowej, metoda RK4 jest metodą deterministyczną, przez co oczekiwana przez nas wartość odchylenia standardowego błędu średniokwadratowego powinna być równa zeru. Jej niezerowa wartość wynika prawdopodobnie z numerycznych błędów zaokrągleń, aczkolwiek nadal jest to wartość zdecydowanie zbliżona do zera.

Podsumowując, metoda RK4 przewyższa sieć neuronową zarówno pod względem czasu wykonania, jak i dokładności. Niemniej jednak, sieć neuronowa może okazać się bardziej elastyczna w modelowaniu skomplikowanych zjawisk, gdzie analityczne lub numeryczne metody są trudne do zastosowania.

# Rozdział 4

# Układy równań różniczkowych zwyczajnych

W poprzednim rozdziale udało nam się uzyskać przybliżenia rozwiązań pojedynczych równań różniczkowych za pomocą sieci neuronowych. Jednakże, jak wykazało porównanie, nie są one w stanie konkurować z tradycyjnymi metodami numerycznymi. Istnieje jednak jeszcze jeden przypadek, w którym jest szansa, że sieć neuronowa może okazać się lepszym wyborem - są nimi układy równań różniczkowych. Głębokie sieci neuronowe uzyskują zaskakująco dobre wyniki w zadaniach o charakterze wielowymiarowym, co poruszane jest w wielu pracach naukowych [19, 20]. Często pojawiającym się zagadnieniem jest tzw. „Przekleństwo Wielowymiarowości“ (ang. *Curse of Dimensionality*), które polega na gwałtownym wzroście złożoności obliczeniowej w miarę wzrostu liczby wymiarów w analizowanych problemach. Problem ten postaram się przeanalizować dla rozważanej przeze mnie architektury perceptronu wielowarstwowego.

## 4.1 Charakterystyka problemu

Dobrym przykładem rozważanego zagadnienia jest układ równań różniczkowych postaci:

$$\begin{cases} \mathbf{y}' = A\mathbf{y}, \\ \mathbf{y}(0) = \mathbf{y}_0, \end{cases}$$

gdzie

$$\mathbf{y} = \begin{bmatrix} y_0(t) \\ y_1(t) \\ \vdots \\ y_{m-1}(t) \end{bmatrix}, \mathbf{y}_0 = \begin{bmatrix} y_{00} \\ y_{01} \\ \vdots \\ y_{0m-1} \end{bmatrix},$$

a $A$ jest kwadratową macierzą stopnia $m$.

Jednakże, w celu przeprowadzenia symulacji, postanowiłem przyjąć za $A$ ujemną macierz jednostkową. Pozwoli to przede wszystkim na dynamiczne generowanie układów równań wraz z ich analitycznymi rozwiązaniami, które niezbędne będą przy ocenie dokładności predykcji sieci. Równanie więc sprowadza się do postaci:

$$\begin{cases} \mathbf{y}' = -I\mathbf{y}, \\ \mathbf{y}(0) = \mathbf{y}_0, \end{cases} \tag{4.1}$$

gdzie za warunki początkowe postanowiłem przyjąć wartości

$$\mathbf{y}_0 = \begin{bmatrix} 0.2 \\ 0.2 + 1.8\frac{1}{m-1} \\ 0.2 + 1.8\frac{2}{m-1} \\ \vdots \\ 2 \end{bmatrix}.$$

Problem w tym momencie redukuje się do $m$ niezależnych od siebie równań różniczkowych, identycznych do równania przedstawionego w Rozdziale 3.1, różniących się jedynie warunkami początkowymi. Oczekiwany przez nas czas rozwiązania za pomocą metod numerycznych powinien rosnąć w sposób **liniowy** wraz ze wzrostem liczby równań.

W tym momencie naszym głównym celem będzie zbadanie tempa wzrostu czasu obliczeń dla sieci neuronowych w zależności od ilości równań w zadanych układach.

## 4.2 Konstrukcja sieci neuronowej

Aby móc rozważać rozwiązywanie układów równań różniczkowych, musimy dokonać pewnych zmian w konstrukcji naszej sieci neuronowej. Po pierwsze, każdy neuron warstwy wyjściowej, zamiast pojedynczej wartości, zwracać będzie wektor o długości $m$, którego elementy będą odpowiadać wartościom każdej kolejnej z poszukiwanej przez nas funkcji w danym punkcie. Następnie, uwzględnić musimy również funkcję straty, która przyjmie postać:

$$J(\theta) = \sum_{k=0}^{m-1} \left( \frac{1}{n} \sum_{i=0}^{n-1} \left( \hat{y}'_{k_\theta}(t_i) - \hat{y}_{k_\theta}(t_i) \right)^2 + \left( \hat{y}_{k_\theta}(t_i) - y_{0k} \right)^2 \right).$$

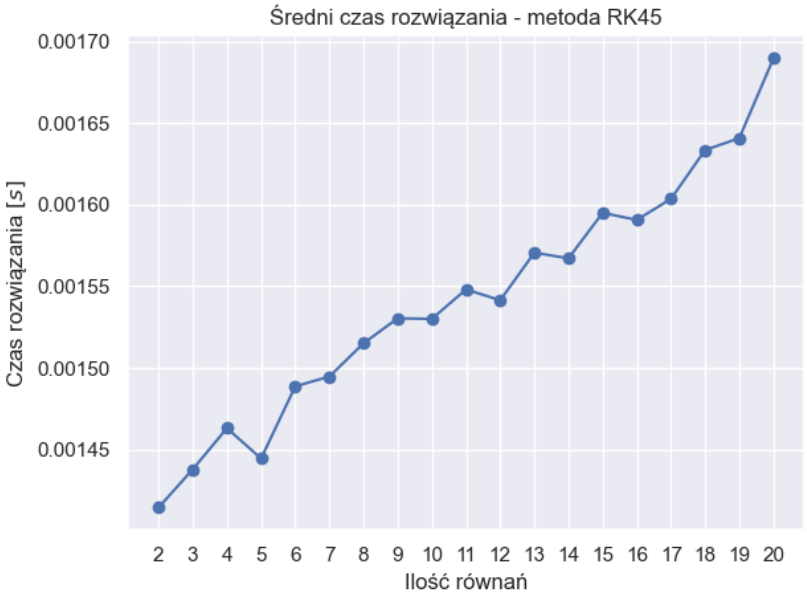
## 4.3 Wyniki

Symulacje w tej sekcji przeprowadzone zostały dla układu równań (4.1), dla liczby równań $m \in \{2, 3, \ldots, 20\}$.

Pierwsza symulacja przeprowadzona została z wykorzystaniem adaptacyjnej metody Rungego-Kutty piątego rzędu (RK45) [7], która wykorzystuje wbudowaną formułę czwartego rzędu do szacowania lokalnego błędu. Jest to metoda w dynamiczny sposób dostosowująca krok czasowy. Jej implementacja w bibliotece SciPy [29] pozwala na rozwiązywanie układów równań różniczkowych. Każdy z układów rozwiązywany był łącznie 10 000 razy, a uśrednione wyniki czasu wykonania przedstawione zostały na Rysunku 4.1.

Tak jak oczekiwaliśmy, średni czas obliczeń dla metody RK45, w przybliżeniu, rośnie w sposób liniowy wraz z liczbą równań.

Symulacja dla sieci neuronowej przeprowadzona została łącznie 25 razy dla każdego z układów równań. Głównym zadaniem sieci neuronowej w każdym procesie uczenia było uzyskanie wartości błędu średniokwadratowego nie większej niż $10^{-4}$, względem rozwiązania analitycznego układu równań. Czas potrzebny do osiągnięcia tego założenia był docelowym, zapisywanym czasem pojedynczej realizacji. Uśrednione rezultaty symulacji przedstawione zostały na Rysunku 4.2. Wykorzystany został model z funkcją aktywacyjną tangensa hiperbolicznego oraz harmonogramem uczenia ReduceLROnPlateau, gdyż jest on lepszym wyborem dla treningu o nieustalonej z góry liczbie epok. Hiperparametry miały stałą wartość we wszystkich przypadkach, aby umożliwić łatwiejsze porównanie wyników.




Rysunek 4.1: Średni czas obliczeń względem ilości równań


Rysunek 4.2: Średni czas treningu względem ilości równań

Okazuje się, że dla sieci neuronowej tempo wzrostu czasu rozwiązania wraz z ilością równań jest szybsze niż liniowe, występujące w przypadku metod numerycznych w danym problemie. Ponownie, sieć neuronowa poradziła sobie z powierzonym jej zadaniem, jakim była aproksymacja rozwiązania układu równań różniczkowych, jednakże uzyskane wyniki są dalekie od dokładności i szybkości, jakie oferują metody numeryczne.

# Podsumowanie

Udało nam się osiągnąć główny cel niniejszej pracy, jakim była aproksymacja rozwiązań równań różniczkowych przez sieci neuronowe. Algorytmy te okazały się być bardzo elastyczne, radząc sobie z równaniami zarówno pierwszego, jak i drugiego rzędu, oraz z układami równań. Co prawda, uzyskane wyniki znacząco odbiegały od tradycyjnych metod numerycznych pod względem dokładności oraz szybkości obliczeń. Jednakże istnieje jeszcze wiele innych rodzajów sieci neuronowych oraz sposobów na zwiększenie ich wydajności, które warto byłoby poddać analizie.

Zauważalny na co dzień prężny rozwój uczenia głębokiego jest tylko kolejną zachętą ku temu, aby starać się jeszcze bardziej zagłębić w poruszaną problematykę. Wiąże się z tym nadzieja, że sieci neuronowe będą w stanie w pozytywny sposób przyczynić się do analizy jakże ważnych modeli matematycznych, opisujących nasz wszechświat i będących jego nieodzowną częścią, jakimi są równania różniczkowe.

# Bibliografia


[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Akiba, T., Sano, S., Yanase, T., Ohta, T., Koyama, M. Optuna: A next-generation hyperparameter optimization framework, 2019.

[3] Banu, M. S. A comparative study on classical fourth order and butcher sixth order runge-kutta methods with initial and boundary value problems. *International Journal of Material and Mathematical Sciences* (2021).

[4] Baty, H., Baty, L. Solving differential equations using physics informed deep learning: a hand-on tutorial with benchmark tests, 2023.

[5] Baydin, A. G., Pearlmutter, B. A., Radul, A. A., Siskind, J. M. Automatic differentiation in machine learning: a survey, 2018.

[6] Cai, S., Wang, Z., Wang, S., Perdikaris, P., Karniadakis, G. E. Physics-informed neural networks for heat transfer problems. *Journal of Heat Transfer 143*, 6 (04 2021), 060801.

[7] Dormand, J., Prince, P. A family of embedded runge-kutta formulae. *Journal of Computational and Applied Mathematics 6*, 1 (1980), 19–26.

[8] Duchi, J., Hazan, E., Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research 12* (07 2011), 2121–2159.

[9] Erden, C., Demir, H. I., Kökçam, A. H. Enhancing machine learning model performance with hyper parameter optimization: A comparative study, 2023.

[10] Es'kin, V. A., Davydov, D. V., Egorova, E. D., Malkhanov, A. O., Akhukov, M. A., Smorkalov, M. E. About optimal loss function for training physics-informed neural networks under respecting causality, 2023.

[11] Goodfellow, I., Bengio, Y., Courville, A. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[12] Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., Oliphant, T. E. Array programming with NumPy. *Nature 585*, 7825 (Sept. 2020), 357–362.

[13] Jagtap, A. D., Kharazmi, E., Karniadakis, G. E. Conservative physics-informed neural networks on discrete domains for conservation laws: Applications to forward and inverse problems. *Computer Methods in Applied Mechanics and Engineering 365* (2020), 113028.

[14] Karlsson Faronius, H. Solving partial differential equations with neural networks. Master's thesis, Uppsala University, Department of Mathematics, 2023.

[15] Kingma, D. P., Ba, J. Adam: A method for stochastic optimization, 2017.

[16] Lagaris, I., Likas, A., Fotiadis, D. Artificial neural networks for solving ordinary and partial differential equations. *IEEE Transactions on Neural Networks 9*, 5 (1998), 987–1000.

[17] Lederer, J. Activation functions in artificial neural networks: A systematic overview, 2021.

[18] Loshchilov, I., Hutter, F. Sgdr: Stochastic gradient descent with warm restarts, 2017.

[19] Poggio, T., Liao, Q. Theory i: Deep networks and the curse of dimensionality. *Bulletin of the Polish Academy of Sciences: Technical Sciences 66* (12 2018), 761–773.

[20] Poggio, T., Mhaskar, H., Rosasco, L., Miranda, B., Liao, Q. Why and when can deep – but not shallow – networks avoid the curse of dimensionality: a review, 2017.

[21] Raissi, M., Perdikaris, P., Karniadakis, G. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics 378* (2019), 686–707.

[22] Raissi, M., Perdikaris, P., Karniadakis, G. E. Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations, 2017.

[23] Ruder, S. An overview of gradient descent optimization algorithms, 2017.

[24] Rumelhart, D. E., Hinton, G. E., Williams, R. J. Learning representations by back-propagating errors. *nature 323*, 6088 (1986), 533–536.

[25] TensorFlow Authors. Cosinedecay schedule - tensorflow v2.16.1 api documentation, 2025. `https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/schedules/CosineDecay` Accessed: 2025-01-01.

[26] TensorFlow Authors. Reducelronplateau - tensorflow v2.16.1 api documentation, 2025. `https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ReduceLROnPlateau` Accessed: 2025-01-01.

[27] Tieleman, T., Hinton, G. E. Lecture 6a: Rmsprop and adagrad. University of Toronto, 2012. Accessed: 2024-12-09.

[28] Verma, A. An introduction to automatic differentiation. *Current Science 78*, 7 (2000), 804–807.

[29] Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods 17* (2020), 261–272.

[30] Wang, Y. D., Blunt, M., Armstrong, R., Mostaghimi, P. Deep learning in pore scale imaging and modeling. *Earth-Science Reviews 215* (02 2021).

[31] Wu, Y., Liu, L., Bae, J., Chow, K.-H., Iyengar, A., Pu, C., Wei, W., Yu, L., Zhang, Q. Demystifying learning rate policies for high accuracy training of deep neural networks, 2019.