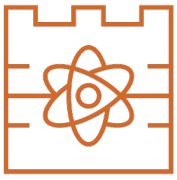
POLITECHNIKA KRAKOWSKA
im. Tadeusza Kościuszki
Wydział Inżynierii Materiałowej i Fizyki
Katedra Fizyki



Kierunek studiów: Fizyka Techniczna
Specjalność: Modelowanie Komputerowe

STUDIA STACJONARNE

PRACA DYPLOMOWA

MAGISTERSKA

Marcin Kubański

135385

Zastosowanie algorytmów sieci neuronowych do analizy zagrażających Ziemi asteroid

Application of neural network algorithms to analyze Earth-threatening asteroids

Promotor pracy dyplomowej:

dr Radosław Kycia

Recenzent pracy dyplomowej:

prof. dr hab. Włodzimierz Wójcik

Kraków, rok akad. 2023/24

Abstrakt

W pracy wybrano zbiór danych dotyczący asteroid zagrażających Ziemi, następnie poddano je analizie algorytmów sieci neuronowych (prosta sieć jednokierunkowa, rekurencyjna sieć neuronowa, multiperceptron wielowarstwowy oraz konwolucyjna sieć neuronowa). Sprawdzono jak bardzo sieci neuronowe są precyzyjne w przypadku wykrywania niebezpieczeństwa ze strony asteroid. W pracy przedstawiono również porównanie wyników otrzymanych w pracy inżynierskiej z tymi otrzymanymi w pracy magisterskiej przy użyciu metod uczenia głębokiego. Analizę przeprowadzono w oparciu o język Python z użyciem bibliotek TensorFlow, Keras, Pandas, Scikit-learn, NumPy. Niniejsza praca stanowi kontynuację pracy inżynierskiej, w której zastosowano identyczne dane, ale skupiono się na metodach statystycznego uczenia maszynowego.

Abstract

In this paper, a dataset on asteroids threatening the Earth was selected, then the data was analyzed by neural network algorithms (simple feed-forward network, recurrent neural network, multilayer perceptron and convolutional neural network). It was tested how accurate the neural networks are in the case of asteroid impact detection. The paper also presents a comparison of the results obtained in the engineering paper with those obtained in the master's thesis using deep learning methods. The analysis was carried out based on the Python language using TensorFlow, Keras, Pandas, Scikit-learn, NumPy libraries. The present work is a continuation of the engineering thesis, which used identical data, but focused on statistical machine learning methods.

Podziękowania

Pragnę wyrazić wdzięczność mojemu promotorowi, dr Radosławowi Kyci za poświęcony czas, wszelakie wskazówki oraz ogromny entuzjazm zarówno podczas pisania pracy magisterskiej jak i poprzednio, pracy inżynierskiej. Chciałbym również podziękować mojej mamie oraz mojej partnerce za okazane wsparcie podczas trwania studiów.

Spis treści

Wstęp

0.1 Cel pracy

Celem niniejszej pracy jest wykorzystanie metod uczenia głębokiego do analizy asteroid w celu zbadania predykcji zagrożeń ze strony asteroid dla Ziemi oraz porównania wyników z algorytmami uczenia maszynowego wykorzystanymi w pracy inżynierskiej. Zestaw danych zawiera parametry asteroid zebranych przez NASA, w ramach programu Sentry.

0.2 Zakres pracy

Praca została podzielona na dwie części - teoretyczną oraz praktyczną. W rozdziale teoretycznym opisano język, środowisko oraz biblioteki wykorzystywane w pracy. Przedstawiono krótką charakterystykę asteroid oraz wyjaśniono działanie systemu Sentry.

W dalszej części opisano definicje, zasady działania sieci neuronowych oraz ich algorytmy, które następnie zostały poddane w analizie. Ukazano również różnice pomiędzy uczeniem maszynowym a sieciami neuronowymi.

W części praktycznej przedstawione zostały wyniki analizy wybranych algorytmów sieci neuronowych, które następnie porównano z wynikami algorytmów uczenia maszynowego. W rozdziale końcowym, będącym podsumowaniem doświadczenia, przedstawiono otrzymane wnioski.

0.3 Metodyka pracy

Analizę danych wykonano w środowisku programistycznym Jupyter z wykorzystaniem języka programowania Python. Jako biblioteki wykorzystano: Tensorflow, Pandas, NumPy, Scikit-learn, Keras, Opendatasets - są to standardowe narzędzia używane przy analizie danych i uczeniu maszynowym.

Część I

Część Teoretyczna

Rozdział 1

Wykorzystane środowisko analizy danych

W dzisiejszych czasach mamy do dyspozycji wiele narzędzi do analizy danych, począwszy od prostych arkuszy kalkulacyjnych, takich jak Excel, aż po zaawansowane języki programowania, takie jak R (interpretowany język programowania oraz środowisko do obliczeń statystycznych i wizualizacji wyników). Wybór narzędzia zależy od wielu czynników, takich jak złożoność danych czy wymagane techniki analizy. Często wykorzystywanym narzędziem jest język Python.

1.1 Python

Python [2] jest rodzajem języka programowania wysokopoziomowego. Charakteryzuje się dużą czytelnością i prostotą. Jego funkcjonalność można rozszerzyć o dodatkowe biblioteki.

1.2 Jupyter

Jupyter [3] to środowisko programistyczne wykorzystujące interpreter IPython, czyli interaktywną wersję Pythona, które umożliwia pracę w przeglądarce internetowej. Działanie środowiska opiera się na uruchomieniu lokalnego serwera na komputerze i otwarciu notatnika w przeglądarce. Ta funkcjonalność pozwala na tzw. "narrację kodu" czyli łączenie kodu z opisami i elementami multimedialnymi, co sprawia, że możemy wykorzystać taki notatnik jako raport z analizy.

1.3 Wykorzystane biblioteki

- **TensorFlow** [4] jest biblioteką używaną w uczeniu maszynowym oraz w głębokich sieciach neuronowych. Służy do obliczeń na wielowymiarowych macierzach, które nazywane są tensorami [1].
- **Keras** [6] to biblioteka do wdrażania i szkolenia modeli sieci neuronowych. Może być wykorzystywana z modułem TensorFlow.
- **Scikit-learn** [7] to zestaw narzędzi wykorzystywany w uczeniu maszynowym.
- **Pandas** [8] to biblioteka, dzięki której można wyświetlać i analizować dane.

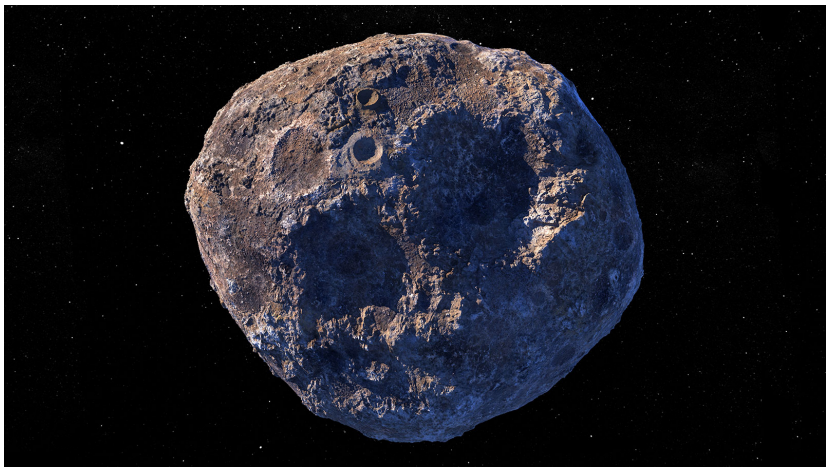
[1] TensorFlow - przez matematyków tensory zdefiniowane są jako odwzorowania wieloliniowe, *E. Karaśkiewicz, Zarys teorii wektorów i tensorów, PWN, Warszawa 1976*

- **NumPy** [9] (ang. Numeric Python) umożliwia pracę z danymi numerycznymi czy macierzami. Oferuje wiele matematycznych operacji związanych głównie z algebrą liniową.
- **Opendatasets** [10] to moduł, który pozwala na wczytywanie zbiorów danych z platformy Kaggle lub dysku Google.

Rozdział 2

Asteroidy

Asteroidy są drobnymi, ciałami skalistymi, które są pozostałością po formowaniu się Układu Słonecznego. Najbliższym skupiskiem asteroid jest pas asteroid znajdujący się między orbitami Marsa i Jowisza. Obecnie odkryto około 1 351 400 takich obiektów [11].



Rysunek 2.1: Zdjęcie przedstawiające asteroidę [12].

Wyróżniamy główne typy spektralne asteroid [13]:

- **Typu C** - składające się głównie z węgla (C), stanowią większość asteroid, posiadają niebieskawą poświatę,
- **Typu S** - składają się z głównie z materiału krzemianowego (związków krzemu) oraz pierwiastków metalicznych takich jak nikiel (Ni) i żelazo (Fe), stanowią około 17% wszystkich asteroid, posiadają czerwonawe, jasne widmo,
- **Typu M** - asteroidy składające się z pierwiastków metalicznych takich jak nikiel oraz żelazo, stanowią 8% wszystkich obiektów.

2.1 Parametry opisujące orbitę asteroidy

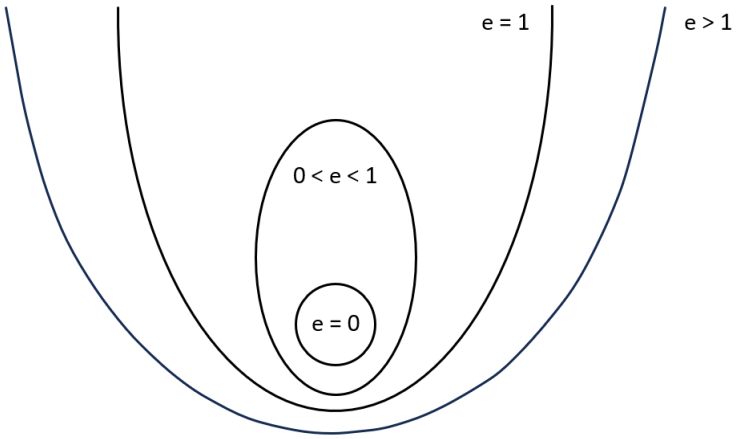
Asteroidy nieustannie przemierzają przestrzeń kosmiczną. Parametry dostarczają nam wielu przydatnych informacji na ich temat.

W danych, które wykorzystałem do analizy mamy następujące parametry [14]:

- **Nazwa obiektu** (ang. Object Name) to nazwa danej asteroidy,
- **Epoka** (ang. Epoch) jest momentem w czasie, dla którego określa się położenie astronomiczne ciała lub parametry jego orbity,
- **Oś orbity** (Orbit Axis) to linia po której krąży ciało w przestrzeni kosmicznej,
- **Ekscentryczność orbity** (ang. Orbit Eccentricity) określa kształt orbity. Dla orbity gdzie ekscentryczność jest większa od 1 mamy kształt hiperboli. Ekscentryczność równa 1 określa parabolę, pomiędzy 0 a 1 mamy kształt elipsy, natomiast jeśli jest równa 0 to wtedy mamy orbitę kołową. Wzór na ekscentryczność wynosi:

$$e = \sqrt{1 + \frac{2EL^2}{m\alpha^2}}, \tag{2.1}$$

  gdzie, E jest całkowitą energią orbitalną, L jest momentem pędu, m jest zredukowaną masą, natomiast α jest współczynnikiem prawa odwrotności kwadratu.



Rysunek 2.2: Rysunek przedstawiający ekscentryczność orbity. Ekscentryczność równa 0 (okrąg), $0 < e < 1$ (elipsa), $e = 1$ (parabola), $e > 1$ (hiperbola) [opracowanie własne].

- **Nachylenie orbity** (Inklinacja orbity, ang. Orbit Inclination) jest kątem znajdującym się pomiędzy płaszczyzną orbity a płaszczyzną odniesienia,
- **Argument peryhelium** (ang. Perihelion Argument) to również kąt ale występujący w płaszczyźnie orbity pomiędzy węzłem wstępującym (punkt na orbicie, gdzie obiekt wznosi się przez płaszczyznę ekliptyki, przechodząc z dołu do góry) a punktem peryhelium,
- **Długość węzła orbity** (dokładniej Długość węzła wstępującego, ang. Node Longitude) to kąt pomiędzy określonym kierunkiem odniesienia, a kierunkiem węzła wstępującego, mierzonego w określonej płaszczyźnie odniesienia,

- **Średnia anomalia** (ang. Mean Anomaly) - jest to kątowa odległość od perycentrum, jaką miałoby fikcyjne ciało, gdyby poruszało się po orbicie kołowej z równomierną prędkością, w takim samym czasie jak rzeczywiste ciało na swojej eliptycznej orbicie. Wzór na średnią anomalię jest wyrażony za pomocą ekscentryczności anomalii E oraz ekscentryczności e z równania Keppler'a:

$$M = E - esinE, \tag{2.2}$$

  lub inaczej:

$$M = M_0 + n(t - t_0), \tag{2.3}$$

  gdzie M_0 jest średnią anomalią w epoce, a t_0 jest epoką,

- **Dystans peryhelium** (ang. Perihelion Distance) to najmniejsza odległość między Słońcem a punktem orbity planety, który znajduje się najbliżej Słońca,

- **Dystans aphelium** (ang. Aphelion Distance) to największa odległość między Słońcem a punktem orbity planety, który znajduje się najdalej od Słońca,

- **Okres orbitalny** (ang. Orbital Period) jest czasem jaki jest potrzebny ciału na pokonanie całej długości orbity,

- **Minimalne przecięcie orbity** (ang. Minimum Orbit Intersection) to wielkość używana do oceny potencjalnych zbliżeń lub zderzeń między ciałami w kosmosie,

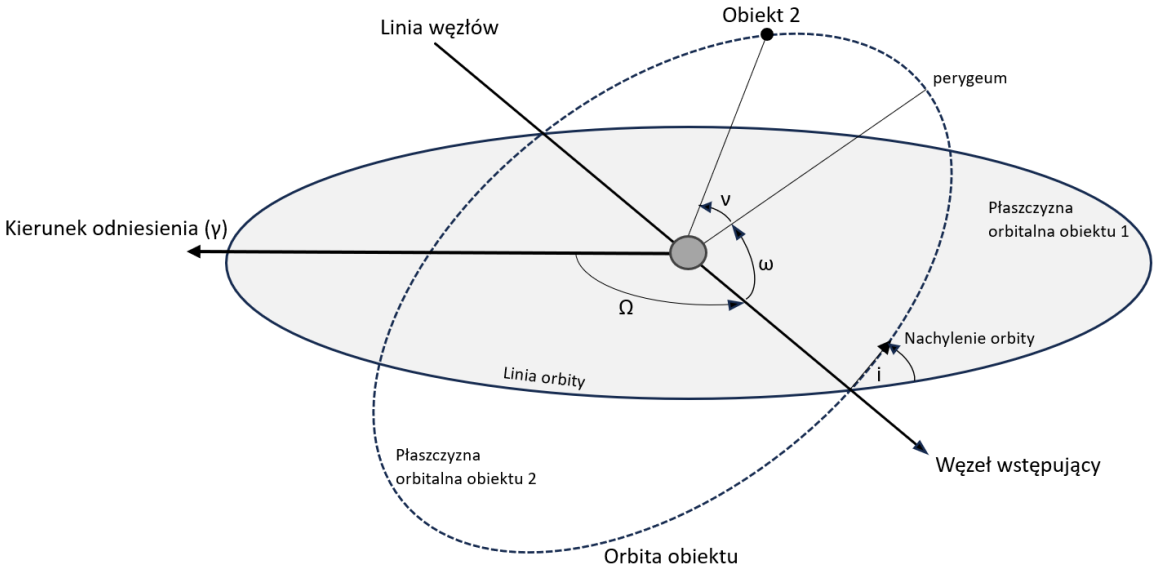
- **Referencja Orbity** (znana jako ramą odniesienia ang. Orbital Reference) to układ współrzędnych używany do opisu ruchu ciał w przestrzeni kosmicznej,

- **Wartość asteroidy** (absolutna wielkość asteroidy, ang. Asteroid Magnitude) to jasność, którą obserwator może ujrzeć jeśli asteroida została umieszczona w odległości 1 jednostki astronomicznej od Słońca przy zerowym kącie fazowym,

- **Klasyfikacja** (ang. Classicifation) jest określeniem jakiego typu jest dana asteroida. W danych mamy inną niż wcześniej opisywaną klasyfikację:

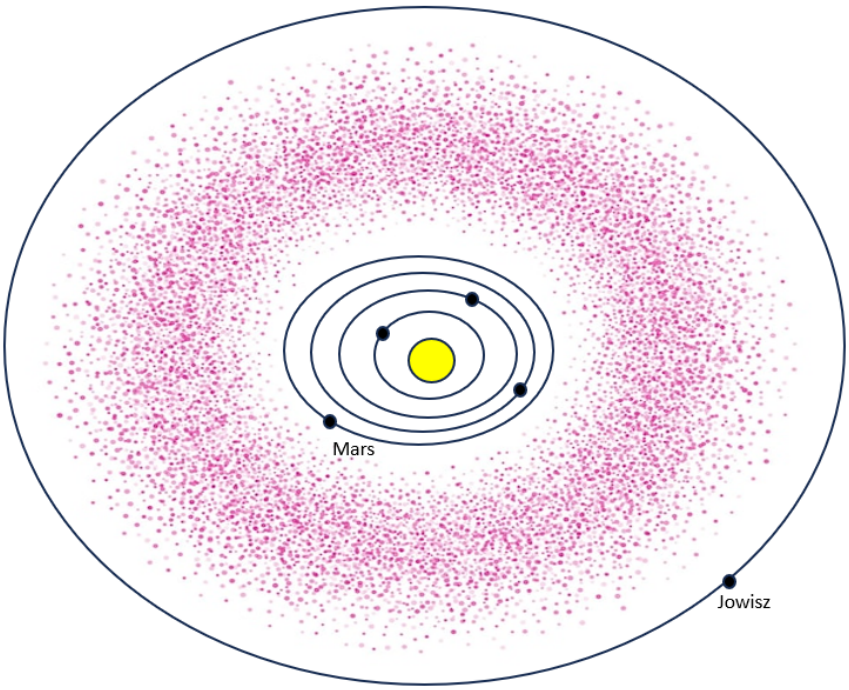
  - **typu Amora** (Amor Asteroid) - przecinają orbitę Marsa, ale nie Ziemi. Ich peryhelium znajduje się w zakresie od 1.017 do 1.3 AU,
  - **typu Apolla** (Apollo Asteroid) - przecinają orbitę Ziemi, a kilka z nich zbliża się nawet bliżej Słońca niż Merkury. Ich odległość od Słońca wynosi ¿= 1 AU, ale peryhelium jest mniejsze niż 1.017 AU,
  - **typu Atena** (Aten Asteroid) - orbity tych asteroid mają mniej niż 1 AU wielkości wielkiej półosi, a peryhelium jest większe niż 0.983 AU (gdzie 0.983 to minimalna odległość Ziemi od Słońca, a 1.017 to maksymalna) [19],
  - **typu Atiry** (ang. Apohele Asteroid) - asteroidy znajdujące się wewnątrz orbity ziemskiej,

- **Zagrażający** (ang. Hazardous) - określa czy dana asteroida zagraża Ziemi czy nie.



Rysunek 2.3: Elementy orbitalne: Ω oznacza długość węzła wstępującego, ω jest argumentem perycentrum, a ν jest prawdziwą anomalią. Opracowanie własne na podstawie [20].

2.2 Pas asteroid



Rysunek 2.4: Schemat poglądowy przedstawiający rozmieszczenie pasa asteroid (oznaczony na kolor różowy) - opracowanie własne.

Większość asteroid znajduje się w pasie znajdującym się między orbitami Marsa i Jowisza. Kształtem przypomina dysk otaczający Układ Słoneczny. Są to obiekty bliskie Ziemi (ang. Near Earth Objects) czyli komety i asteroidy, które w wyniku grawitacji pobliskich planet

znalazły się na orbitach umożliwiających im wejście w sąsiedztwo Ziemi.

Naukowe zainteresowanie asteroidami wynika w dużej mierze z ich statusu jako stosunkowo niezmienionych pozostałości po procesie formowania się Układu Słonecznego około 4,6 miliarda lat temu. Podobnie, dzisiejsze asteroidy są kawałkami pozostałymi po początkowej aglomeracji planet wewnętrznych, w tym Merkurego, Wenus, Ziemi i Marsa.

Rozdział 3

System Sentry

Sentry jest technologią opracowaną przez NASA JPL (ang. Jet Propulsion Laboratory). Zadaniem systemu jest wykrywanie asteroid potencjalnie zagrażających Ziemi z przewidywaniem na najbliższe 100 lat. Został wprowadzony w 2002 roku i działał nieustannie do roku 2021 kiedy to wprowadzono jego następną generację - Sentry II [21].

Asteroidy znajdujące się blisko Ziemi są wczytywane do systemu a następnie dostarczane są bieżące informacje na temat ich trajektorii, co umożliwia dalsze monitorowanie ruchu w kosmosie.

Nowsza generacja jest o wiele skuteczniejsza w wykrywaniu zagrożenia niż poprzednik, jego wyliczenia są bardziej precyzyjne. Wykorzystuje technikę Pseudo-Obserwację Zderzeń (ang. Impact Pseudo-Observation, w skrócie IOBS), gdy poprzedni system używał Linii Zmienności (ang. Line of Variations, w skrócie LOV). W nadzwyczajnych przypadkach używana jest Symulacja Monte Carlo [21].

3.1 Pseudo-Obserwacja Zderzeń (IOBS)

Jest to metoda wykorzystująca próbkowanie niepewności orbity asteroidy, generując przy tym duże ilości "wirtualnych" asteroid i analizując ich trajektorie. W pierwszym kroku Sentry przeprowadza analizę Monte Carlo z ograniczoną liczbą asteroid w celu wykrycia potencjalnych zagrożeń w ciągu najbliższych stu lat. Następnie rozpoczyna bardziej szczegółową analizę, gdzie za pomocą zaawansowanych filtrów określania orbity, próbuje dopasować orbitę kolizyjną do danych obserwacyjnych. Prawdopodobieństwo uderzenia jest szacowane na podstawie tej analizy, uwzględniając rozkład niepewności orbity. W tej metodzie istotną rolę gra parametr Sigma VI, który określa ilościowo, jak orbita kolizyjna pasuje do obserwacji [21].

3.2 Linia Zmienności (LOV)

Linia Zmienności (ang. Line of Variations) wykorzystuje próbkowanie "wirtualnych" asteroid wzdłuż osi centralnej obszaru niepewności orbity. Sztucznie stworzone asteroidy są numerycznie propagowane w przód w czasie, aby określić bliskie zbliżenia w stosunku do Ziemi. Jeśli algorytm znajdzie powtórzone zbliżenia, uruchamiana jest procedura znalezienia najbliższej asteroidy, która przelatuje obok Ziemi. Następnie ruch tej planetoidy jest analizowany przy pomocy technik liniowych, aby oszacować prawdopodobieństwo zderzenia [21].

Rozdział 4

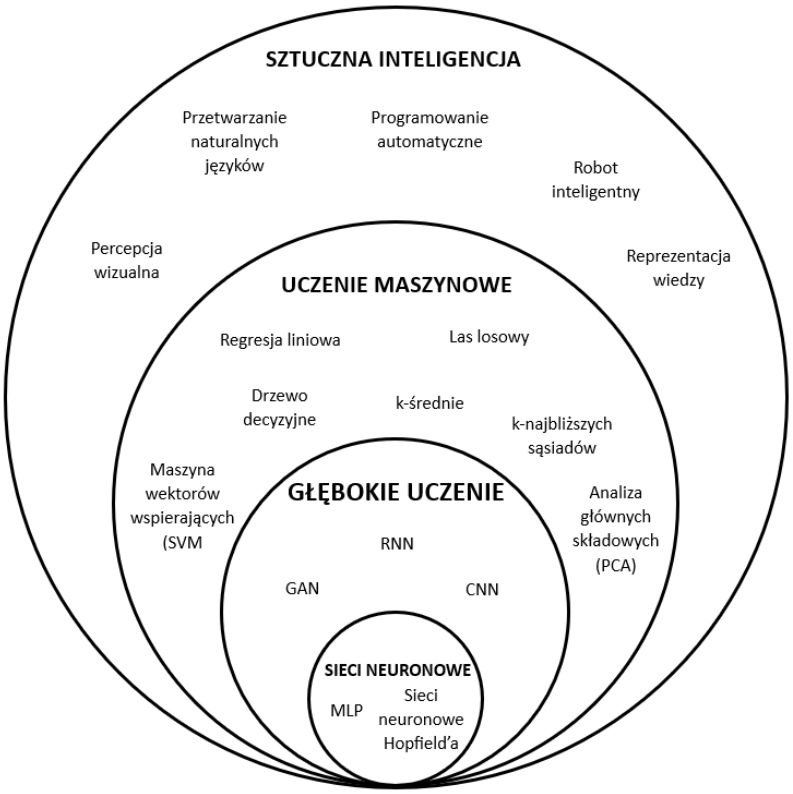
Systemy Sztucznej Inteligencji

Sztuczna Inteligencja jest terminem w dziedzinie informatyki określający rodzaj algorytmów obsługiwanym przez maszyny, który modelem i działaniem ma być zbliżony do działania ludzkiego mózgu.

Wszystko zaczęło się w roku 1956, gdzie Allen Newell, Cliff Shaw i Herbert Simon wprowadzili program Logic Theorist [22]. Jego zadaniem było naśladowanie umiejętności rozwiązywania problemów przez człowieka. Całe to dokonanie zostało zaprezentowane podczas konferencji Dartmouth Summer Reserach Project on Artificial Intelligence gdzie po raz pierwszy poruszony został problem sztucznej inteligencji. Był to dopiero początek rozwoju tej technologii.

Aktualnie Sztuczna Inteligencja posiada wiele zaawansowanych algorytmów, które ułatwiają pracę w wielu dziedzinach wiedzy. Wyróżniamy następujące gałęzie systemów Sztucznej Inteligencji:

- **Uczenie Maszynowe** - jest podzbiorem sztucznej inteligencji, w którym komputer uczy się rozpoznawać wzorce i podejmować decyzje na podstawie danych. Wyróżniamy uczenie maszynowe nienadzorowane, nadzorowane i częściowo nadzorowane.
- **Głębokie Uczenie** - to gałąź Sztucznej Inteligencji gdzie wykorzystuje się wielowarstwowe sieci neuronowe, zwane głębokimi sieciami neuronowymi. Sieć wykorzystuje trzy lub więcej warstw. Modele tego rodzaju służą do identyfikowania i klasyfikacji obiektów, rozpoznawania wzorców i relacji, przewidywania i podejmowania decyzji.

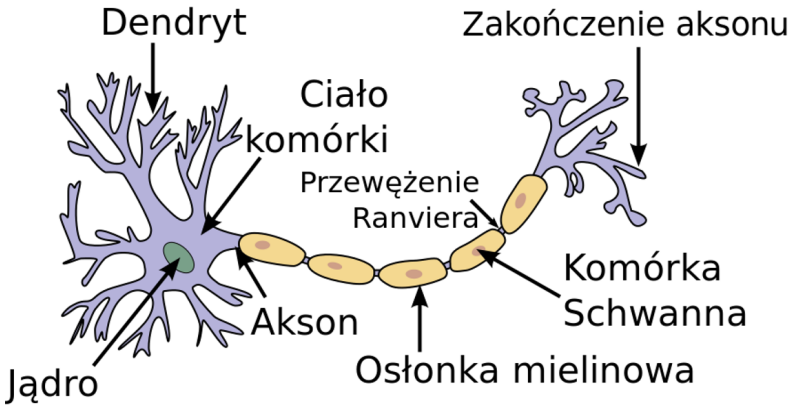


Rysunek 4.1: Podział systemów Sztucznej Inteligencji - opracowanie własne na podstawie [23].

Rozdział 5

Sieci Neuronowe

Próbując zrozumieć, jak działa biologiczny mózg, aby zaprojektować sztuczną inteligencję, Warren McCullock i Walter Pitts opublikowali w 1943 roku pierwszą koncepcję uproszczonej komórki mózgu, czyli neuronu McCullock-Pitt'a (MCP). Neurony to połączone ze sobą komórki nerwowe w mózgu, które biorą udział w przetwarzaniu i przesyłaniu sygnałów chemicznych i elektrycznych, co przedstawia poniższy rysunek:

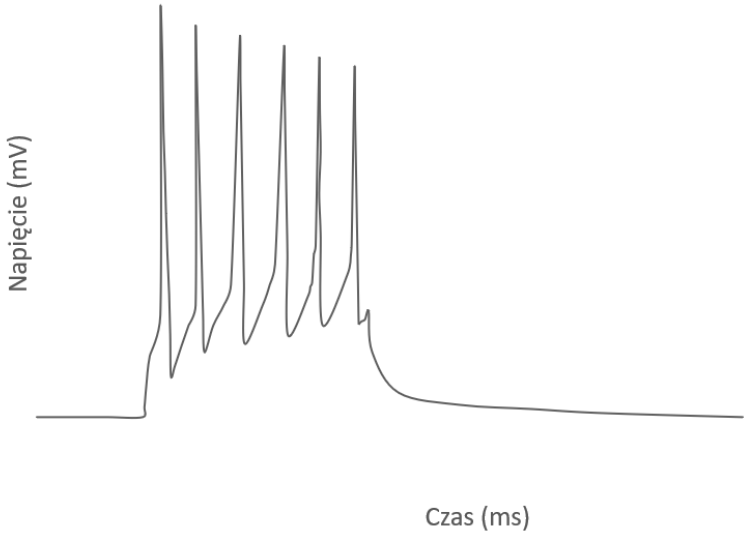


Rysunek 5.1: Budowa neuronu biologicznego u człowieka [24].

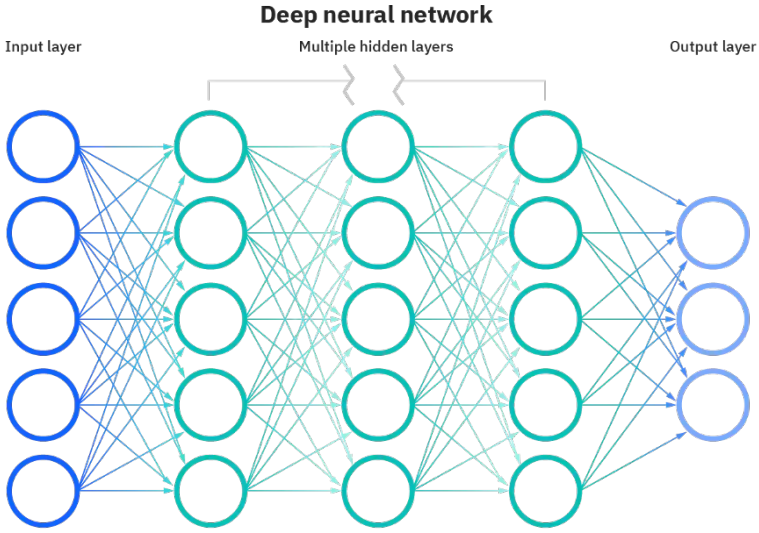
Naukowcy opisali model neuronu jako jednostkę, która pobiera dane i w wyniku daje wartość numeryczną. Wiele sygnałów dociera do dendrytów, gdzie jest następnie integrowany w ciele komórki. Jeśli skoncentrowany sygnał przekroczy pewien próg, generuje się wtedy sygnał wyjściowy, który jest później przekazany przez akson.

Dodatkowo, ten proces generowania sygnału wyjściowego jest często nazywany "spike'em neuronu". Spike neuronu to krótkotrwały impuls elektryczny, który jest generowany, gdy suma sygnałów wejściowych przekroczy próg aktywacji neuronu (rysunek 5.2). Ten impuls, znany również jako potencjał czynnościowy, jest następnie przekazywany do innych neuronów przez akson.

Kilka lat później, Frank Rosenblatt opublikował pierwszy koncept perceptronu na podstawie modelu neuronu MCP (Neuron McCullocha-Pittsa) [1]. Algorytm opierał się na automatycznym uczeniu się optymalnych współczynników wagowych, które są następnie mnożone z cechami wejściowymi w celu podjęcia decyzji, czy neuron uruchomi się czy nie. W odniesieniu do uczenia nadzorowanego, taki algorytm może być wykorzystany do przewidywania wyniku.



Rysunek 5.2: Wykres przedstawiający przebieg czasowy neuron spike'a - opracowanie własne.



Rysunek 5.3: Model głębokiej sieci neuronowej.

5.1 Matematyczne przedstawienie modelu sieci neuronowej

Bardziej formalnie, możemy umieścić ideę sztucznych neuronów w kontekście zadania klasyfikacji binarnej, gdzie dla uproszczenia odnosimy się do naszych dwóch klas jako 1 (klasa pozytywna) i -1 (klasa negatywna). Następnie możemy zdefiniować funkcję decyzyjną ($\phi(z)$), która przyjmuje liniową kombinację pewnych wartości wejściowych x i odpowiedniego wektora wag wektor w, gdzie z jest tak zwanym wejściem siatki $z = w_1x_1 + ... + w_mx_m$ (iloczyn skalarny wektora X i wektora wag W):

$$w = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}, x = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}. \tag{5.1}$$

Teraz, jeśli wejście siatki konkretnej próbki x^i jest większe niż zdefiniowany próg θ, inaczej przewidujemy klasę 1 oraz klasę -1. W algorytmie perceptronu, funkcja decyzyjna ϕ jest wariantem funkcji skoku jednostkowego:

$$\phi(z) = \begin{cases} 1 \text{ jeśli } z \geqslant \theta \\ -1 \text{ w przeciwnym wypadku .} \end{cases} \tag{5.2}$$

Dla uproszczenia możemy przenieść próg theta na lewą stronę równania i zdefiniować wagę zero jako $w_0 = -\theta ix + 0 = 1$ więc piszemy z w bardziej zgrabnej formie:

$$z = w_0 x_0 + w_1 x_1 + ... + w_m x_m = w^T x \tag{5.3}$$

oraz:

$$\phi(z) = \begin{cases} 1 \text{ jeśli } z \geqslant \theta \\ -1 \text{ w przeciwnym wypadku .} \end{cases} \tag{5.4}$$

W literaturze dotyczącej uczenia maszynowego [1] ujemny próg lub waga $w_0 = -\theta$ jest zwykle nazywana jednostką odchylenia "bias".

5.2 Różnica pomiędzy uczeniem maszynowym a siecią neuronową

Podstawową różnicą między statystycznym uczeniem maszynowym a uczeniem głębokim jest sposób, w jaki każdy algorytm się uczy i ile danych wykorzystuje każdy typ algorytmu.

Uczenie głębokie automatyzuje znaczną część procesu ekstrakcji cech, co zdecydowanie zmniejsza potrzebę ręcznej interwencji człowieka. Umożliwia również efektywne wykorzystanie dużych zbiorów danych, co czyni je skalowalnym podejściem do uczenia maszynowego. Ta zdolność jest ekscytująca, zwłaszcza biorąc pod uwagę rosnące zastosowanie niestrukturalnych danych, które, jak szacuje się, stanowią ponad 80% danych organizacyjnych.

Jednakże, to nie oznacza, że w sieciach neuronowych nie musimy odpowiednio przygotować danych wejściowych. Również w uczeniu głębokim, odpowiednie warstwy sieci neuronowej mogą automatycznie przeprowadzić "preprocessing", który prowadzi do ekstrakcji cech - na przykład, warstwa konwolucyjna może łączyć sąsiednie piksele w celu ekstrakcji cech.

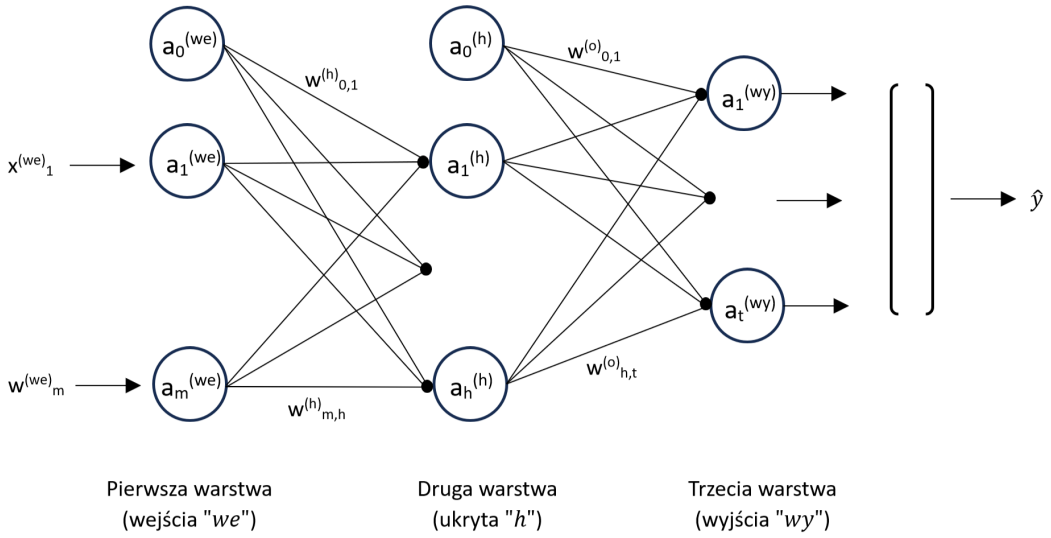
W uczeniu statystycznym, podobny proces ekstrakcji cech można zrealizować poprzez połączenie w potok (pipe) algorytmów ekstrakcji cech z klasyfikatorem. W sieciach neuronowych jest to robione bardziej automatycznie, ale jeżeli źle dobierzemy warstwy, to możemy dostać gorsze wyniki niż w uczeniu statystycznym.

Obserwowanie wzorców w danych pozwala modelowi głębokiego uczenia na odpowiednie grupowanie danych wejściowych. Model głębokiego uczenia wymaga większej liczby punktów danych, aby poprawić dokładność, podczas gdy model uczenia maszynowego opiera się na mniejszej liczbie danych, biorąc pod uwagę jego podstawową strukturę danych.

5.3 Rodzaje sieci neuronowych

5.3.1 Perceptron wielowarstwowy

Perceptron wielowarstwowy (ang. Multilayer Perceptron, w skrócie MLP) jest przykładem jednokierunkowej sieci neuronowej. Poniższy rysunek 5.4 ukazuje ten model, gdzie mamy jedną warstwę wejściową, jedną ukrytą oraz jedną warstwę wyjściową. Wartości w warstwie ukrytej są połączone z warstwą wejściową, a warstwa wyjściowa jest połączona z warstwą ukrytą. Jeśli taka sieć ma więcej niż jedną warstwę ukrytą, nazywamy ją głęboką siecią neuronową.



Rysunek 5.4: Przedstawienie graficzne modelu perceptronu wielowarstwowego (MLP) - opracowanie własne na podstawie [1].

Oznaczamy i-tą jednostkę aktywacji w l-tej warstwie jako $a_i^{(l)}$. Nie używamy indeksów numerycznych do odwoływania się do warstw tylko stosujemy indeks górny we dla warstwy wejściowej, h dla warstwy ukrytej i wy dla wyjściowej. Dla przykładu $a_i^{(we)}$ nawiązuje do i-tej wartości w warstwie wejściowej, $a_i^{(h)}$ do i-tej wartości w warstwie ukrytej, a $a_i^{(wy)}$ do i-tej wartości w warstwie wyjściowej. Taki zapis jest wygodniejszy. W modelu, jednostki aktywacji $a_0^{(we)}$ oraz $a_0^{(h)}$ są biasem (dodatkowym wejściem (neuronem), na którym występuje stała wartość), które ustawiamy na 1. Aktywacja jednostek w warstwie wejściowej to tylko jej wejście, oraz bias, czyli:

$$a^{(we)} = \begin{bmatrix} a_0^{(wy)} \\ a_1^{(wy)} \\ \vdots \\ a_m^{(wy)} \end{bmatrix} = \begin{bmatrix} 1 \\ x_1^{(wy)} \\ \vdots \\ x_m^{(wy)} \end{bmatrix}. \tag{5.5}$$

Każda wartość w warstwie l jest połączona ze wszystkimi wartościami w warstwie $l+1$ poprzez współczynnik wagowy. Przykładem może być połączenie między k-tą wartością w warstwie l z j-tą wartością w warstwie $l+1$ i będzie to zapisane jako $w_{k,j}^{(l)}$. Macierz wag, która łączy wejście z warstwą ukrytą, będzie oznaczona jako $W^{(h)}$, a $W^{(wy)}$ to macierz łącząca warstwę ukrytą z wyjściem.

5.3.2 Konwolucyjne sieci neuronowe

Konwolucyjne sieci neuronowe (ang. Convolutional Neural Networks, w skrócie CNN) to model używany do klasyfikacji obrazów i nie tylko. Inspiracją do stworzenia algorytmu było to, w jaki sposób kora wzrokowa ludzkiego mózgu działała podczas rozpoznawania obrazów.

Konwolucja łączy sąsiednie piksele obrazu (2D) lub sąsiednie elementy szeregu czasowego (1D) żeby wydobyć cechy angażujące sąsiednie piksele. Dlatego jeżeli interesują nas cechy angażujące większa liczbę elementów to musimy zwiększyć rozmiar jądra konwolucyjnego.

Sieć neuronowa jest w stanie automatycznie uczyć się cech z surowych danych, które są najbardziej przydatne dla konkretnego zadania. Wczesne warstwy (czyli te po wyjściu z warstwy wejściowej) wyodrębniają cechy niskiego poziomu. Wielowarstwowe sieci neuronowe, a w szczególności głęboki konwolucyjne sieci neuronowe, tworzą hierarchię cech, gdzie łączą cechy niskiego poziomu w sposób warstwowy, tworząc cechy cechy wysokiego poziomu. W ten sposób cechy niskiego poziomu, takie jak krawędzie i plamy, są wyodrębniane z wcześniejszych warstw, które po złączeniu, tworzą cechy wysokiego poziomu czyli gotowe kształty obiektów.

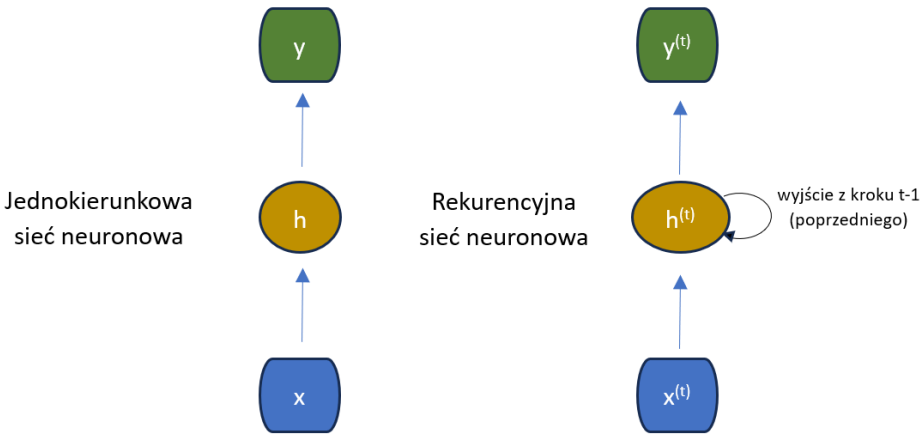
CNN oblicza mapę cech (trójwymiarowy tensor (wysokość, szerokość, głębia) z obrazu wejściowego, gdzie każdy element pochodzi z lokalnego fragmentu na obrazie wejściowym. Ten lokalny obszar pikseli jest określany jako lokalne pole recepcyjne. Sieć zazwyczaj dobrze sobie radzi z zadaniami związanymi z obrazem - wynika to z dwóch rzeczy:

- Rzadkie łączenie (ang. sparse-connectivity) - Pojedynczy element w mapie cech jest połączony z niewielkim fragmentem pikseli,
- Udostępnianie parametrów (ang. parameter-sharing) - Te same wagi są używane dla różnych fragmentów obrazu wejściowego.

Zazwyczaj sieć składa się z kilku warstw konwolucyjnych i warstw podpróbkowania, po których następuje jedna lub więcej warstw w pełni połączonych na końcu. W pełni połączone warstwy są zasadniczo wielowarstwowym perceptronem, w którym każda wartość wejściowa i jest połączona z każdą wartością wyjściową j z wagą w_{ij}.

5.3.3 Rekurencyjne sieci neuronowe

Rekurencyjne sieci neuronowe (ang. Recurrent Neural Networks, w skrócie RNN) przetwarza dane sekwencyjnie, a rekurencja zapewnia, że wyjście z poprzedniego kroku jest wejściem do kolejnego wraz z nową porcją danych.



Rysunek 5.5: Porównanie jednokierunkowej sieci neuronowej z rekurencyjną siecią neuronową. Opracowanie własne na podstawie [1].

Na rysunku 5.5 mamy przedstawione porównanie obu sieci neuronowych. Obie posiadają jedną warstwę ukrytą oznaczoną literą h, warstwą wejścia jest x, natomiast y to warstwa wyjścia.

W standardowym modelu sieci jednokierunkowej, informacja przepływa z wejścia do warstwy ukrytej i następnie do wyjścia. Z kolei w sieci rekurencyjnej warstwa ukryta ma informacje z warstwy wejścia jak i warstwy ukrytej.

Każda jednostka ukryta w standardowej sieci neuronowej otrzymuje tylko jedno wejście - aktywację sieci związaną z warstwą wejściową. Natomiast w sieci RNN, każda jednostka ukryta otrzymuje dwa różne zestawy danych wejściowych - wstępną aktywację z warstwy wejściowej i aktywację tej samej warstwy ukrytej z poprzedniego kroku czasowego $t-1$.

W kroku czasowym $t = 0$, jednostki ukryte są inicjowane jako zera lub małe wartości losowe. Następnie, dla każdego kroku czasowego. Następnie, w kroku czasowym, w którym $t > 0$, jednostki ukryte otrzymują swoje wejścia z próbki w kroku bieżącym $x^{(t)}$ i wcześniejszych wartościach ukrytych przy $t-1$ zapisanych jako $h^{(t-1)}$.

Te powiązania w RNN, są skojarzone z macierzą wag. Wagi te zmieniają się w wykonywaniu kroków rekurencyjnych w procesie uczenia. Wyróżniamy następujące macierze wag:

- W_{xh} - macierz wagi pomiędzy warstwą wejścia $x^{(t)}$ a warstwą ukrytą h,
- W_{hh} - macierz wagi powiązana z krawędzią rekurencyjną,
- W_{hy} - macierz wagi pomiędzy warstwą ukrytą a warstwą wyjścia.

W niektórych przypadkach (łączenie macierzy przyspiesza obliczenia, ułatwia implementację sieci i zwiększa jej czytelność), możemy obserwować, że macierze wag W_{xh} i W_{hh} są połączone w macierz $W_h = [W_{xh}; W_{hh}]$.

Obliczanie aktywacji jest podobne do modelu perceptronów wielowarstwowych i innych rodzajów jednokierunkowych sieci neuronowych. W przypadku warstwy ukrytej, wejście z sieci z_h (wstępna aktywacja) jest obliczana za pomocą kombinacji liniowe. Oznacza to, że obliczamy sumę iloczynów macierz wag z odpowiednimi wektorami i dodajemy jednostkę bias - $z_h^{(t)} = W_{xh}x^{(t)} + W_{hh}h^{(t-1)} + b_h$.

Następnie aktywację ukrytych parametrów w kroku czasowym t obliczamy w następujący sposób:

$$h^{(t)} = \phi_h(z_h^{(t)} = \phi_h(W_{xh}x^{(t)} + W_{hh}h^{(t-1)} + b_h) \tag{5.6}$$

Tutaj b_h jest wektorem bias dla warstwy ukrytej, a ϕ_h jest funkcją aktywacji warstwy ukrytej. Wzór na obliczanie warstw ukrytych wynosi:

$$h^{(t)} = \phi_h\left([W_{xh}; W_{hh}]\begin{bmatrix} x^{(t)} \\ h^{(t-1)} \end{bmatrix} + b_h\right) \tag{5.7}$$

Po obliczeniu aktywacji parametrów ukrytych w bieżącym kroku czasowym, aktywacje parametrów wyjściowych zostaną obliczone w następujący sposób:

$$y^{(t)} = \phi_y\left(W_{hy}h^{(t)} + b_y\right) \tag{5.8}$$

Ważnym aspektem trenowania sieci rekurencyjnych jest zastosowanie algorytmu wstecznej propagacji przez czas (BPTT, ang. Backpropagation Through Time), który jest rozszerzeniem standardowej wstecznej propagacji stosowanej w sieciach neuronowych. W BPTT, gradienty błędów są obliczane poprzez rozwinięcie sieci w czasie, co umożliwia propagowanie błędów wstecznie przez wiele kroków czasowych.

Podczas aktualizacji wag, problemy z wybuchającymi i zanikającymi gradientami są powszechne. W przypadku zanikania gradientów, wartości gradientów stają się bardzo małe, co sprawia, że wagi zmieniają się bardzo wolno. Z kolei wybuchające gradienty prowadzą do bardzo dużych wartości gradientów, co może powodować niestabilność procesu uczenia. Rozwiązaniem tego problemu jest użycie sieci LSTM (ang. Long Short-Term Memory). Struktura tej sieci składa się z komórek pamięci (ang. memory cells), które mogą przechowywać informacje przez długie okresy czasu, dzięki mechanizmom bramkowania.

5.4 Optymalizacja wyników

5.4.1 Przeszukiwanie siatki

Przeszukiwanie siatki (ang. Grid Search) jest metodą optymalizacji algorytmów sztucznej inteligencji. Służy na odpowiednim doborze parametrów (tzw. hiperparametrów) analizy tak, aby uzyskać większą precyzję modelu. Hiperparametry to specjalne rodzaje parametrów, które są ustalane przed rozpoczęciem procesu uczenia się modelu. Są one kluczowe dla regulacji i kierowania procesem uczenia się. W przeciwieństwie do zwykłych parametrów, które są dostosowywane i optymalizowane podczas uczenia się, hiperparametry są ustawiane ręcznie na podstawie doświadczenia lub za pomocą technik optymalizacji.

Przeszukiwanie siatki działa poprzez testowanie wszystkich możliwych kombinacji hiperparametrów zdefiniowanych przez użytkownika. Dla każdej unikalnej kombinacji, model jest trenowany i jego wydajność jest oceniana za pomocą określonej metryki. Na koniec wyświetlany jest najlepszy wynik z wybranymi hiperparametrami.

5.5 Preprocesowanie danych - standaryzacja

Standaryzacja pozwala na ujednolicenie wartości wszystkich cech w danych, co poprawia wydajność algorytmów sztucznej inteligencji. Polega na przekształceniu danych wejściowych tak, aby ich rozkład miał średnią wartość równą 0 oraz odchylenie standardowe równe 1:

$$x_{standaryzacja} = \frac{x - \mu}{\sigma}, \tag{5.9}$$

gdzie, x jest pojedynczą wartością danej cechy, μ jest średnią wartością cechy, a σ to odchylenie standardowe cechy.

5.6 Opis parametrów sieci neuronowych

5.6.1 Algorytmy optymalizacji

Algorytmy optymalizacji są kluczowym elementem uczenia się maszynowego i sieci neuronowych. Ich głównym celem jest minimalizacja (lub maksymalizacja) funkcji kosztu, która jest miarą błędu między przewidywaniami modelu a rzeczywistymi danymi. Poniżej opisano algorytmy, które zostały użyte w analizie [25]:

Adam (ang. Adaptive Moment Estimation) - algorytm ten służy do optymalizacji działania sieci neuronowej. Opiera się na adaptacyjnej estymacji momentów pierwszego i drugiego rzędu.

Stochastyczny spadek gradientowy (ang. Stochastic Gradient Descent, w skrócie SGD). Stochastyczny, czyli odnoszący się do losowego wyboru próbek danych do aktualizacji parametrów modelu w każdej iteracji algorytmu. W każdym kroku, algorytm wybiera losowo

jedną próbkę z zestawu danych treningowych, oblicza gradient funkcji straty dla tej próbki, a następnie aktualizuje wagi sieci.

Adagrad (z ang. Adaptive Gradient Algorithm) dostosowuje prędkość uczenia się dla każdego parametru, zmniejszając prędkość uczenia się dla często aktualizowanych parametrów i zwiększając dla rzadko aktualizowanych.

Adadelta rozszerza algorytm Adagrad, który próbuje redukować jego agresywne, monotoniczne zmniejszanie prędkości uczenia się. Zamiast akumulować wszystkie poprzednie gradienty kwadratowe, Adadelta ogranicza akumulację do określonej liczby okresów.

Adamax to wariant algorytmu Adam. Zamiast obliczać drugi moment gradientu (niecentrowany) jako średnią geometryczną kwadratów, Adamax oblicza go jako maksimum. Dzięki temu jest bardziej stabilny niż Adam w przypadku rzadkich gradientów.

Nadam (Nesterov-accelerated Adaptive Moment Estimation) jest to wariant algorytmu Adam z dodanym Momentum Nesterova. Momentum Nesterova najpierw wykonuje duży krok w kierunku poprzedniego gradientu, oblicza gradient w tym nowym miejscu, a następnie wykonuje korektę.

RMSprop (Root Mean Square Propagation) to metoda, która próbuje rozwiązać problem Adagrad z szybkim zmniejszaniem stopy uczenia się. W RMSprop, zamiast akumulować wszystkie poprzednie gradienty kwadratowe, wprowadza termin zapominania, który używa średniej ruchomej poprzednich gradientów kwadratowych. Wzór aktualizacji dla RMSprop jest następujący:

$$w(t+1) = w(t) - \frac{\eta}{\sqrt{v(t)}} \nabla E(t), \tag{5.10}$$

gdzie w_t to wagi w czasie t, η to współczynnik uczenia, $\nabla E(t)$ to gradient funkcji straty w czasie t, natomiast $v(t)$ jest wykładniczo ważoną średnią kroczącą gradientów:

$$v(t) = \beta v(t-1) + (1-\beta)[\nabla E(t)]^2, \tag{5.11}$$

$$w_{t+1} = w_t - \eta \nabla L(w_t; x_i, y_i), \tag{5.12}$$

gdzie w_t to wagi przed aktualizacją w kroku t, współczynnik uczenia η kontroluje wielkość kroku, jaki wykonamy w kierunku przeciwnym do gradientu. Im mniejsza wartość, tym mniejsze kroki są wykonywane, co może prowadzić do bardziej stabilnej, ale wolniejszej konwergencji. Gradient funkcji straty $\nabla L(w_t; x_i, y_i)$ jest wektorem, który wskazuje kierunek najszybszego wzrostu funkcji straty. Określa jak zmieniać wagi aby zminimalizować funkcję straty. SGD z zanikającą wykładniczo wartością stałej uczenia może dorównać algorytmom takim jak 'adam'.

5.6.2 Warstwy w sieciach neuronowych

Sieci neuronowe składają się z wielu warstw, które przetwarzają informacje na różne sposoby. Każda warstwa składa się z wielu neuronów, które przetwarzają dane i przekazują je do następnej warstwy. Wyróżniamy [27]:

- **Dense** łączy każdy neuron w warstwie ze wszystkimi neuronami w poprzedniej warstwie. Pozwala to na wyuczenie algorytmu nieliniowych zależności pomiędzy cechami.
- **SimpleRNN** przetwarza dane sekwencyjnie, co pozwala na uwzględnienie pamięci z poprzednich kroków w obliczeniach bieżącego kroku.

- **Dropout** stosowana jest po to aby zapobiec przeuczeniu modelu. Pewien procent neuronów w warstwie jest wyłączanych losowo, co zmusza sieć do nauki bardziej sprzyjających cech.
- **Conv1D** to warstwa konwolucyjna, gdzie filtr przesuwa się przez wejściowy tensor i wykonuję operację konwolucji na wejściach, generując mapę cech.
- **BatchNormalization** - normalizuje aktywacje poprzedniej warstwy na poziomie wsadu, czyli na każdym kroku uczenia się, utrzymując średnią aktywacji bliską zeru i odchylenie standardowe bliskie jeden.
- **GlobalAveragePooling1D** - warstwa oblicza średnią wartość dla każdego wymiaru wejścia dla każdego punktu danych w wsadzie.

Ważnym elementem jest właśnie **funkcja aktywacji** [32]. Przetwarza ona sumę sygnałów wejściowych neuronu, przekształcając je w sygnał wyjściowy. W tym przykładzie sieci neuronowej zostały użyte następujące funkcje aktywacji:

- **liniowa** (ang. *linear*), to nic innego jak rodzaj funkcji matematycznej, gdzie każda zmienna niezależna występuje w stopniu pierwszym, tworząc prostą liniową na wykresie:

$$f(x) = ax + b, \tag{5.13}$$

  gdzie a i b są stałymi oraz $a \neq 0$,
- **tangens hiperboliczny** (ang. hyperbolic tangent, *tanh*) mieści się w przedziale $(-1, 1)$ co sprawia, że est bardziej symetryczny niż sigmoidalna funkcja logistyczna. Ta symetryczność pomaga w równoważeniu sygnałów przekazywanych przez sieć neuronową:

$$y(x) = \frac{1}{1 + e^{-\beta x}} - 1 = \frac{1 - e^{-\beta x}}{1 + e^{-\beta x}}, \tag{5.14}$$

- **poprawiona funkcja liniowa** (ang. Rectified linear unit, $ReLU$) opiera się na szkoleniu cech dzięki rzadszym aktywacjom jednostek. Funkcja zwraca wartość zero dla wszystkich wartości ujemnych oraz zachowuje wartość wejściową dla wartości dodatnich:

$$x^+ \doteq \begin{cases} 0 & \text{jeśli } x \leqslant 0 \\ x & \text{jeśli } x > 0 \end{cases}. = max(0, x) = x_{1x>0}. \tag{5.15}$$

Część II

Część praktyczna

Rozdział 6

Wyniki analizy

6.1 Opis danych

6.1.1 Wstępne przetworzenie danych

Na wstępie, importujemy wszystkie potrzebne do analizy biblioteki oraz pobieramy zbiór danych z platformy Kaggle:

```
#zaimportowanie bibliotek
import os
import keras
import tensorflow as tf
import opendatasets as ods
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPRegressor
from tensorflow.keras.layers import Input, Dense, Dropout, LeakyReLU
from tensorflow.keras import Model
from tensorflow.keras.optimizers import Adam, RMSprop
from keras import backend
from keras.models import Sequential
from sklearn.metrics import r2_score, mean_squared_error, make_scorer
from sklearn.model_selection import GridSearchCV
from sklearn.base import BaseEstimator, RegressorMixin

#pobranie danych z platformy Kaggle
ods.download(
    "https://www.kaggle.com/nasa/asteroid-impacts?select=orbits.csv")
```

Następnie wczytujemy dane i je wyświetlamy:

```
# wczytanie danych i ich wyświetlenie
plik =('asteroid-impacts/\
orbits-orbits.csv')
dane = pd.read_csv(plik)
dane.head()
```

	Object Name	Epoch (TDB)	Orbit Axis (AU)	Orbit Eccentricity	Orbit Inclination (deg)	Perihelion Argument (deg)	Node Longitude (deg)	Mean Anomoly (deg)	Perihelion Distance (AU)	Aphelion Distance (AU)	Orbital Period (yr)	Minimum Orbit Intersection Distance (AU)	Orbital Reference	Asteroid Magnitude	Classification	Hazardous
0	433 Eros	57800	1.4579	0.2226	10.8277	178.8050	304.3265	319.3111	1.1335	1.78	1.76	0.1492	598	11.16	Amor Asteroid	False
1	719 Albert	57800	2.6385	0.5479	11.5822	156.1409	183.9204	224.5535	1.1928	4.08	4.29	0.2004	78	15.50	Amor Asteroid	False
2	887 Alinda	57800	2.4787	0.5671	9.3561	350.3482	110.5444	351.3730	1.0731	3.88	3.90	0.0925	188	13.40	Amor Asteroid	False
3	1036 Ganymed	57800	2.6628	0.5338	26.6929	132.4690	215.5551	92.5640	1.2413	4.08	4.35	0.3421	597	9.45	Amor Asteroid	False
4	1221 Amor	57800	1.9191	0.4356	11.8795	26.6572	171.3448	313.7379	1.0832	2.76	2.66	0.1068	70	17.70	Amor Asteroid	False

Rysunek 6.1: Pierwsze pięć wyników z ramki danych - opracowanie własne.

Wybieramy cechy (kolumny z danymi), które będą potrzebne do modelu, tworzymy zmienną 'cechy'. Następnie usuwamy kolumnę 'Object Name' - nie wnosi nic do analizy:

```
# definiujemy zmienna zawierającą cechy asteroid
cechy= ['Epoch (TDB)', 'Orbit Eccentricity', 'Perihelion Argument (deg)',
       'Node Longitude (deg)', 'Mean Anomoly (deg)', 'Orbit Inclination (deg)
    ',
       'Aphelion Distance (AU)','Asteroid Magnitude',
       'Orbital Period (yr)', 'Minimum Orbit Intersection Distance (AU)',
       'Orbital Reference' ]

# usuwamy nazwy asteroid, nie będą potrzebne
var_usun = ['Object Name']
dane.drop(var_usun, axis=1, inplace=True)
#kategoryzacja
klasyfikacja = ['Classification']
```

Konwertujemy cechy na wartości liczbowe, będzie prościej pracować na modelach sieci neuronowych:

```
# konwersja cech kategorycznych na wartości liczbowe za pomocą kodowania
# kategorii, następnie wyświetlenie danych
for column in klasyfikacja:
    dane[column] = dane[column].astype('category').cat.codes
dane.head()
```

Znormalizowaliśmy cechy poprzez mapowanie ich wartości na przedziały. Dla każdej cechy określamy wartość minimalną i maksymalną w danej kolumnie, po czym dzielimy je na przedziały rozmieszczone równomiernie:

```
# minimalne i maksymalne wartości w kolumnie
for column in cechy:
    min = dane[column].min()
    max = dane[column].max()
    feature_bins = pd.cut(dane[column], bins=np.linspace(min, max, 21),
                                                         labels=False)
    dane.drop([column], axis=1, inplace=True)
    dane = pd.concat([dane, feature_bins], axis=1)
```

Jeśli w zbiorze danych są jakieś niezdefiniowane wartości to je usuwamy:

```
# zamiana wartości NaN na wartości puste oraz usunięcie pustych wartości
dane.replace(to_replace=np.nan, inplace = True)
dane_po=dane.dropna()
dane_po
```

Przygotowujemy dane w celu łatwiejszej analizy:

```
# standaryzacja danych
sc = StandardScaler()
```

```
dane_po_scaled = sc.fit_transform(dane_po[dane_po.columns])

# Przypisanie przeskalowanych danych do zmiennej dane_po
dane_po[dane_po.columns] = dane_po_scaled

# Wyświetlenie danych po standaryzacji
dane_po
\end{verbatim}

\noindent Przystępujemy do podziału danych na treningowe i testowe:

\begin{lstlisting}[language=Python]
# podział danych na treningowe i testowe
X = dane_po.drop(columns=[hazard])
y = dane_po[hazard]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
                                                    random_state=42)
```

6.1.2 Jednokierunkowa sieć neuronowa typu Feed Forward

W analizie stworzono prostą, jednokierunkową sieć neuronową. Zastosowano tutaj trzy warstwy neuronów:

- warstwa wejściowa *Dense* z 24 neuronami i funkcją aktywacji *tanh*,
- warstwa ukryta *Dense* z 12 neuronami i funkcją aktywacji *relu*,
- warstwa wyjściowa *Dense* z 1 neuronem i funkcją aktywacji *linear*.

Do treningu modelu wykorzystano algorytm optymalizacji Adam. Sieć jest trenowana przez 100 epok, obliczany jest również współczynnik determinacji dla wartości przewidywanych i rzeczywistych wraz z błędem średniokwadratowym:

```
# Inicjalizacja danych
adam_optimizer = Adam(learning_rate=0.005)

# Model sekwencyjny sieci neuronowej
model = Sequential()
model.add(Dense(24, activation='tanh', input_dim=X_train.shape[1]))
model.add(Dense(12, activation='relu'))
model.add(Dense(1, activation='linear'))
model.compile(loss='mean_squared_error', optimizer=adam_optimizer)

# Pętla treningowa
for epoch in range(100):
    # Trening modelu
    model.fit(X_train, y_train, epochs=1, batch_size=256, verbose=0)

    # Przewidywanie
    y_train_pred = np.squeeze(model.predict(X_train))
    y_val_pred = np.squeeze(model.predict(X_test))

    # Obliczenie wyników
    r2_train = r2_score(y_train, y_train_pred)
    r2_val = r2_score(y_test, y_val_pred)

    # Wyświetlenie wyników
    print(f"Epoka {epoch+1}: wartość treningowa =
    {r2_train:.2%}, wartość końcowa = {r2_val:.2%}")

```

```
# Przewidywanie wartości
y_pred = np.squeeze(model.predict(X_test))

# Obliczenie i wyświetlenie wyniku
r2_val = r2_score(y_test, y_pred)
print(f"\nWynik modelu wynosi: {r2_val:.2%}")

# Obliczenie błędu średniokwadratowego między
# wartościami testowymi a przewidywanymi
mse = mean_squared_error(y_test, y_pred)
# Wyświetlenie wartości MSE
print(f'Błąd średniokwadratowy (MSE): {mse:.4f}')
```

6.1.3 Rekurencyjna sieć neuronowa

Drugim algorytmem, który wykorzystano w analizie jest rekurencyjna sieć neuronowa. Trening modelu opierał się na wykorzystaniu stochastycznego spadku gradientowego. W tym przypadku użyto następujących warstw:

- pierwsza warstwa *SimpleRNN* z 60 neuronami, funkcją aktywacji *relu* i zwracaniem sekwencji,
- warstwa *Dropout* z wartością 0.4, która pomaga zapobiegać przeuczeniu sieci neuronowej,
- druga warstwa *SimpleRNN* z 40 neuronami, funkcją aktywacji *relu* i zwracaniem sekwencji.
- kolejna warstwa *Dropout* z wartością 0.4.
- trzecia warstwa *SimpleRNN* z 20 neuronami i funkcją aktywacji *relu*.
- warstwa *Dropout* z wartością 0.3.
- warstwa wyjściowa *Dense* z jednym neuronem i funkcją aktywacji *linear*.

```
# Funkcja budująca model RNN
def build_model_using_rnn(input_shape):
    model = keras.Sequential([
        keras.layers.SimpleRNN(60, input_shape=input_shape,
        activation="relu", return_sequences=True),
        keras.layers.Dropout(0.4),
        keras.layers.SimpleRNN(40, activation="relu", return_sequences=True),
        keras.layers.Dropout(0.4),
        keras.layers.SimpleRNN(20, activation="relu"),
        keras.layers.Dropout(0.3),
        keras.layers.Dense(1, activation='linear')
    ])
    return model

input_shape = (X_train.shape[1], 1)

# Budowa modelu
RNN_model = build_model_using_rnn(input_shape)

# Kompilacja modelu
RNN_model.compile(loss="mean_squared_error", optimizer='SGD', metrics=["mse"
    ])

```

```
# Trening modelu z przechwytywaniem historii
RNN_model.fit(X_train, y_train, epochs=100, batch_size=256,
verbose=2, validation_data=(X_test, y_test))

# Ewaluacja modelu
RNN_model.evaluate(X_test, y_test, verbose=2)

# Predykcje
y_pred = RNN_model.predict(X_test)

# Obliczenie R^2
r2_val = r2_score(y_test, y_pred)
print(f"Model RNN: {r2_val:.2%}")

# Obliczenie MSE
mse = mean_squared_error(y_test, y_pred)
print(f'\nBłąd średniokwadratowy (MSE): {mse:.4f}')
```

6.1.4 MLP

Model MLP został zaimportowany za pomocą biblioteki scikit-learn, który zawierał algorytm sieci. Do treningu modelu użyto optymalizator *Adam* z poziomem uczenia równym 0.005. Liczbę próbek ustawiono na 256. Do sieci dodano dwie warstwy ukryte po 100 neuronów w każdej. Sieć posiada tolerancję na optymalizację równą 0.01. Model wykonuje 100 iteracji. Stworzono podstawową konfigurację modelu:

```
#przypisanie wielowartstowego perceptrona do zmiennej mlp
mlp=MLPRegressor(learning_rate_init=0.005, solver='adam', batch_size=256,
activation='relu', hidden_layer_sizes=(100, 100), tol=1e-2, max_iter=100,
random_state=0)

#dopasowanie modelu do danych treningowych
mlp.fit(X_train,y_train)

#wyświetlenie wyniku treningowego oraz testowego modelu
print(f"Wynik treningowy: {mlp.score(X_train, y_train) * 100:.2f}%")
print(f"Wynik testowy: {mlp.score(X_test, y_test) * 100:.2f}%")

#przewidywanie dla danych testowych
y_pred= mlp.predict(X_test)

#obliczenie błędu średniej kwadratowej między wartościami testowymi a
    przewidzianymi
mse = mean_squared_error(y_test, y_pred)

#wyświetlenie wartości MSE
print(f'\nBłąd średniej kwadratowej (MSE): {mse:.4f}')
```

6.1.5 CNN

Ostatnim przykładem jest konwolucyjna sieć neuronowa. Inicjalizacja modelu zawiera następujące warstwy:

- **Conv1D** - 128 filtrów, rozmiar jądra równy 10, funkcja aktywacji *relu*, w kolejnej warstwie o tej samej nazwie zmniejszono rozmiar filtrów oraz rozmiar jądra
- **BatchNormalization**
- **GlobalAveragePooling1D**

- **Dropout** - warstwa z wartościami odcięcia neuronów 0.25, następnie 0.4
- **Dense** - 128 neuronów, funkcja aktywacji *relu*, ostatnia warstwa zawierała jeden neuron oraz funkcję aktywacji *linear*

```
# Budowa modelu CNN
def build_model_using_cnn(input_shape):
    model = keras.Sequential([
        # Pierwsza warstwa konwolucyjna z 128 filtrami, rozmiarem jądra 10 i funkcją aktywacji ReLU
        keras.layers.Conv1D(filters=128, kernel_size=10, activation='relu',
                                                    input_shape=input_shape),
        # Normalizacja
        keras.layers.BatchNormalization(),
        # Druga warstwa konwolucyjna z 64 filtrami, rozmiarem jądra 3 i funkcją aktywacji ReLU
        keras.layers.Conv1D(filters=64, kernel_size=3, activation='relu'),
        # Normalizacja
        keras.layers.BatchNormalization(),
        # Warstwa GlobalAveragePooling
        keras.layers.GlobalAveragePooling1D(),
        # Wyłączenie 25% neuronów
        keras.layers.Dropout(0.25),
        # warstwa połączeń z 128 neuronami i funkcją aktywacji ReLU
        keras.layers.Dense(128, activation='relu'),
        # Wyłączenie 40% neuronów
        keras.layers.Dropout(0.4),
        # Ostatnia warstwa wyjściowa z jednym neuronem i liniową funkcją aktywacji
        keras.layers.Dense(1, activation='linear')
    ])
    return model

# Określenie kształtu wejściowego na podstawie danych treningowych
input_shape = (X_train.shape[1], 1)

# Budowa modelu CNN
CNN_model = build_model_using_cnn(input_shape)

# Kompilacja modelu
CNN_model.compile(loss="mean_squared_error", optimizer='SGD', metrics=["mse"])

# Trenowanie modelu na danych treningowych przez 100 epok z rozmiarem wsadu 256
CNN_model.fit(X_train, y_train, epochs=100, batch_size=256, verbose=2)

# Ewaluacja modelu na danych testowych
CNN_model.evaluate(X_test, y_test, verbose=2)

# Przewidywanie wartości na danych testowych
y_pred = CNN_model.predict(X_test)
# Obliczanie współczynnika determinacji R^2 dla przewidywań
r2_val = r2_score(y_test, y_pred)
print(f"Model CNN: {r2_val:.2%}")

# Obliczanie błędu średniokwadratowego dla przewidywań
mse = mean_squared_error(y_test, y_pred)
print(f'\nBłąd średniej kwadratowej (MSE): {mse:.4f}')
```

6.1.6 Użycie przeszukiwania siatki w celu poprawy wyników sieci neuronowych

Do poprawy wyników algorytmów sieci neuronowych, zostały wykonane analizy z użyciem przeszukiwania siatki. Modyfikacji poddano następujące parametry:

- *batch_size* o wartościach 10, 20, 40, 60, 80 oraz 100,
- *epochs* o wartościach 10, 50 i 100,
- różne rodzaje propagacji - *SGD*, *RMSprop*, *Adagrad*, *Adadelta*, *Adam*, *Adamax*, *Nadam*.

Analogicznie przeszukiwanie siatki zostało zastosowane dla pozostałych algorytmów. Poniżej kod dla prostej, jednokierunkowej sieci neuronowej:

```
# Funkcja pomocnicza do oceny modelu przy użyciu wskaźnika R^2
def r2_scorer(y_true, y_pred):
    return r2_score(y_true, y_pred)

# Definicja klasy KerasWrapper, która rozszerza klasy BaseEstimator
# i RegressorMixin
class KerasWrapper(BaseEstimator, RegressorMixin):
    def __init__(self, batch_size=10, epochs=10):
        # Inicjalizacja parametrów batch_size i epochs
        # oraz budowa modelu Keras
        self.batch_size = batch_size
        self.epochs = epochs
        self.model = self.build_model()

    def build_model(self):
        # Tworzenie modelu Keras
        adam_optimizer = Adam(learning_rate=0.005)
        model = Sequential()
        # Dodanie pierwszej warstwy Dense z 24 neuronami
        # i funkcją aktywacji 'tanh'
        model.add(Dense(24, activation='tanh', input_dim=X_train.shape[1]))
        # Dodanie drugiej warstwy Dense z 12 neuronami
        # i funkcją aktywacji 'relu'
        model.add(Dense(12, activation='relu'))
        # Dodanie wyjściowej warstwy Dense z 1 neuronem
        # i funkcją aktywacji 'linear'
        model.add(Dense(1, activation='linear'))
        # Kompilacja modelu z użyciem funkcji straty 'mean_squared_error'
        # i optymalizatora Adam
        model.compile(loss='mean_squared_error', optimizer=adam_optimizer)
        return model

    def fit(self, X, y):
        # Trenowanie modelu Keras na danych X i y
        self.model.fit(X, y, epochs=self.epochs, batch_size=self.batch_size,
                                                                verbose=0)
        return self

    def predict(self, X):
        # Predykcja wartości y na podstawie danych X
        return self.model.predict(X)

    def score(self, X, y):
        # Obliczanie wskaźnika R^2 dla przewidywanych
        # i rzeczywistych wartości y
        y_pred = self.model.predict(X)
        return r2_score(y, y_pred)
```

```
48
49      def get_params(self, deep=True):
50          # Pobieranie parametrów modelu
51          return {'batch_size': self.batch_size, 'epochs': self.epochs}
52
53      def set_params(self, **params):
54          # Ustawianie parametrów modelu
55          if 'batch_size' in params:
56              self.batch_size = params['batch_size']
57          if 'epochs' in params:
58              self.epochs = params['epochs']
59          return self
60
61  # Inicjalizacja instancji KerasWrapper
62  keras_wrapper = KerasWrapper()
63
64  # Definicja siatki przeszukiwania dla parametrów batch_size i epochs
65  batch_size = [10, 20, 40, 60, 80, 100]
66  epochs = [10, 50, 100]
67  optimizer = ['SGD','RMSprop','Adagrad','Adadelta','Adam','Adamax','Nadam']
68  param_grid = dict(batch_size=batch_size, epochs=epochs, optimizer=optimizer)
69
70  # Utworzenie instancji GridSearchCV z KerasWrapper jako estymatorem
71  grid = GridSearchCV(estimator=keras_wrapper, param_grid=param_grid,
72  scoring=make_scorer(r2_scorer), n_jobs=-1, cv=3)
73  # Dopasowanie siatki przeszukiwania do danych treningowych
74  grid_result = grid.fit(X_train, y_train)
75
76  # Podsumowanie wyników GridSearchCV
77  print("Najlepszy: %f używając %s" % (grid_result.best_score_,
78  grid_result.best_params_))
79  means = grid_result.cv_results_['mean_test_score']
80  stds = grid_result.cv_results_['std_test_score']
81  params = grid_result.cv_results_['params']
82  for mean, stdev, param in zip(means, stds, params):
83      print("%f (%f) z: %r" % (mean, stdev, param))
84
85  # Dopasowanie modelu KerasWrapper z najlepszymi parametrami
86  # uzyskanymi z GridSearchCV
87  best_batch_size = grid_result.best_params_['batch_size']
88  best_epochs = grid_result.best_params_['epochs']
89  keras_wrapper = KerasWrapper(batch_size=best_batch_size, epochs=best_epochs)
90  keras_wrapper.fit(X_train, y_train)
91
92  # Predykcja wartości y dla danych testowych
93  y_pred = keras_wrapper.predict(X_test)
94
95  # Obliczenie i wyświetlenie wskaźnika R^2 dla predykcji na danych testowych
96  r2_val = r2_score(y_test, y_pred)
97  print(f"\nWynik modelu: {r2_val:.2%}")
```

6.2 Wyniki analizy

Wyniki algorytmów (tabela 6.1) sieci neuronowych prezentują się następująco:

Tabela 6.1: Wyniki analizy dla wybranych modeli sieci neuronowych.

	Precyzja (%)	Błąd MSE
Sieć jednokierunkowa	73.66 %	0.2626
Sieć rekurencyjna	73.96 %	0.2597
Perceptron wielowarstwowy	78.04 %	0.2995
Sieć konwolucyjna	73.58 %	0.2635

Perceptron wielowarstwowy osiągnął najwyższy wynik 78.04% przy największej wartości błędu średniokwadratowego równego 0.2995. Najmniejszą wartość osiągnął model prostej sieci jednokierunkowej - 73.66% z błędem równym 0.2626.

Z użyciem algorytmu przeszukiwania siatki, wyniki (tabela 6.2) prezentują się następująco:

Tabela 6.2: Wyniki analizy dla wybranych modeli sieci neuronowych z uwzględnieniem przeszukiwania siatki.

	Precyzja (%)
Sieć jednokierunkowa	74.08 %
Sieć rekurencyjna	72.97 %
Perceptron wielowarstwowy	79.53 %
Sieć konwolucyjna	72.27 %

Jak widać po wykonanych analizach, nie zauważono znaczącej poprawy wyników w przypadku użycia metody przeszukiwania siatki. Do poprawy wyniku doszło w przypadku prostej sieci jednokierunkowej oraz w przypadku perceptronu wielowarstwowego. Wyniki pogorszyły się w przypadku sieci rekurencyjnej oraz sieci konwolucyjnej.

Na koniec, porównałem wyniki analizy uczenia maszynowego, pochodzące z mojej pracy inżynierskiej z obecnymi wynikami sieci neuronowych (tabela 6.3):

Tabela 6.3: Porównanie wyników analizy uczenia maszynowego z wynikami sieci neuronowych

	Precyzja (%)	Algorytmy użyte w pracy inżynierskiej	Precyzja
Sieć jednokierunkowa	74.08 %	Perceptron	0.95
Sieć rekurencyjna	72.97 %	SVC	0.95
Perceptron wielowarstwowy	79.53 %	Drzewo Decyzyjne	1.00
Sieć konwolucyjna	72.27 %	Las Losowy	1.00

Rozdział 7

Podsumowanie i wnioski

Sieci neuronowe wymagają innego podejścia do przygotowania danych niż w przypadku algorytmów uczenia maszynowego. Różne wyniki dla sieci neuronowych i tradycyjnych algorytmów uczenia maszynowego mogą wskazywać na to, że proces przygotowania danych miał duże znaczenie. Możliwe, że dane były lepiej przygotowane do pracy z tradycyjnymi algorytmami.

Użycie przeszukiwania siatki również nie przyniosło znaczącej poprawy. Dodatkowo w analizie zmodyfikowano ilość warstw oraz ich parametrów - również bez skutku. Może to wynikać z indywidualnej specyfiki danych.

Porównując wyniki analizy uczenia maszynowego z wynikami sieci neuronowych, można zauważyć, że tradycyjne metody uczenia maszynowego (Perceptron, SVC, Drzewo Decyzyjne, Las Losowy) osiągnęły wyższą precyzję. Wszystko wskazuje na to, że dla tego konkretnego problemu i zestawu danych, algorytmy uczenia maszynowego były bardziej odpowiednie.

Wyniki wskazują, że zwiększenie złożoności modelu sieci neuronowej nie gwarantuje zawsze lepszej precyzji. Na przykład, pomimo że sieci rekurencyjne i konwolucyjne są zazwyczaj bardziej skomplikowane niż sieci jednokierunkowe, nie osiągnęły one wyższej precyzji w tym specyficznym przypadku. To sugeruje, że dla pewnych problemów, modele o prostszej strukturze mogą okazać się bardziej efektywne.

Bibliografia

[1] S. Raschka, V. Mirjalili, *Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow 2*, wydanie trzecie, Packt Publishing, 2019

[2] `https://www.python.org/` (Dostęp 18.04.2024)

[3] `https://jupyter.org/` (Dostęp 18.04.2024)

[4] `https://www.tensorflow.org/` (Dostęp 18.04.2024)

[5] E. Karaśkiewicz, *Zarys teorii wektorów i tensorów*, PWN, Warszawa 1976

[6] `https://keras.io/` (Dostęp 18.04.2024)

[7] `https://scikit-learn.org/` (Dostęp 18.04.2024)

[8] `https://pandas.pydata.org/` (Dostęp 18.04.2024)

[9] `https://numpy.org/` (Dostęp 18.04.2024)

[10] `https://github.com/JovianML/opendatasets` (Dostęp 18.04.2024)

[11] `https://science.nasa.gov/solar-system/asteroids/` (Dostęp 19.04.2024)

[12] `https://www.rawpixel.com/image/9975323` (Dostęp 19.04.2024)

[13] `https://nineplanets.org/asteroids/` (Dostęp 19.04.2024)

[14] `https://www.kaggle.com/datasets/nasa/asteroid-impacts` (Dostęp 13.05.2024)

[15] J. E. Prussing, B. A. Conway, *Orbital mechanics*, wydanie drugie, Oxford university Press, 2012

[16] J. S. Lewis, *Physics and Chemistry of the Solar System*, wydanie drugie, Academic press, 2004

[17] J. M. A. Danby, *Fundamentals of Celestial Mechanics*, wydanie drugie, Willmann-Bell, Inc., 1988

[18] A. Ralph, J. E. Marsden, 1987, *Foundations of Mechanics*, wydanie drugie, Redwood City, CA: Addison-Wesley Publishing Company Inc.

[19] `https://apollo.astro.amu.edu.pl/PAD/pmwiki.php?n=Dybol.DydaktykaMalecNEO1` (Dostęp 19.04.2024)

[20] `https://www.spaceacademy.net.au/watch/track/orbspec.htm` (Dostęp 19.04.2024)

[21] `https://cneos.jpl.nasa.gov/sentry/intro.html` (Dostęp 19.04.2024)

[22] https://sitn.hms.harvard.edu/flash/2017/history-artificial-intelligence/ (Dostęp 25.04.2024)

[23] https://www.researchgate.net/figure/Relationship-between-artificial-intelligence-machine-learning-neural-network-and-deep_fig3_354124420 (Dostęp 25.04.2024)

[24] https://commons.wikimedia.org/wiki/File:Neuron-budowa.svg (Dostęp 12.05.2024)

[25] A. Mustapha, L. Mohamed, K. Ali, Comparative study of optimization techniques in deep learning: Application in the ophthalmology field, Journal of Physics Conference Series 1743(1), styczeń 2021

[26] R. Tadeusiewicz, M. Szaleniec, *Leksykon sieci neuronowych*, 2015

[27] L. Moroney, *Sztuczna inteligencja i uczenie maszynowe dla programistów. Praktyczny przewodnik po sztucznej inteligencji*, Helion, 2021

[28] W. H. Steeb, *The Nonlinear Workbook*, wydanie trzecie, World Scientific Publishing, 2005

[29] S. Lynch, *Dynamical Systems with Applications using Python*, wydanie pierwsze, Birkhauser, 2018

[30] H. Karttunen, P. Kröger, H. Oja, M. Poutanen, Karl J. Donner, *Fundamental Astronomy*, wydanie piąte, wydawnictwo Springer, 2014

[31] M. Kubański, *Przegląd metod analizy danych i uczenia maszynowego w zastosowaniach do zagadnień fizycznych*, Politechnika Krakowska, 2023

[32] M. Szaleniec, R. Tadeusiewicz, *Lexicon on Neural Networks*, Wydawnictwo Fundacji "Projekt Nauka", 2015

Spis rysunków

Spis tabel