POLITECHNIKA POZNAŃSKA

WYDZIAŁ INŻYNIERII MECHANICZNEJ

Praca dyplomowa inżynierska

ZASTOSOWANIE ALGORYTMÓW SZTUCZNEJ INTELIGENCJI W STEROWANIU PRĘDKOŚCIĄ OBROTOWĄ SILNIKA PRĄDU STAŁEGO

Jakub Pawłowicz, 153029

Promotor
dr inż. Patryk Nowak

POZNAŃ 2025

KARTA PRACY DYPLOMOWEJ INŻYNIERSKIEJ

Imiona i nazwisko:	**Jakub Mariusz Pawłowicz**	Numer albumu:	153029
Studia:	Mechatronika, pierwszego stopnia, profil ogólnoakademicki, stacjonarne		
Specjalność:	—		

PRACA DYPLOMOWA

Tytuł pracy:	Zastosowanie algorytmów sztucznej inteligencji w sterowaniu prędkością obrotową silnika prądu stałego
Tytuł pracy w języku angielskim:	Application of artificial intelligence algorithms in speed control of a DC motor
Promotor:	dr inż. Patryk Nowak
Słowa kluczowe:	
Praca zespołowa:	nie
Czy przedmiot pracy jest objęty tajemnicą prawnie chronioną:	nie

Streszczenie:

ZŁOŻENIE PRACY

Regulaminowy termin złożenia pracy:	31.01.2025 r.
Data złożenia pracy:	—
Data wgrania plików:	—
Informacje o przechowywaniu pracy:	Praca przechowywana w postaci elektronicznej w systemie Archiwum Prac Dyplomowych Politechniki Poznańskiej. Adres w systemie: *https://usosapd.put.poznan.pl/diplomas/104433/* Ścieżka w zasobie dyskowym: —

Spis treści

Streszczenie

Praca inżynierska dotyczy integracji sztucznej inteligencji z tradycyjnym regulatorem PID w celu poprawy wydajności i adaptacyjności układów sterowania na przykładzie silnika DC, oraz zastosowania algorytmów SI do bezpośredniego sterowania pracą silnika.

W pracy opisano teoretyczne podstawy wybranych algorytmów sztucznej inteligencji, takich jak algorytmy genetyczne, sztuczne sieci neuronowe oraz drzewa decyzyjne, i ich zastosowanie w optymalizacji parametrów regulatora PID i generowania sekwencji napięć.

Realizując projekt, wykonane zostało stanowisko badawcze, które obejmuje m.in. silnik prądu stałego, płytkę Nucleo F439ZI oraz enkoder obrotowy, a także opracowany został kod pozwalający na generowanie sygnałów PWM, odczyt enkodera oraz sterowanie kierunkiem obrotów silnika. W symulacjach i eksperymentach praktycznych oceniono zastosowanie algorytmów sztucznej inteligencji w dynamicznych warunkach, takich jak zróżnicowane obciążenie czy zmieniające się napięcie zasilania.

Abstract

The engineering thesis focuses on integrating artificial intelligence with a traditional PID controller to improve the performance and adaptability of control systems using a DC motor as a case study. It also explores the application of AI algorithms for direct control of motor operation.

The work discusses the theoretical foundations of selected artificial intelligence algorithms, such as genetic algorithms, artificial neural networks, and decision trees, and their application in optimizing PID controller parameters and generating voltage sequences.

As part of the project, a research setup was developed, including a DC motor, a Nucleo F439ZI board, and a rotary encoder. Additionally, code was created to enable PWM signal generation, encoder reading, and motor direction control. Simulations and practical experiments were conducted to evaluate the use of artificial intelligence algorithms in dynamic conditions, such as variable loads and fluctuating supply voltages.

Spis skrótów i zwrotów technicznych

AG (Algorytm Genetyczny) Heurystyczna metoda optymalizacji inspirowana procesami biologicznymi, takimi jak selekcja naturalna i mutacja.

AI (Artificial Intelligence) Sztuczna inteligencja, technologia naśladująca ludzką inteligencję.

Anti-Windup Mechanizm zapobiegający przeładowaniu członu całkującego w regulatorach PID.

APB1 (Advanced Peripheral Bus 1) Magistrala komunikacyjna używana w mikrokontrolerach STM32.

Backpropagation Algorytm propagacji wstecznej używany do obliczania gradientów w uczeniu sieci neuronowych.

CAN (Controller Area Network) Magistrala komunikacyjna używana w motoryzacji i automatyce.

CCR1 (Capture/Compare Register 1) Rejestr timera w STM32 używany do generacji sygnałów PWM.

DC (Direct Current) Prąd stały, przepływający w jednym kierunku.

Dropout Technika regularizacji w sieciach neuronowych polegająca na wyłączaniu losowych neuronów podczas uczenia.

Drzewo regresyjne Algorytm uczenia maszynowego używany do przewidywania wartości liczbowych.

Encoder Mode Tryb pracy timera w STM32 pozwalający na odczyt pozycji enkodera.

Ewaluacja Proces oceny wydajności modelu na danych testowych w uczeniu maszynowym.

Fitness Function Funkcja oceny używana w algorytmach genetycznych do mierzenia jakości rozwiązań.

Funkcja aktywacji Funkcja w sztucznych sieciach neuronowych, która wprowadza nieliniowość.

GPIO (General Purpose Input/Output) Uniwersalne piny wejścia/wyjścia w mikrokontrolerach STM32.

HAL (Hardware Abstraction Layer) Warstwa abstrakcji sprzętowej w bibliotekach STM32.

Huber Loss Funkcja strat łącząca właściwości MSE i MAE, odporna na wartości odstające.

HVAC (Heating, Ventilation, and Air Conditioning) Systemy grzewczo-wentylacyjne.

IDE (Integrated Development Environment) Zintegrowane środowisko programistyczne, np. STM32CubeIDE.

JSON (JavaScript Object Notation) Format wymiany danych, często używany do przechowywania konfiguracji.

Macierz konfuzji Narzędzie używane w uczeniu maszynowym do oceny skuteczności klasyfikacji.

Modbus Protokół komunikacyjny używany w automatyce przemysłowej do przesyłania danych między urządzeniami.

MPC (Model Predictive Control) Sterowanie predykcyjne, optymalizujące działanie systemu w czasie rzeczywistym.

MSE (Mean Squared Error) Błąd średniokwadratowy, stosowany jako metryka w uczeniu maszynowym.

Normalizacja Proces przekształcania danych wejściowych do wspólnego zakresu, np. [0, 1].

Overfitting Zjawisko, w którym model dopasowuje się zbyt dobrze do danych treningowych, tracąc zdolność generalizacji.

Overshoot Przeregulowanie, największe przekroczenie wartości zadanej w układzie dynamicznym.

PID (Proportional-Integral-Derivative) Regulator proporcjonalno-całkująco-różniczkujący w systemach sterowania.

PWM (Pulse Width Modulation) Modulacja szerokości impulsów, używana do regulacji napięcia i prędkości.

Selekcja ruletkowa Metoda selekcji osobników w algorytmach genetycznych, proporcjonalna do ich wartości fitness.

Selekcja turniejowa Metoda selekcji, gdzie osobniki są losowo grupowane i najlepszy z grupy przechodzi dalej.

Settling Time (Czas ustalania) Czas, jaki układ potrzebuje, aby osiągnąć wartość zadaną w granicach tolerancji.

SGD (Stochastic Gradient Descent) Stochastyczna metoda optymalizacji gradientowej.

SI (Sztuczna Inteligencja) Polski odpowiednik skrótu AI, używany w pracy do opisu technologii naśladującej inteligencję ludzką.

Softmax Funkcja aktywacji w sieciach neuronowych, przekształcająca dane na prawdopodobieństwa.

SSN (Sztuczne Sieci Neuronowe) Algorytmy inspirowane biologicznymi sieciami neuronowymi, używane w uczeniu maszynowym.

STM32 Rodzina mikrokontrolerów produkowanych przez firmę STMicroelectronics.

TIM2, TIM3, TIM4 Timery w STM32 do obsługi sygnałów PWM, przerwań i odczytu enkoderów.

UART (Universal Asynchronous Receiver-Transmitter) Uniwersalny moduł komunikacji szeregowej.

Rozdział 1

Wstęp

1.1 Cel i zakres pracy

Sztuczna inteligencja stanowi obecnie jedno z najważniejszych narzędzi w dziedzinie technologii, oferując możliwości automatyzacji i optymalizacji złożonych procesów. Jej zastosowanie obserwujemy w różnorodnych obszarach, takich jak systemy rekomendacji, autonomiczne pojazdy, filtry spamu w mailach czy zaawansowane sterowanie procesami przemysłowymi. W przypadku układów regulacji, takich jak sterowanie silnikami prądu stałego, integracja algorytmów sztucznej inteligencji z klasycznym regulatorem PID może znacząco poprawić wydajność i niezawodność systemu.

Tradycyjny regulator PID, szeroko stosowany w układach automatyki, charakteryzuje się prostotą implementacji i dużą skutecznością w wielu aplikacjach. Jego ograniczeniem jest konieczność ręcznego strojenia parametrów, co wymaga doświadczenia operatora i może być czasochłonne. Zastosowanie algorytmów sztucznej inteligencji, takich jak algorytmy genetyczne czy sztuczne sieci neuronowe, może pozwolić na automatyzację tego procesu, zapewniając adaptacyjność układu do zmiennych warunków pracy.

Wybór tematu pracy inżynierskiej wynika z zainteresowania stale rozwijającym się obszarem sztucznej inteligencji i co za nią idzie nadchodzącej potrzeby optymalizacji i automatyzacji procesów sterowania w złożonych układach mechatronicznych. Sterowanie silnikami DC jest istotnym elementem wielu systemów, takich jak napędy robotyczne, układy automatyki przemysłowej czy systemy sterowania pojazdów. Klasyczne podejście do strojenia regulatorów PID nie jest wystarczające w sytuacjach, gdy układ sterowania podlega dynamicznym zmianom, takim jak zmiana momentu bezwładności czy różne napięcia zasilania. Wprowadzenie sztucznej inteligencji do strojenia parametrów regulatora PID wpływa na:

- Adaptacyjność - algorytm będzie dostosowywał nastawy wszystkich członów w czasie rzeczywistym na podstawie zmiennych takich jak obciążenie i napięcie zasilania.
- Energooszczędność - precyzyjne sterowanie np. silnikami DC zapewni minimalizację oscylacji i zjawiska przeregulowania, dodatkowo ograniczy pobór prądu w rozruchu i hamowaniu.
- Szybkość działania - wytrenowany algorytm będzie w stanie szybko wyznaczyć wartości członów na podstawie poprzednich doświadczeń i obecnie panujących warunków pracy.

Dodatkowo zostały zastosowane algorytmy sztucznej inteligencji które bezpośrednio sterują silnikiem DC poprzez dynamiczną zmianę wartości podawanego napięcia. Praca dyplomowa opiera się na dobraniu silnika, urządzeń komunikacyjnych i sterujących przy pomocy których sprawdzona zostanie szybkość działania i niezawodność wybranych algorytmów sztucznej inteligencji.

Rozdział 2

Teoria

2.1 Historia rozwoju systemów sterowania i regulatorów

Poczatki systemów sterowania sięgają czasów starożytnych, gdzie podstawowe mechanizmy były wykorzystywane do automatyzacji prostych procesów, takich jak systemy nawadniania czy zegary wodne. Już w starożytności wprowadzono elementy regulacji, które umożliwiały utrzymanie stabilnych warunków działania. Jednak dopiero rewolucja przemysłowa w XVIII wieku umożliwiła dynamiczny rozwój teorii i praktyki sterowania, co wiązało się z rosnącym zapotrzebowaniem na automatyzację w przemyśle i rolnictwie.

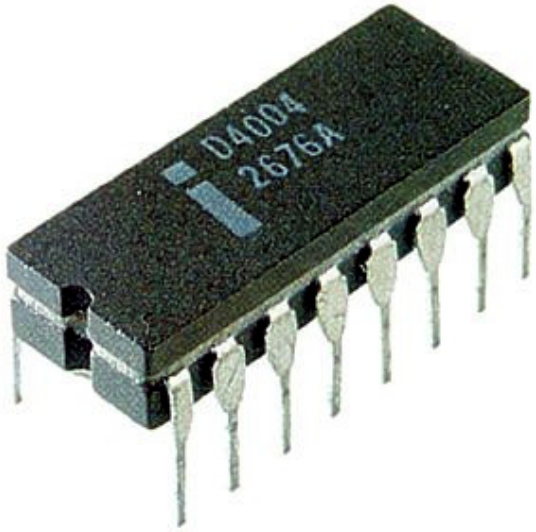
Pierwszym przełomowym urządzeniem był regulator odśrodkowy Jamesa Watta, zaprojektowany w 1788 roku w celu automatycznego sterowania prędkością maszyn parowych. Urządzenie to, bazujące na mechanizmie z przeciwwagą, umożliwiało regulację dopływu pary w zależności od prędkości obrotowej wału. Praca maszyny powodowała podniesienie się kulek co skutkowało zamyknięciem przepustnicy. Przymknięcie przepustnicy blokuje przepływ pary i zwolnienie maszyny - kulki opadały. Zastosowanie tego typu regulatorów znacznąco poprawiło wydajność i stabilność pracy maszyn, co było kluczowe dla dynamicznie rozwijającego się przemysłu. James Watt nie tylko przyczynił się do praktycznego zastosowania regulacji, ale również zainspirował prace teoretyczne nad stabilnością układów sterowania.

W XIX wieku rozwój teorii matematycznej układów sterowania nabrał tempa. W 1868 roku James Clerk Maxwell opublikował prace [1], w których zastosował metody analizy równań różniczkowych do badania stabilności systemów dynamicznych. Maxwell zaprezentował metody pozwalające oceniać, czy dany system będzie stabilny, co stanowiło podstawę do dalszego rozwoju teorii sterowania. Wkrótce po nim Edward Routh rozwinął kryterium stabilności, znane jako kryterium Routha-Hurwitza[2], które do dziś jest wykorzystywane w analizie systemów liniowych.

Rozwój technologii elektrycznej na początku XX wieku otworzył nowe możliwości w projektowaniu systemów sterowania. Pierwsze analogowe układy sterowania były stosowane w przemyśle energetycznym, gdzie precyzyjna regulacja napięcia i prądu miała kluczowe znaczenie. W latach 30. XX wieku wprowadzono regulatory PID[3], które łączyły trzy podstawowe człony: proporcjonalny (P), całkujący (I) oraz różniczkujący (D). Ich elastyczność i prostota sprawiły, że stały się standardem w automatyce przemysłowej.

Regulatory PID działają w oparciu o obliczenie sygnału sterującego na podstawie błędu między wartością zadaną a rzeczywistą. Człon proporcjonalny reaguje na bieżący błąd, człon całkujący eliminuje błędy długoterminowe, a człon różniczkujący przewiduje przyszłe zmiany uchybu. Wdrożenie regulatorów PID było możliwe dzięki zastosowaniu lamp elektronowych[4], a później tranzystorów, co umożliwiło miniaturyzację układów sterujących.

Kolejny przełom nastąpił w latach 70. XX wieku, wraz z rozwojem mikroprocesorów. Sterowanie cyfrowe pozwoliło na implementację bardziej złożonych algorytmów, takich jak sterowanie predykcyjne (MPC- Model Predictive Control[5]; regulator wprowadza zmiany w swoim działaniu z wyprzedzeniem) czy adaptacyjne regulatory PID. Mikroprocesory umożliwiły także integrację systemów sterowania z czujnikami i układami wykonawczymi w jedną spójną platformę. To właśnie w tym okresie zaczęto intensywnie rozwijać teorię sterowania nieliniowego oraz metody optymalizacji. Pierwszy komercyjny procesor zaprezentowano na rysunku 2.1.



Rysunek 2.1: Intel 4004- pierwszy komercyjny jednoukładowy procesor[6]

Obecnie systemy sterowania coraz częściej korzystają z algorytmów sztucznej inteligencji, takich jak sieci neuronowe, algorytmy genetyczne czy szeroko rozumiane uczenie maszynowe. Nowoczesne podejścia umożliwiają adaptacyjne dostrajanie parametrów regulatorów w czasie rzeczywistym, co jest szczególnie ważne w przypadku skomplikowanych procesów przemysłowych.

Współczesne systemy sterowania znajdują zastosowanie w niemal każdej dziedzinie technologii, od robotyki i lotnictwa po przemysł energetyczny i motoryzacyjny. Postępy w dziedzinie sterowania są ściśle powiązane z rozwojem technologii komputerowych, co pozwala na ciągłe podnoszenie efektywności i precyzji układów sterujących.

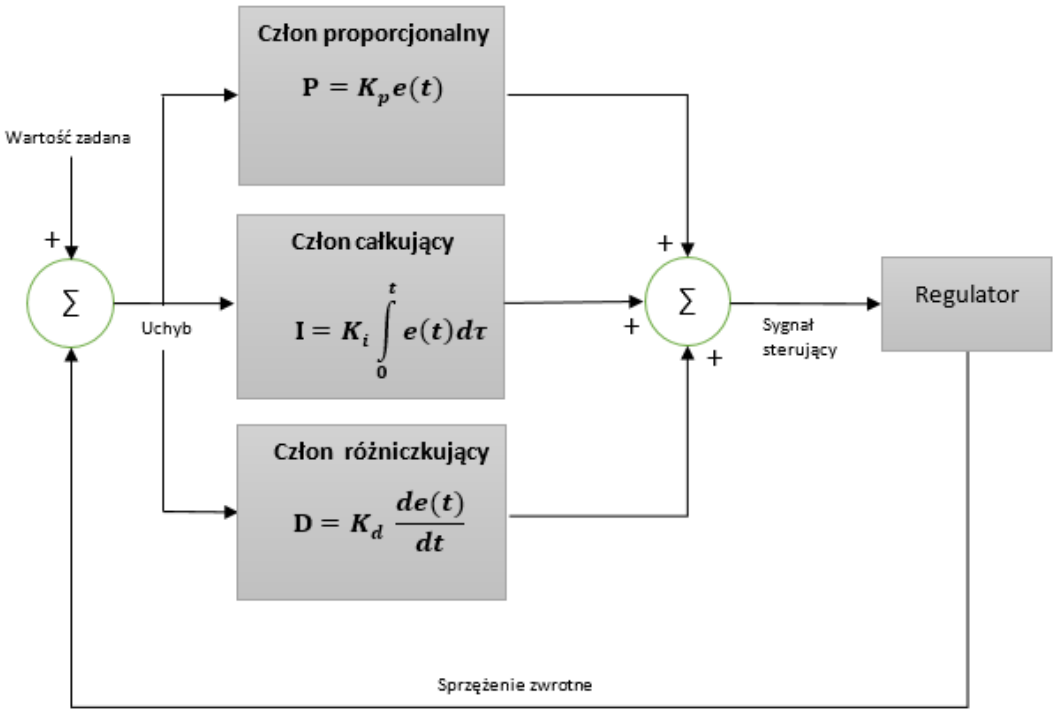
2.2 Regulator PID

Regulator PID, czyli Proporcjonalno-Całkująco-Różniczkujący, jest jednym z najpopularniejszych algorytmów sterowania w układach automatyki. Jego podstawowym celem jest redukcja uchybu, czyli różnicy pomiędzy wartością zadaną (np. pożądaną prędkością obrotową silnika) a wartością bieżącą. Idea polega na obliczaniu i sumowaniu trzech składowych: proporcjonalnej, całkującej oraz różniczkującej. Człon proporcjonalny reaguje na aktualny błąd i stanowi główny składnik wpływający na sygnał sterujący. Jeśli ustawimy zbyt duże wzmocnienie K_p, układ może stać się niestabilny i generować przeregulowania. Zbyt małe wzmocnienie spowoduje natomiast wolniejszą odpowiedź i słabą dokładność regulacji.

Człon całkujący dąży do wyeliminowania uchybu w stanie ustalonym, czyli sprawia, że w dłuższym czasie uchyb powinien dążyć do zera. Problemem może być jednak nadmierne kumulowanie błędów z przeszłości, co powoduje wydłużenie czasu regulacji i zwiększenie przeregulowania. Z

tego powodu zbyt duża wartość czasu całkowania K_i może prowadzić do niestabilnego zachowania systemu.

Człon różniczkujący pozwala „wyprzedzić" przyszłe zmiany błędu, reagując na jego pochodną. Jeśli wartość uchybu szybko narasta, to człon różniczkujący zapewnia dodatkowy sygnał sterujący zapobiegający dalszemu narastaniu błędu. Rysunek 2.2 przedstawia schemat regulatora PID z zastosowanym sprzężeniem zwrotnym:



Rysunek 2.2: Schemat regulatora PID ze sprzężeniem zwrotnym [7]

Działanie pętli regulacji PID ze sprzężeniem zwrotnym zaczyna się od obliczenia błędu między wartością zadaną a rzeczywistą. Następnie generowany jest sygnał sterujący u(t) ze wzoru[3]:

$$u(t) = K_p \left[\varepsilon(t) + \frac{1}{K_i} \int_0^t \varepsilon(\tau)\, d\tau + K_d \frac{d\varepsilon(t)}{dt} \right]$$

gdzie:

- $\varepsilon(t) = w(t) - y(t)$ oznacza uchyb, czyli różnicę między wartością zadaną $w(t)$ a wartością rzeczywistą $y(t)$,
- K_p – wzmocnienie członu proporcjonalnego,
- K_i– czas całkowania, związany z członem całkującym,
- K_d– czas różniczkowania).

Sygnał u(t) steruje elementem wykonawczym, w kontekście silnika DC może być to zmiana wypełnienia PWM. Przez sprzężenie zwrotne nowa wartość wyjściowa znowu porównywana jest z wartością zadaną i zadawana jest dalsza korekcja do momentu osiągnięcia przez układ wartości zadanej. Ważnym wzorem jest również reprezentacja regulatora PID w dziedzinie operatora

Laplace'a, która stanowi jedno z kluczowych narzędzi podczas inżynierskiej analizy układów sterowania. Umożliwia precyzyjne badanie stabilności i dynamiki systemu. Transmitancję regulatora PID można zapisać w postaci:

$$G_{\mathrm{PID}}(s) \;=\; \frac{U(s)}{E(s)} \;=\; K_p\Big(1 + \frac{1}{T_i\, s} + T_d\, s\Big).$$

Tak sformułowana transmitancja regulatora PID może zostać wykorzystana do analizy kompletnego układu sterowania z obiektem w pętli sprzężenia zwrotnego.

Z punktu widzenia dynamiki układu, badanie transmitancji w dziedzinie Laplace'a pozwala ustalić, w jaki sposób zmiana parametrów regulatora wpłynie na położenie biegunów układu zamkniętego. Położenie tych biegunów decyduje o odpowiedzi na wymuszenia, określa szybkość narastania sygnału wyjściowego czy tłumienie oscylacji. Na przykład zbyt duża wartość wzmocnienia proporcjonalnego może spowodować przesunięcie biegunów układu w obszar niestabilności, natomiast źle dobrany czas całkowania może przyczynić się do nadmiernego przeregulowania oraz długiego czasu ustalania. Z kolei właściwe dobranie czasu różniczkowania może poprawić tłumienie drgań i przyspieszyć odpowiedź, ale również wzmocnić niechciane szumy pomiarowe.

Dzięki analizie układu właśnie w dziedzinie Laplace'a możliwe jest więc systematyczne wyznaczenie lub oszacowanie optymalnego obszaru pracy regulatora PID.

Aby dostroić układ regulatora, w praktyce najczęściej korzysta się z metod doświadczalnych (np. Zieglera-Nicholsa) lub bardziej zaawansowanych algorytmów optymalizacyjnych. PID, mimo swojej powszechności, nie jest rozwiązaniem pozbawionym wad, co należy wziąć pod uwagę przy jego praktycznym zastosowaniu.

Pierwszym utrudnieniem jest wrażliwość członu różniczkującego na szumy w sygnale pomiarowym. Różniczkowanie samo w sobie wzmacnia składniki wysokoczęstotliwościowe, przez co może wymagać zastosowania filtrów cyfrowych lub analogowych ograniczających wpływ tych zakłóceń.

Kolejnym problemem jest zjawisko przeładowania członu całkującego, określane mianem „windup". Pojawia się ono wtedy, gdy sygnał sterujący napotka nasycenie, czyli fizyczne ograniczenie urządzenia wykonawczego, a regulator PID dalej wypracowuje rosnącą wartość całki błędu. Taka sytuacja wydłuża czas, po którym wartość wyjściowa powróci ze stanu nasycenia, co z kolei niekorzystnie wpływa na dynamikę układu. Rozwiązaniem jest stosowanie dodatkowych algorytmów „anti-windup" lub korekcji sygnału całkującego.

Występują także problemy związane z dłuższymi opóźnieniami w układzie, ponieważ regulator nie jest zaprojektowany do kompensacji znacznych opóźnień, co może prowadzić do niepożądanych oscylacji i utrudniać stabilizację sygnału wyjściowego. W systemach o rozbudowanej dynamice, dużej bezwładności czy obiektach niestabilnych sam klasyczny PID może nie wystarczyć; czasami konieczne jest dodanie innego typu regulatora lub wdrożenie bardziej zaawansowanych strategii sterowania.

2.3 Algorytmy sztucznej inteligencji - algorytm genetyczny

Algorytm genetyczny (AG) jest jedną z najskuteczniejszych heurystycznych metod optymalizacji, inspirowaną mechanizmami ewolucji biologicznej. Jego głównym celem jest znalezienie rozwiązania, które zbliża się do globalnego optimum problemu. Dzięki swojej zdolności do eksplorowania rozległych i złożonych przestrzeni poszukiwań, algorytm genetyczny doskonale sprawdza się tam, gdzie klasyczne metody analityczne lub gradientowe zawodzą.

Proces ewolucji w AG polega na stopniowym ulepszaniu populacji rozwiązań poprzez iteracyjne generowanie nowych pokoleń. Mechanizm ten naśladuje naturalne procesy: lepiej przystosowane osobniki mają większe szanse na przetrwanie i przekazywanie swoich cech potomstwu. Dzięki takim operacjom jak selekcja, krzyżowanie i mutacja, algorytm dąży do znalezienia najlepszego rozwiązania lub wartości zbliżonej do idealnego[8].

AG wyróżnia się na tle innych metod optymalizacyjnych przede wszystkim:

- Elastycznością, ponieważ może być stosowany w problemach o różnej naturze (ciągłe, dyskretne, kombinatoryczne).
- Odpornością na lokalne minima, dzięki wbudowanej losowości i zdolności do eksploracji szerokiego obszaru przestrzeni poszukiwań.

Proces działania algorytmu genetycznego rozpoczyna się od utworzenia populacji początkowej, składającej się z ustalonej liczby osobników. Populacja to zbiór potencjalnych rozwiązań problemu optymalizacyjnego. Każdy osobnik w populacji jest reprezentowany przez chromosom, który zawiera zestaw genów. Geny te opisują parametry rozwiązania i mogą być kodowane na różne sposoby w zależności od rodzaju problemu:

1. **Kodowanie binarne** – najprostsze i najczęściej stosowane, w którym chromosom jest ciągiem bitów, np. [1, 0, 1, 1, 0]. Każdy bit może reprezentować obecność lub brak cechy albo przyjmować wartość parametru.
2. **Kodowanie rzeczywiste** – chromosom to wektor liczb rzeczywistych, np. [2.5, 1.0, 4.3]. Jest stosowane w problemach, w których rozwiązania przyjmują ciągłe wartości.
3. **Kodowanie permutacyjne** – używane w problemach, w których kluczowe jest ustalenie kolejności elementów, np. [A, B, C, D] w problemie komiwojażera.

Populacja początkowa zazwyczaj jest generowana losowo, aby zapewnić szerokie pokrycie przestrzeni poszukiwań i różnorodność rozwiązań. W niektórych przypadkach stosuje się wcześniejsze wyniki, aby ukierunkować proces inicjalizacji na bardziej obiecujące obszary przestrzeni.

Każdy osobnik w populacji jest oceniany na podstawie funkcji dopasowania (fitness), która określa jakość rozwiązania reprezentowanego przez dany chromosom. Funkcja fitness pełni rolę miernika „przystosowania" danego osobnika do problemu optymalizacyjnego i umożliwia wyróżnienie lepszych rozwiązań od gorszych.

W przypadku problemów maksymalizacyjnych wartość funkcji fitness może być równa wartości funkcji celu $f(x)$, którą algorytm próbuje maksymalizować:

$$F(x) = f(x)$$

,gdzie $f(x)$ jest wartością funkcji celu, a $F(x)$ wartością fitness.

W problemach minimalizacyjnych sytuacja jest bardziej złożona. Funkcja celu $f(x)$ opisuje wartość kosztu lub błędu, który należy minimalizować. Aby zgodnie z założeniem algorytmu dążyć do maksymalizacji wartości fitness, stosuje się przekształcenia, które zamieniają problem minimalizacji na maksymalizację. Jednym z najprostszych sposobów jest odwrotność funkcji celu:

$$F(x) = \frac{1}{1 + f(x)}$$

Dzięki temu osobniki o niższej wartości kosztu uzyskują wyższe wartości fitness, co umożliwia ich faworyzowanie w procesie selekcji. W algorytmach genetycznych, poza tradycyjnym podejściem do maksymalizacji funkcji fitness, można zastosować strategię, w której algorytm dąży do minimalizacji wartości fitness. Takie podejście jest mniej popularne, ale znajduje zastosowanie w szczególnych przypadkach, zwłaszcza gdy wartości funkcji fitness są wprost proporcjonalne do kosztów lub odległości od akceptowalnych rozwiązań. W tym podejściu większa wartość fitness oznacza gorsze dopasowanie, co prowadzi do odrzucenia osobników o wysokiej wartości fitness.

W ramach minimalizacji fitness można wprowadzić mechanizm kar, który dodatkowo zwiększa wartość fitness dla osobników naruszających ograniczenia lub oddalonych od akceptowalnych obszarów. Taki mechanizm kary działa w odwrotny sposób niż w klasycznym podejściu maksymalizacyjnym: zamiast obniżać wartość fitness, zwiększa ją, co dodatkowo obniża atrakcyjność danego rozwiązania.

Przy minimalizacji fitness funkcję można zapisać jako sumę wartości funkcji celu $f(x)$ i funkcji kary $P(x)$:

$$F(x) = f(x) + P(x)$$

gdzie:

- $f(x)$ to funkcja celu, której wartość chcemy zminimalizować,
- $P(x)$ to funkcja kary, która przyjmuje wartość $P(x) > 0$, jeśli rozwiązanie narusza ograniczenia problemu, np. $g(x) > 0$.

Jeżeli rozwiązanie spełnia wszystkie ograniczenia $g(x) \leq 0$, funkcja kary $P(x)$ przyjmuje wartość 0. W takim przypadku fitness osobnika odpowiada jedynie wartości funkcji celu $f(x)$. Natomiast gdy ograniczenia są naruszane, wartość $P(x)$ zwiększa $F(x)$, co oznacza, że osobnik jest oceniany jako gorzej dopasowany. W podejściu minimalizacyjnym niższa wartość fitness jest pożądana, ponieważ oznacza mniejszy koszt lub mniejsze naruszenie ograniczeń. Mechanizm kar [9] zwiększa wartość fitness osobników, które:

- Naruszają ograniczenia problemu, takie jak warunki brzegowe lub inne wymagania,
- Znajdują się daleko od akceptowalnego zakresu, czyli od obszarów przestrzeni rozwiązań, które są uznawane za sensowne.
- Nie spełniają określonych wymagań dodatkowych, które mogą wynikać z natury problemu optymalizacyjnego.

W takim podejściu algorytm nadal minimalizuje funkcję celu, ale jednocześnie rozwiązania naruszające zasady otrzymują kary.

Mechanizm kar w minimalizacji funkcji celu niesie ze sobą kilka istotnych korzyści:

- Osobniki, które naruszają ograniczenia problemu, są mniej atrakcyjne w procesie selekcji. Dzięki temu algorytm skupia się na obszarach przestrzeni, gdzie istnieje większa szansa na znalezienie poprawnych i optymalnych rozwiązań.

- Funkcja kary może być dostosowana do specyfiki problemu. Można ją skonstruować w różny sposób, np. liniowo lub kwadratowo, w zależności od tego, jak silnie algorytm powinien unikać naruszeń ograniczeń. Na przykład funkcja kary może rosnąć liniowo wraz z odległością od dopuszczalnych wartości lub kwadratowo, aby silniej karać większe odchylenia. Można stosować wiele różnych kar które są ukierunkowane na zwalczanie przykładów prowadzących do konkretnych niechcianych wyników(np. oscylacja, niemożliwe do osiągnięcia wartości prędkości w przypadku silnika DC)

- Mechanizm kar pozwala zachować tradycyjne podejście do minimalizacji funkcji celu, jednocześnie wprowadzając dodatkowe kryteria oceny. Algorytm w naturalny sposób faworyzuje osobniki o niższym koszcie i bez naruszeń.

Przykład zastosowania: Załóżmy, że optymalizujemy funkcję celu $f(x)$, np. koszt produkcji, z ograniczeniem $g(x) \leq 0$, które oznacza maksymalny dopuszczalny poziom emisji. Funkcja fitness przyjmuje postać:

$$F(x) = f(x) + \alpha \cdot g(x)$$

w której:

- $f(x)$ to koszt, który chcemy zminimalizować,

- $g(x)$ to poziom emisji (naruszenie ograniczenia, gdy $g(x) > 0$,

- α to współczynnik kary, określający wagę naruszenia ograniczenia.

Dla rozwiązań spełniających ograniczenie $g(x) \leq 0$, wartość kary $P(x) = 0$, co oznacza, że $F(x) = f(x)$. Dla rozwiązań naruszających ograniczenie ($g(x) > 0$), funkcja fitness zostaje zwiększona proporcjonalnie do stopnia naruszenia ograniczenia.

Podsumowując: mechanizm kar w minimalizacji funkcji celu pozwala skutecznie ukierunkować algorytm na poprawne obszary przestrzeni rozwiązań. Dzięki temu osobniki niespełniające ograniczeń otrzymują niskie wartości fitness, co zmniejsza ich szanse na dostanie się do kolejnych generacji. Funkcja kary może być elastycznie dostosowywana, aby algorytm w odpowiedni sposób reagował na różne rodzaje naruszeń.

Funkcja fitness jest kluczowym elementem algorytmu genetycznego, ponieważ na jej podstawie wybierane są osobniki do dalszego krzyżowania. Dobrze skonstruowana funkcja fitness powinna:

- Efektywnie różnicować rozwiązania, aby algorytm mógł skutecznie rozróżnić poprawne rozwiązania od złych.

- Być monotoniczna i stabilna, aby zmiany wartości fitness były proporcjonalne do jakości rozwiązania.

- Unikać wartości ekstremalnych, które mogą prowadzić do problemów numerycznych (np. dzielenie przez zero).

Proces selekcji ma na celu wybór najlepszych osobników z obecnej populacji, które będą miały szansę przekazać swoje cechy kolejnej generacji. Selekcja faworyzuje osobniki o lepszej wartości fitness (w zależności czy fitness ma być minimalizowane czy maksymalizowane), co symuluje mechanizm „przetrwania najsilniejszych" w naturze. Najpopularniejsze metody selekcji to:

- Selekcja ruletkowa (proporcjonalna): prawdopodobieństwo wyboru danego osobnika jest proporcjonalne do jego wartości fitness. Osobniki o lepszej wartości fitness mają większy „obszar" na kole ruletki, co zwiększa ich szanse na wybór.
- Selekcja turniejowa: Losowo wybierana jest grupa osobników, spośród których do następnego etapu przechodzi najlepszy.
- Selekcja elitarną: Najlepsze osobniki z danej populacji są przenoszone do nowej generacji bez zmian, co gwarantuje zachowanie najlepszego rozwiązania.

Krzyżowanie polega na łączeniu cech dwóch rodziców w celu wygenerowania nowego potomstwa. Jest to kluczowy proces, który pozwala na eksplorację przestrzeni rozwiązań i łączenie korzystnych cech istniejących osobników. Przykładowe metody krzyżowania to:

- Jednopunktowe: Chromosom jest dzielony w jednym losowo wybranym punkcie. Geny przed punktem pochodzą od jednego rodzica, a po punkcie od drugiego.
- Dwupunktowe: Podział chromosomu następuje w dwóch miejscach, a geny między nimi są wymieniane.
- Jednorodne: Każdy gen potomka jest wybierany losowo z odpowiadających genów rodziców.

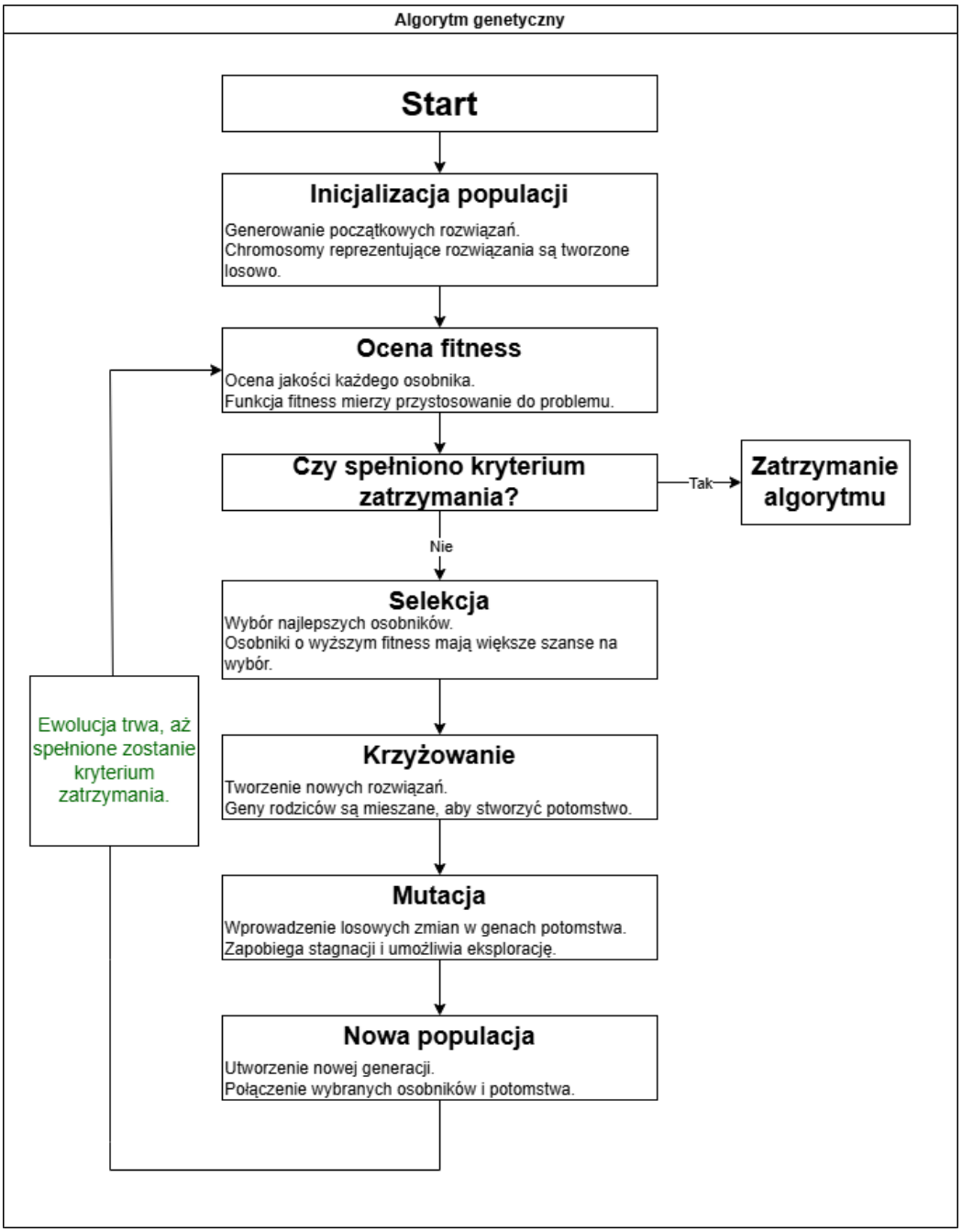
Innym czynnikiem wpływający na populacje w danej generacji jest mutacja. Polega na wprowadzeniu losowych zmian w genach chromosomu, co pozwala algorytmowi uniknąć stagnacji i eksplorować nowe obszary przestrzeni rozwiązań. Mutacja zachodzi z niskim prawdopodobieństwem i może polegać na:

- odwróceniu bitu (dla kodowania binarnego),
- dodaniu losowej wartości (dla kodowania rzeczywistego),
- przestawieniu elementów (dla kodowania permutacyjnego).

Po zakończeniu procesów selekcji, krzyżowania i mutacji nowa populacja zastępuje starą. Proces ten jest powtarzany aż do spełnienia jednego z kryteriów zatrzymania, takich jak:

- osiągnięcie rozwiązania o zadowalającej wartości fitness,
- brak poprawy jakości rozwiązań przez określoną liczbę generacji,
- przekroczenie maksymalnej liczby iteracji.

Ostatecznym wynikiem algorytmu genetycznego jest najlepszy osobnik w ostatniej populacji, który reprezentuje rozwiązanie optymalne lub bliskie optymalnemu. Podsumowanie działania algorytmu genetycznego przedstawia rysunek nr. 2.3.

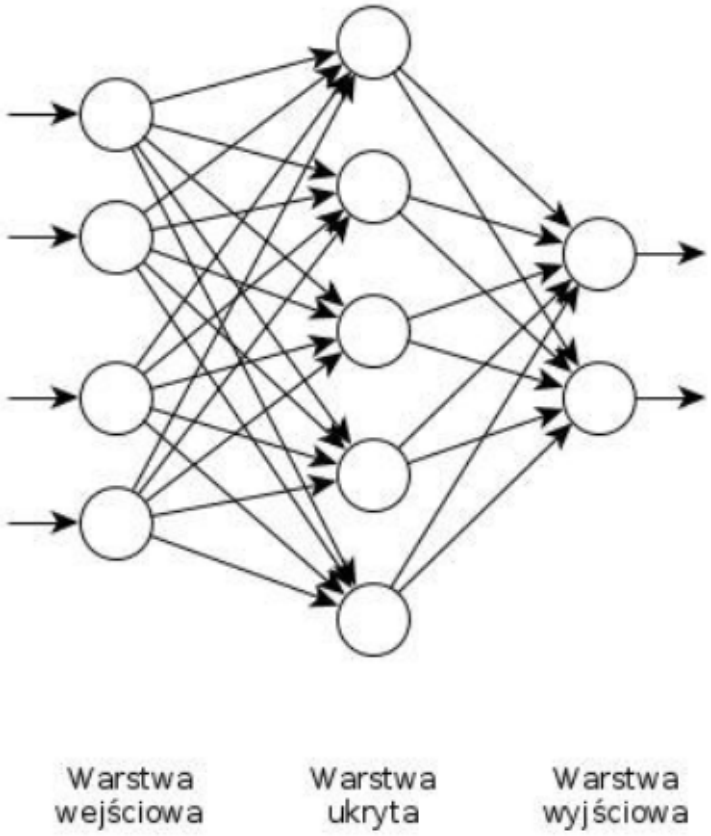


Rysunek 2.3: Schemat blokowy działania przykładowego algorytmu genetycznego

2.4 Sztuczne Sieci Neuronowe

Sztuczne sieci neuronowe (SSN) to algorytmy inspirowane biologicznym układem nerwowym, które wykorzystują warstwową strukturę do przetwarzania danych wejściowych i generowania wyników. Podstawowym elementem sieci jest sztuczny neuron, który naśladuje funkcjonowanie komórki nerwowej, wykonując operacje matematyczne na otrzymanych danych.

Sztuczne sieci neuronowe składają się z trzech głównych typów warstw: wejściowej, ukrytych oraz wyjściowej [10]. Warstwa wejściowa odbiera dane wejściowe w formie wektorów liczbowych, takich jak obrazy, tekst lub dane sensoryczne. Dane te następnie trafiają do warstw ukrytych, w których neurony przetwarzają informacje i uczą się zależności między wejściami a wyjściami. Liczba warstw ukrytych oraz liczba neuronów w każdej z nich wpływają na zdolność sieci do modelowania złożonych zależności. Proces przetwarzania danych kończy się w warstwie wyjściowej, która produkuje wyniki w postaci klasyfikacji, wartości numerycznych lub innych informacji w zależności od zastosowania. Neurony w sieci wykonują przekształcenia liniowe na danych wejściowych i stosują funkcje aktywacji, aby wprowadzić nieliniowość do modelu. Rysunek 2.4 przedstawia budowę przykładowej sieci neuronowej w architekturze MLP:

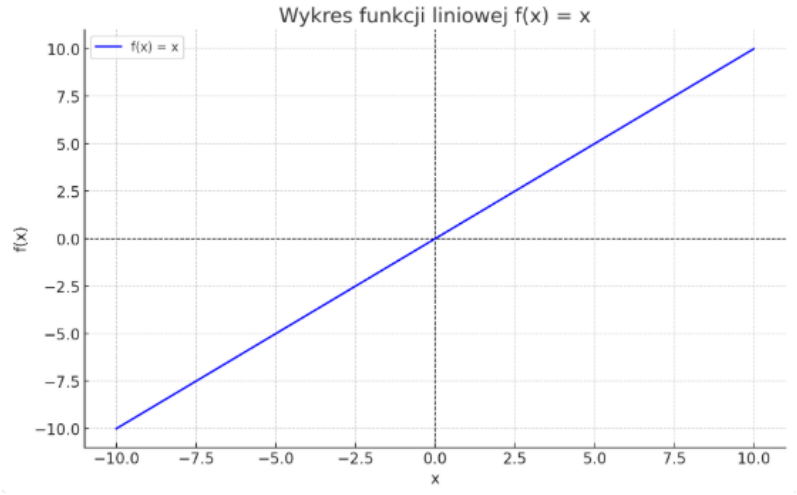


Rysunek 2.4: Model sztucznej sieci neuronowej MLP z jedną warstwą ukrytą [11]

2.4.1 Funkcje aktywacji

Funkcje aktywacji pełnią kluczową rolę w sztucznych sieciach neuronowych, pozwalając na modelowanie złożonych, nieliniowych zależności między danymi. Najczęściej stosowane funkcje aktywacji [12] to:

- Funkcja liniowa: jest to najprostsza możliwa funkcja aktywacji. Jej postać matematyczna to $f(x) = x$. Neurony z funkcją liniową nie wprowadzają żadnej nieliniowości do sieci – pełnią rolę zwykłego przekształcenia liniowego. Sieć złożona wyłącznie z warstw liniowych (bez innych nieliniowych funkcji aktywacji) jest w stanie wyrazić jedynie funkcje liniowe. Oznacza to, że niezależnie od liczby warstw w takiej sieci, całość dalej będzie działać jak pojedyncze przekształcenie liniowe. Wykres funkcji liniowej przedstawiono na rysunku 2.5:

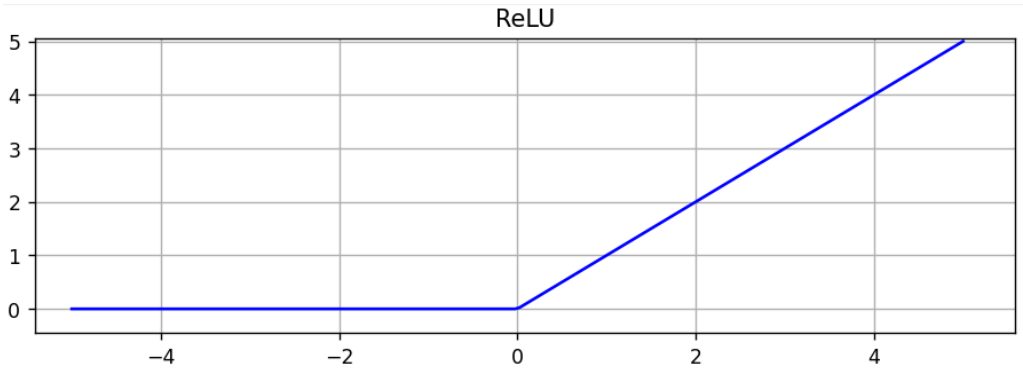


Rysunek 2.5: Wykres funkcji liniowej

- ReLU (Rectified Linear Unit): definiuje wyjście jako maksymalną wartość między zerem a wejściem. Przy jej zastosowaniu istnieje ryzyko wystąpienia martwych neuronów (zwraca wartość 0 dla dowolnego wejścia, co praktycznie eliminuje go z dalszego procesu uczenia - nie jest w stanie poprawiać swoich wag w czasie trenowania, ponieważ gradient w tym obszarze wynosi 0) ponieważ zeruje ujemne wartości. Jest szczególnie popularna w warstwach ukrytych głębokich sieci neuronowych dzięki swojej prostocie i efektywności obliczeniowej.

$$\mathrm{ReLU}(x) = \max(0, x) \quad \text{dla} \quad \begin{cases} 0, & x < 0, \\ x, & x \geq 0. \end{cases}$$

Rysunek 2.6 pokazuje wykres funkcji aktywacji ReLU:



Rysunek 2.6: Wykres funkcji ReLu

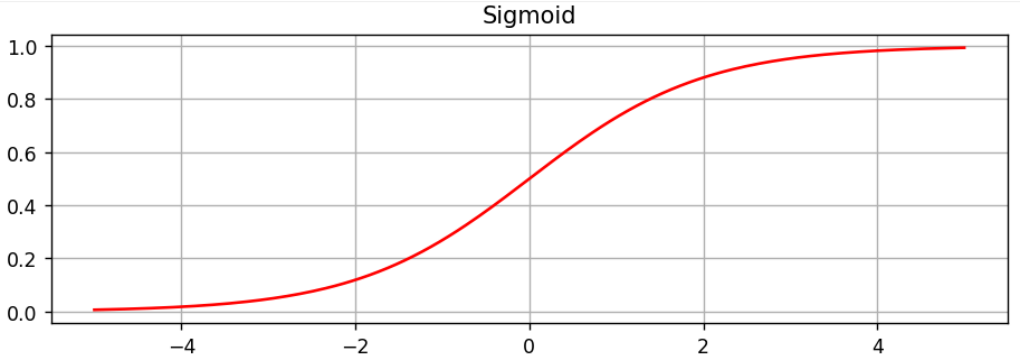
- Funkcja sigmoidalna: wyjście mieści się w przedziale od 0 do 1, co umożliwia interpretację wyników jako prawdopodobieństwo. Jest często stosowana w zadaniach binarnej klasyfikacji. Bywa problematyczna w głębokich sieciach przez zjawisko zanikania gradientu (wartości

zbliżają się do zera)

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

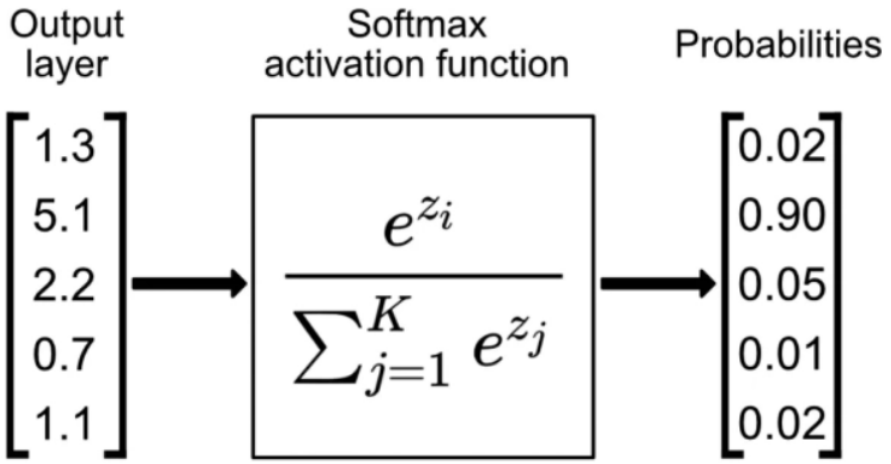
$$\lim_{x \to -\infty} \sigma(x) = 0, \quad \lim_{x \to +\infty} \sigma(x) = 1.$$

Wykres funkcji aktywacji Sigmoid przedstawiono na rysunku 2.7:



Rysunek 2.7: Wykres funkcji Sigmoidalnej

- Softmax: przekształca dane wejściowe w prawdopodobieństwa przypisane do poszczególnych klas, które sumują się do jedności. Jest powszechnie stosowana w warstwach wyjściowych przy zadaniach wieloklasowej klasyfikacji [13]. Ponieważ jest to funkcja wielowymiarowa, nie przedstawia się jej dokładnie jednym wykresem 2D, często pokazuje się jej działanie na komponentach wektora. Schemat działania funkcji Softmax ukazuje rysunek 2.8:



Rysunek 2.8: Działanie funkcji Softmax [13]

Uczenie sztucznych sieci neuronowych opiera się na minimalizacji funkcji kosztu poprzez optymalizację wag połączeń między neuronami. Proces rozpoczyna się od losowej inicjalizacji wag, po czym dane wejściowe przepływają przez warstwy sieci podczas etapu propagacji w przód. Wynik uzyskany w warstwie wyjściowej jest porównywany z rzeczywistą wartością, a różnica między nimi, zwana błędem, jest mierzona za pomocą funkcji kosztu, takiej jak np. błąd średniokwadratowy. Następnie, podczas etapu propagacji wstecznej, obliczany jest gradient funkcji kosztu, który służy do aktualizacji wag za pomocą wybranego algorytmu optymalizacji, na przykład Adam [14], lub

SGD (stochastic gradient descent). Proces ten powtarza się przez wiele epok, aż do osiągnięcia pożądanego poziomu dokładności modelu.

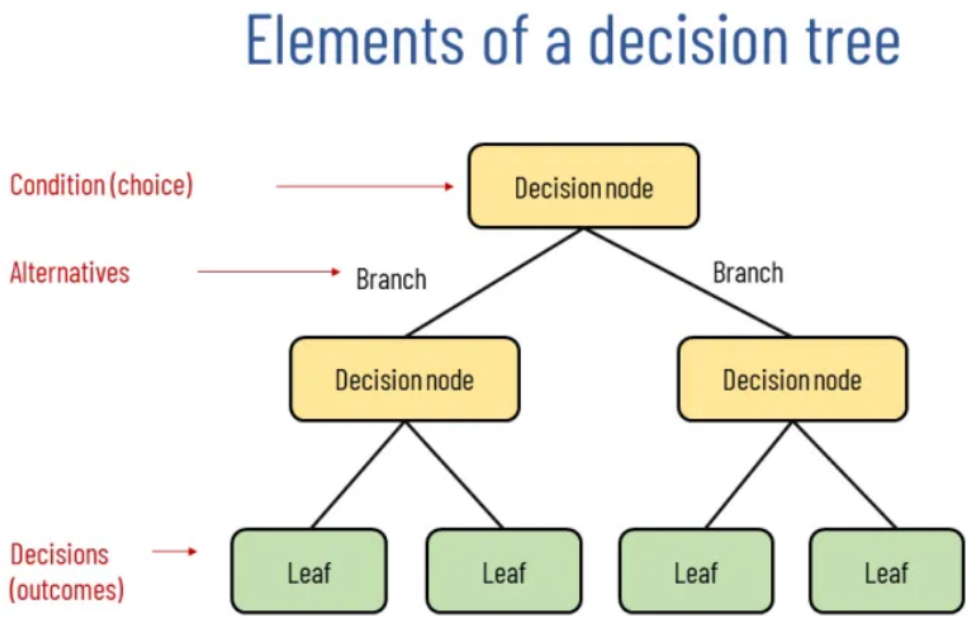
Sztuczne sieci neuronowe znajdują zastosowanie w wielu dziedzinach współczesnej technologii [15]. W rozpoznawaniu obrazów umożliwiają klasyfikację obiektów, wykrywanie twarzy oraz analizę scen. W przetwarzaniu języka naturalnego (NLP- natural language processing [16]) wspierają tłumaczenie maszynowe, analizę sentymentu oraz rozpoznawanie mowy. W systemach rekomendacyjnych pomagają w sugerowaniu produktów na podstawie preferencji użytkowników, a w autonomicznych pojazdach służą do interpretacji danych z kamer i sensorów, umożliwiając podejmowanie decyzji w czasie rzeczywistym. W dziedzinie predykcji danych sieci neuronowe wykorzystywane są do prognozowania cen akcji, analizy trendów oraz wykrywania anomalii w danych.

Rodzaje sztucznych sieci neuronowych obejmują wiele różnorodnych architektur, które zostały opracowane z myślą o rozwiązywaniu specyficznych problemów. Jednym z podstawowych typów są perceptrony wielowarstwowe (MLP- multi layer perceptron [17]), które charakteryzują się w pełni połączonymi warstwami. W tego rodzaju sieciach każdy neuron w danej warstwie jest połączony z każdym neuronem w warstwie następnej, co czyni je uniwersalnym narzędziem do rozwiązywania problemów klasyfikacyjnych i regresyjnych. MLP znajdują zastosowanie w wielu dziedzinach, takich jak rozpoznawanie wzorców czy prognozowanie, jednak ich pełne połączenia sprawiają, że są mniej efektywne w analizie danych przestrzennych, takich jak obrazy.

Oprócz wymienionych jest jeszcze więcej innych architektur z których każda charakteryzuje się unikalnymi cechami, które czynią ją odpowiednią do danych zastosowań. W trakcie tworzenia i trenowania sztucznych sieci neuronowych można natrafić na wiele problemów i wyzwań. Jednym z nich jest przeuczenie (overfitting), które sprawia, że model dobrze działa na danych treningowych, ale ma trudności z generalizacją nowych danych. Aby temu zapobiec, stosuje się techniki regularyzacji, takie jak dropout (losowe wyłączanie neuronów które nie uczestniczą potem w propagacji wstecznej) czy L2 regularization (dodawanie kar w postaci sumy kwadratów wag -norma L2- do funkcji kosztu modelu. Dzięki temu model jest zachęcany do utrzymywania mniejszych wartości wag). Innym wyzwaniem jest wybór odpowiednich hiperparametrów, takich jak liczba warstw, liczba neuronów w warstwie czy wartości parametrów optymalizacji, które mają kluczowy wpływ na jakość modelu. Dodatkowo, efektywne uczenie sieci wymaga dużych zbiorów danych oznaczonych, co może być kosztowne i czasochłonne w przygotowaniu. W praktyce istotne jest również odpowiednie skalowanie sieci i dostosowanie jej do dostępnych zasobów obliczeniowych.

2.5 Drzewa decyzyjne i regresyjne

Drzewa decyzyjne to wszechstronne narzędzia wykorzystywane w klasyfikacji i regresji. Ich struktura przypomina drzewo binarne, gdzie węzły wewnętrzne reprezentują kryteria podziału danych na podstawie wybranych cech, a węzły końcowe (liście) przedstawiają wyniki modelu (etykiety klas w klasyfikacji lub wartości ciągłe w regresji) [18]. Schemat przedstawiono na rysunku 2.9.



Rysunek 2.9: Schemat budowy drzewa decyzyjnego [18]

Dzięki prostocie interpretacji i intuicyjności drzewa decyzyjne są jednymi z najczęściej stosowanych algorytmów uczenia maszynowego. Stanowią także podstawę bardziej zaawansowanych metod, takich jak Random Forest (działanie wielu drzew decyzyjnych na raz) czy Gradient Boosting (łączenie wielu słabych modeli w celu stworzenia silnego).

Drzewa decyzyjne tworzy się w procesie rekurencyjnego podziału przestrzeni cech wejściowych. Celem tego podziału jest maksymalizacja czystości danych w każdym z powstałych węzłów. Pierwszym krokiem w algorytmie budowy drzewa jest wybór cechy podziału. W każdym węźle drzewa wybierana jest cecha która maksymalizuje poprawę podziału. Kryteria wyboru cechy zależą od zadania:

- W przypadku klasyfikacji wykorzystywana jest między innymi entropia [19]:

$$H(S) = -\sum_{i=1}^{k} p_i \log_2(p_i)$$

  gdzie p_i to prawdopodobieństwo wystąpienia klasy i. Niższa entropia oznacza większą czystość węzła.

- Drugą miarą jest indeks Gini [20]:

$$G(S) = 1 - \sum_{i=1}^{k} p_i^2$$

  który wynosi 0 dla idealnie czystych węzłów i wzrasta, gdy klasy są bardziej wymieszane.

- Dla regresji kryterium podziału opiera się na minimalizacji średniego błędu kwadratowego:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

  gdzie y_i to rzeczywiste wartości, a $\hat{y}_i$ to przewidywania modelu.

Kolejnym krokiem jest podział danych w węźle- wybrana cecha i próg podziału dzielą dane na dwie grupy: lewą gdzie zostają przydzielone próbki spełniające warunek podziału, oraz prawą do której zostają dodane próbki niespełniające warunku. Następnie proces podziału jest kontynuowany rekurencyjnie, aż zostanie osiągnięte jedno z kryteriów zakończenia:

- Maksymalna głębokość drzewa.
- Minimalna liczba próbek w liściu.
- Brak dalszej poprawy jakości podziału.

Drzewa regresyjne służą do przewidywania wartości ciągłych. Węzły końcowe zawierają średnią wartość zmiennej zależnej dla próbek przypisanych do danego liścia. Dane są dzielone na regiony, które modelują zarówno liniowe, jak i nieliniowe zależności. Dzięki rekursywnym podziałom drzewa regresyjne są zdolne do odwzorowania złożonych nieliniowych relacji. Dużymi zaletami drzew decyzyjnych i regresyjnych są:

- Intuicyjność i interpretowalność- struktura drzewa jest łatwa do zrozumienia, nawet dla osób nietechnicznych.
- Wszechstronność - obsługują dane jakościowe i ilościowe.
- Brak konieczności skalowania danych - działają niezależnie od skali cech wejściowych.

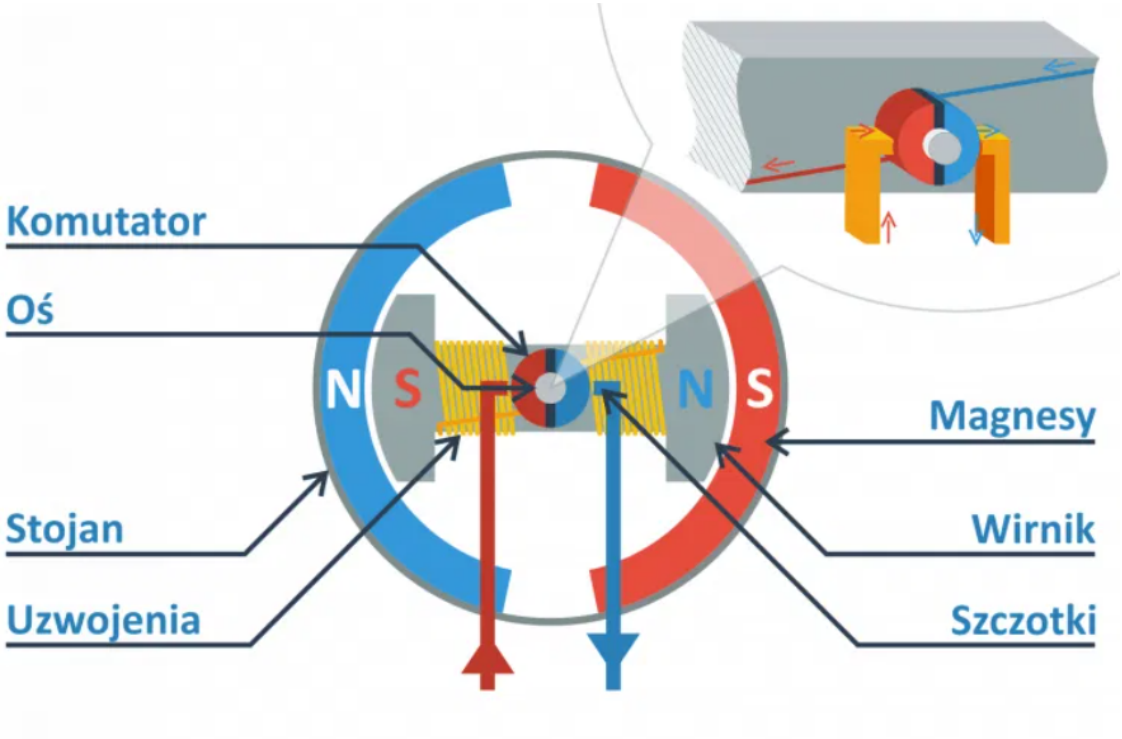
Wady natomiast są:

- Nadmierne dopasowanie (overfitting) - głębokie drzewa mogą dopasowywać się do szumu w danych, co skutkuje słabą generalizacją.
- Wrażliwość na dane wejściowe - małe zmiany w danych mogą znacznie zmienić strukturę drzewa.
- Ograniczona dokładność - w porównaniu z bardziej złożonymi modelami, takimi jak Random Forest czy Gradient Boosting, pojedyncze drzewa często mają niższą wydajność.

2.6 Silniki prądu stałego (DC)

Silniki prądu stałego stanowią jedne z najbardziej uniwersalnych i szeroko stosowanych maszyn elektrycznych w różnych gałęziach przemysłu. Ich popularność wynika z prostej budowy, precyzji w sterowaniu oraz szerokiego zakresu zastosowań obejmujących robotykę, transport, przemysł produkcyjny i wiele innych dziedzin. Kluczowym elementem pracy z silnikami DC jest ich odpowiednie sterowanie i regulacja, co umożliwia dostosowanie parametrów pracy do wymagań danej aplikacji.

Silniki prądu stałego są maszynami, których działanie opiera się na zjawisku wzajemnego oddziaływania prądu elektrycznego płynącego przez uzwojenia z polem magnetycznym wytwarzanym przez stojan. Podstawowe elementy konstrukcyjne silnika DC to wirnik, stojan, komutator i szczotki (Istnieją również bezszczotkowe silniki DC np. BLDC, są to jednak rozwiązania bardziej zaawansowane technologicznie.). Schemat budowy silnika szczotkowego DC przedstawiono na rysunku 2.10. Wirnik, czyli obracająca się część maszyny, jest zasilany za pośrednictwem komutatora, który odpowiada za zmianę kierunku prądu w uzwojeniach. Pole magnetyczne stojana, generowane przez magnesy trwałe lub uzwojenia elektromagnetyczne, oddziałuje na przewody wirnika, wywołując siłę elektromotoryczną zgodnie z prawem Ampère'a i Lorentz'a. W wyniku tego oddziaływania powstaje moment obrotowy wprawiający wirnik w ruch [21].



Rysunek 2.10: Schemat budowy silnika szczotkowego DC [21]

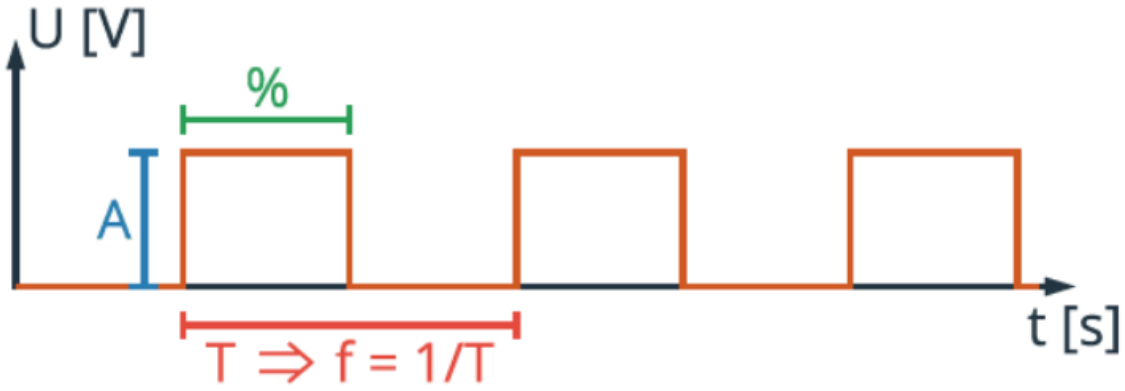
2.6.1 Metody sterowania

Sterowanie silnikami DC może być realizowane przy użyciu kilku różnych technik, z których każda odpowiada na konkretne potrzeby aplikacyjne oraz charakterystykę układu. Najprostsze rozwiązania bazują na zmianie napięcia zasilania, podczas gdy bardziej zaawansowane metody obejmują techniki oparte na modulacji szerokości impulsów oraz sterowaniu wektorowym.

Sterowanie napięciem zasilania polega na bezpośredniej zmianie wartości napięcia dostarczanego do silnika. Wzrost napięcia skutkuje zwiększeniem prędkości obrotowej, natomiast jego obniżenie powoduje spadek prędkości. Choć metoda ta jest prosta i intuicyjna, jej skuteczność ogranicza

się do układów o niewielkich wymaganiach dynamicznych, gdzie nie występują szybkie zmiany obciążenia.

PWM, czyli modulacja szerokości impulsów, oferuje bardziej zaawansowaną kontrolę poprzez regulację średniego napięcia dostarczanego do silnika. W tej metodzie sygnał zasilający ma charakter impulsowy, a jego współczynnik wypełnienia decyduje o wartości średniej napięcia. PWM pozwala na płynne i precyzyjne sterowanie prędkością obrotową. Przykładowy przebieg sygnału przedstawiono na rysunku 2.11:



Rysunek 2.11: Przykładowy sygnał PWM [21]

Wyjaśnienie symboli przedstawionych na zdjęciu:

- A – Amplituda: największa osiągana wartość sygnału.
- T – Okres sygnału: czas pomiędzy kolejnymi powtórzeniami cyklu sygnału.
- f – Częstotliwość: liczba cykli sygnału, które zachodzą w ciągu jednej sekundy.
- % – Współczynnik wypełnienia: proporcja czasu, w którym sygnał znajduje się w stanie wysokim, względem całego okresu.

Sterowanie adaptacyjne i oparte na sztucznej inteligencji reprezentuje nowoczesne podejście do zarządzania pracą silników prądu stałego, wykorzystując zaawansowane algorytmy i techniki uczenia maszynowego. W tego typu rozwiązaniach stosuje się zarówno klasyczne metody, takie jak adaptacyjne regulatory PID, jak i bardziej zaawansowane podejścia, w tym algorytmy optymalizacyjne, systemy rozmyte, algorytmy ewolucyjne czy sztuczne sieci neuronowe.

2.6.2 Regulacja prędkości i momentu

W celu utrzymania zadanej prędkości obrotowej lub momentu obrotowego stosuje się odpowiednie regulatory, które korygują parametry pracy silnika w odpowiedzi na zmienne warunki zewnętrzne. Najbardziej rozpowszechnionymi rozwiązaniami są:

- Regulator PID - pozwala na precyzyjne dostosowanie sygnału sterującego dzięki składowym proporcjonalnej, całkującej i różniczkującej. Jest to klasyczne rozwiązanie zapewniające stabilność i dokładność układu regulacji.
- Regulatory adaptacyjne - rozszerzają możliwości PID poprzez dynamiczne dostosowanie parametrów w czasie rzeczywistym do zmieniających się warunków pracy. Dzięki temu układ potrafi lepiej reagować na zmiany obciążenia oraz charakterystyki silnika.

- Metody predykcyjne - opierają się na modelach matematycznych układu, umożliwiając przewidywanie przyszłego zachowania silnika. W praktyce pozwala to na odpowiednio wczesne dostosowanie sygnałów sterujących, co skutkuje minimalizacją zakłóceń i optymalizacją pracy silnika.

W zaawansowanych systemach metody te mogą być łączone z algorytmami opartymi na sztucznej inteligencji, co pozwala osiągnąć maksymalną efektywność oraz niezawodność pracy układu.

2.7 Przegląd istniejących rozwiązań

W obszarze sterowania silnikami prądu stałego i regulatorami PID przy pomocy sztucznej inteligencji istnieje wiele przykładów zastosowania zaawansowanych algorytmów, które znacząco poprawiają efektywność oraz adaptacyjność systemów sterowania. Jednym z popularnych rozwiązań jest wykorzystanie algorytmów genetycznych do optymalizacji parametrów regulatora PID. Algorytmy te, bazujące na mechanizmach ewolucyjnych, pozwalają na dynamiczne dostosowywanie parametrów P, I i D w czasie rzeczywistym. Przykładem może być adaptacyjny regulator PID, który wykorzystuje algorytm genetyczny do optymalizacji parametrów w systemach sterowania silnikami DC. Tego typu rozwiązanie jest szczególnie przydatne w systemach, gdzie warunki pracy ulegają częstym zmianom, na przykład w przypadku zmiennego obciążenia lub fluktuacji napięcia zasilania. Przykładowa publikacja [22], opisuje dokładnie, jak algorytmy genetyczne mogą być zaimplementowane w praktycznych zastosowaniach, co pokazuje ich efektywność w poprawie dynamicznej reakcji układów sterowania.

Kolejnym popularnym podejściem jest zastosowanie sztucznych sieci neuronowych do sterowania regulatorami PID. Sieci neuronowe uczą się zależności między zmiennymi systemowymi na podstawie danych wejściowych. W sterowaniu silnikami DC sieci neuronowe mogą dynamicznie dostosowywać parametry regulatora PID, ucząc się wzorców w systemie, takich jak zmieniające się obciążenie czy warunki pracy. Przykład takiego rozwiązania można znaleźć w innej pracy [23], która szczegółowo opisuje, jak sieć neuronowa może być zaimplementowana do optymalizacji sterowania silnikiem. W tego typu systemie sieci neuronowe mogą zastąpić tradycyjne metody doboru parametrów, co eliminuje potrzebę ręcznej konfiguracji i pozwala na szybsze reagowanie na zmiany w środowisku pracy.

Oprócz algorytmów genetycznych i sieci neuronowych, innym ciekawym rozwiązaniem jest wykorzystanie logiki rozmytej w sterowaniu silnikami DC. Logika rozmyta pozwala na adaptacyjne sterowanie systemem, opierając się na regułach opisujących zachowanie systemu w różnych warunkach. Przykład takiego zastosowania można znaleźć w artykule [24] opublikowanym w czasopiśmie Elsevier. Systemy bazujące na logice rozmytej są szczególnie przydatne w sytuacjach, gdzie istnieje wysoki poziom niepewności lub zmienność parametrów, na przykład w przemyśle.

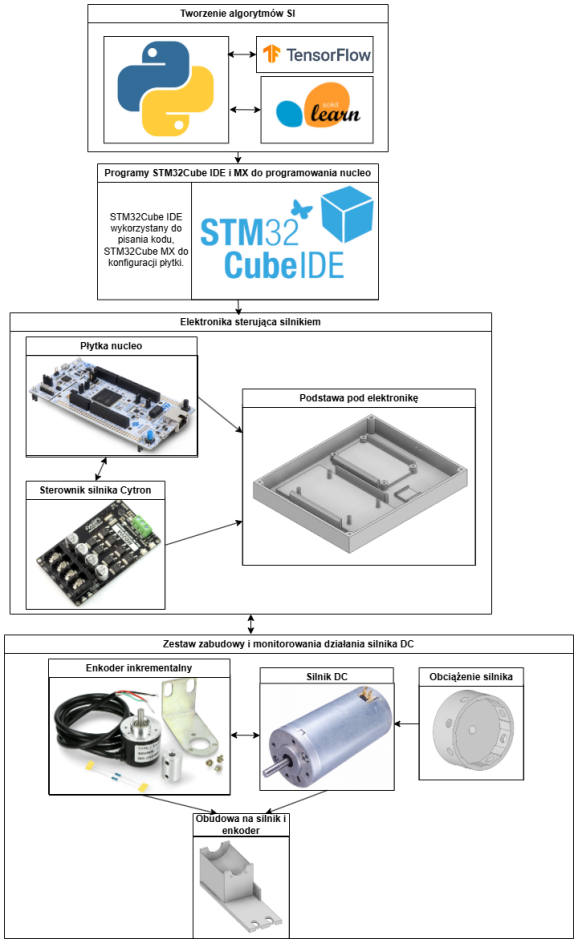
Rozdział 3

Stanowisko pomiarowe

3.1 Elementy do stanowiska pomiarowego

W ramach pracy zbadano możliwość zastosowania algorytmów sztucznej inteligencji w sterowaniu prędkością obrotową silnika DC dla różnych obciążeń jak i wykorzystania tych metod do strojenia regulatora PID. W celu sprawdzenia proponowanych rozwiązań zbudowano stanowisko badawcze.

W skład stanowiska wchodzą: laptop lub komputer stacjonarny, płytka nucleoF439ZI, sterownik silnika Cytron MD20A, zasilacz, ekoder obrotowy SEN0230, sprzęgło i silnik DC. Układ stanowiska zaprezentowano na rysunku 3.1.



Rysunek 3.1: Schemat stanowiska sterowania silnikiem DC przy zastosowaniu algorytmów sztucznej inteligencji

3.2 Projekt obudowy

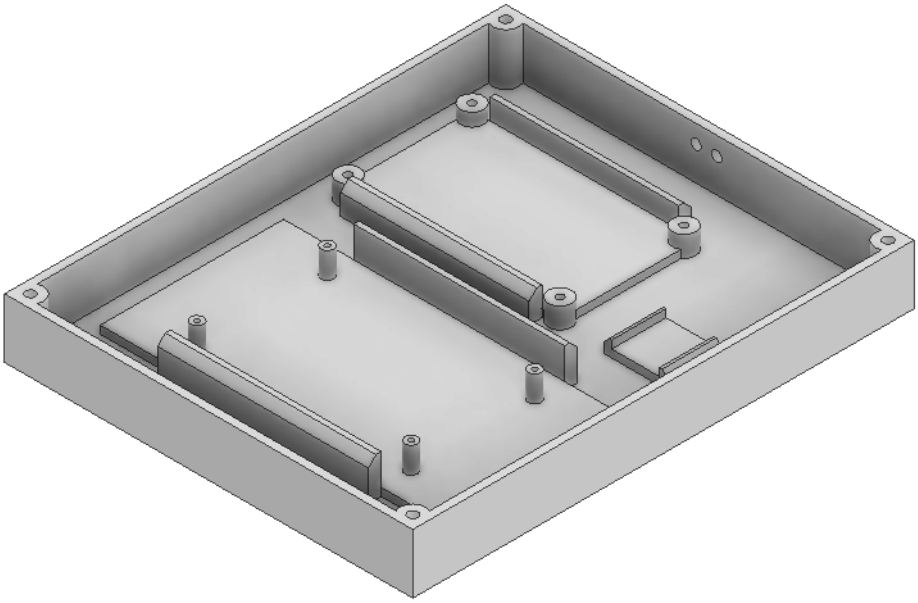
W ramach projektu zaplanowano wykonanie obudowy w technologii druku 3D. Ze względu na ograniczony obszar roboczy dostępnej drukarki 3D, obudowę podzielono na trzy mniejsze części:

- Podstawa główna na której zamontowano płytkę Nucleo, sterownik silnika oraz czujnik prądu.
- Pokrywa podstawy do zabezpieczenia elektroniki.
- Osobna podstawa do której przymocowano silnik i enkoder.

Dodatkowo zamodelowano zmienne obciążenie na wał silnika

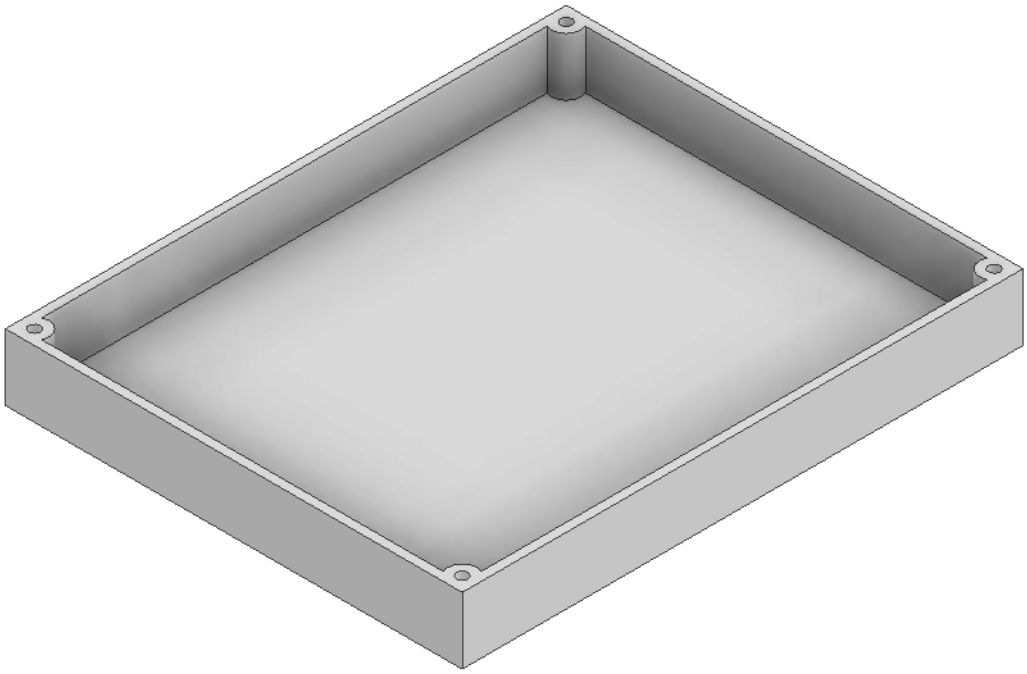
Aby zapewnić odpowiednią sztywność konstrukcji, obie podstawy można łączyć za pomocą śrub.

Wszystkie części zaprojektowano w programie Autodesk Inventor 2025, a następnie wyeksportowano w formacie STL do oprogramowania typu slicer. Oprogramowanie to przekształciło modele STL na pliki w formacie G-code, gotowe do przeprowadzenia procesu wydruku.



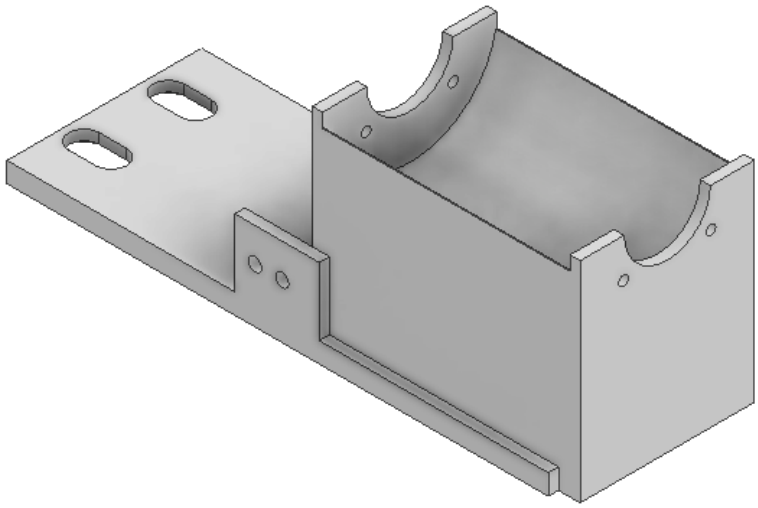
Rysunek 3.2: Model podstawki do elektroniki

Podstawka została zaprojektowana z myślą o umieszczeniu na niej płytki nucelo i sterownika silnika Cytron. Przewiduje jeszcze miejsce na mały czujnik prądu który końcowo nie został wykorzystany w stanowisku



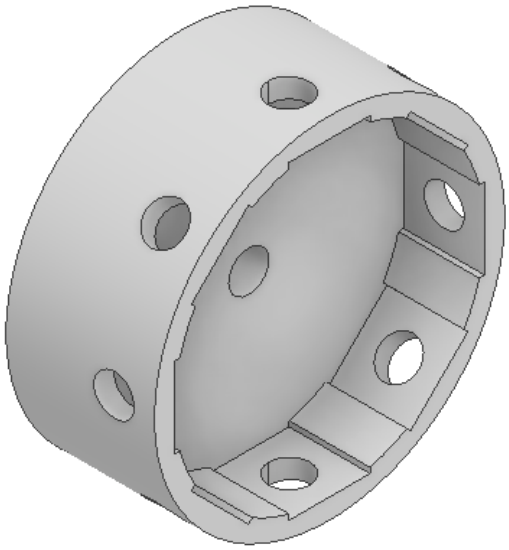
Rysunek 3.3: Model pokrywy do elektroniki

Model pokrywy posiada 4 otwory przeznaczone na śruby scalające go z podstawką.



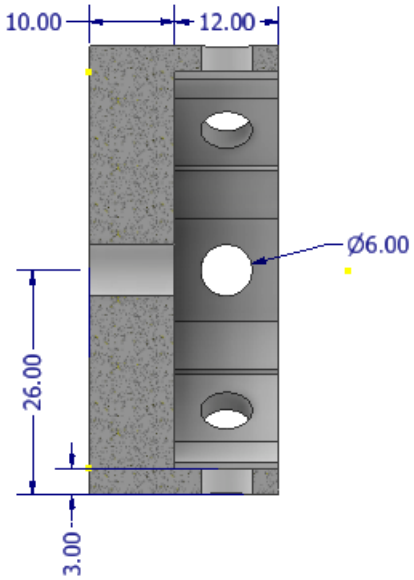
Rysunek 3.4: Podstawa pod silnik i enkoder

W modelu podstawy pod silnik i enkoder zastosowano ciasne pasowanie silniki w podstawie i dodatkową opcję połączenia go z wydrukiem za pomocą 4 śrub M3. Otwory na stelaż enkodera przewidują dodanie poszerzanych podkładek od spodu. Cała podstawa może być złączona z podstawą pod elektronikę za pomocą 2 śrub M4 co zapewni całości konstrukcji stabilność.



Rysunek 3.5: Model obciążenia silnika

Zaprojektowany model obciążenia silnika na wale, posiada 8 otworów na śruby M6.



Rysunek 3.6: Wymiary obciążnika

Moment bezwładności elementu wyliczono przez podzielenie go na dwa mniejsze i uproszczenie struktury.

3.3 Obliczanie momentu bezwładności obciążenia

Część 1: „Krążek” ($R = 26\,\mathbf{mm}$, $r = 3\,\mathbf{mm}$, $H = 10\,\mathbf{mm}$)

1.1. Objętość Objętość pierścienia walcowego:

$$V_A = \pi \left(R^2 - r^2\right) H.$$

Dla $R = 26\,\text{mm}$, $r = 3\,\text{mm}$, $H = 10\,\text{mm}$:

$$R^2 = 26^2 = 676, \quad r^2 = 3^2 = 9.$$

$$R^2 - r^2 = 676 - 9 = 667.$$

W mm^3:

$$V_A = \pi \times 667 \times 10 = 6670\pi\,\text{mm}^3 \approx 20963.9\,\text{mm}^3.$$

Aby uzyskać cm^3, dokonywany jest podział przez 1000:

$$V_A \approx 20.964\,\text{cm}^3.$$

1.2. Masa krążka Dla PLA o $\rho \approx 1.24\,\text{g/cm}^3$:

$$m_A = \rho \cdot V_A \approx 1.24 \times 20.964\,\text{g} \approx 26.0\,\text{g} = 0.026\,\text{kg}.$$

1.3. Moment bezwładności krążka Oś obrotu przechodzi przez środek walca, wzdłuż wysokości. Dla jednorodnego pierścienia moment bezwładności względem tej osi wynosi:

$$I_A = \frac{1}{2} m_A \left(R^2 + r^2\right),$$

gdzie R i r trzeba wstawić w metrach, a m_A w kilogramach:

$$R = 0.026\,\text{m}, \quad R^2 = 0.000676,$$

$$r = 0.003\,\text{m}, \quad r^2 = 0.000009,$$

$$R^2 + r^2 = 0.000685, \quad m_A = 0.026\,\text{kg}.$$

Ostatecznie:

$$I_A = \frac{1}{2} \times 0.026 \times 0.000685 = 0.013 \times 0.000685 \approx 8.9 \times 10^{-6}\,\text{kg}\,\text{m}^2.$$

Część 2: „Kołnierz” ($R = 26\,\text{mm}$, $r = 23\,\text{mm}$, $H = 12\,\text{mm}$)

To pierścień/tuleja o wysokości 12 mm i grubości 3 mm (od 23 mm do 26 mm promienia).

2.1. Objętość Objętość pierścienia walcowego:

$$V_B = \pi \left(R^2 - r^2\right) H.$$

Dla $R = 26\,\text{mm} \implies R^2 = 676$, $r = 23\,\text{mm} \implies r^2 = 529$, $R^2 - r^2 = 147$, $H = 12\,\text{mm}$:

$$V_B = \pi \times 147 \times 12 = 1764\pi\,\text{mm}^3 \approx 5542\,\text{mm}^3.$$

Aby uzyskać cm^3:

$$V_B \approx \frac{5542}{1000} = 5.542\,\text{cm}^3.$$

2.2. Masa kołnierza Dla $\rho \approx 1.24\,\mathrm{g/cm^3}$:

$$m_B = \rho \cdot V_B \approx 1.24 \times 5.542\,\mathrm{g} \approx 6.88\,\mathrm{g} = 0.00688\,\mathrm{kg}.$$

2.3. Moment bezwładności kołnierza

$$I_B = \frac{1}{2} m_B \left(R^2 + r^2\right).$$

Dla $R = 0.026\,\mathrm{m} \implies R^2 = 0.000676$, $r = 0.023\,\mathrm{m} \implies r^2 = 0.000529$, $R^2 + r^2 = 0.001205$, $m_B = 0.00688\,\mathrm{kg}$:

$$I_B = \frac{1}{2} \times 0.00688 \times 0.001205 = 0.00344 \times 0.001205 \approx 4.15 \times 10^{-6}\,\mathrm{kg\,m^2}.$$

3. Całe urządzenie: masa i moment bezwładności

Zakładamy, że krążek (A) i kołnierz (B) są sztywno połączone wzdłuż tej samej osi. Wtedy całkowity moment bezwładności to suma momentów (bo mają ten sam środek i oś).

3.1. Całkowita masa

$$m_{\mathrm{total}} = m_A + m_B = 0.026 + 0.00688 = 0.03288\,\mathrm{kg} \approx 32.9\,\mathrm{g}.$$

3.2. Całkowity moment bezwładności

$$I_{\mathrm{total}} = I_A + I_B = 8.9 \times 10^{-6} + 4.15 \times 10^{-6} = 1.305 \times 10^{-5}\,\mathrm{kg\,m^2} \approx 1.31 \times 10^{-5}\,\mathrm{kg\,m^2}.$$

4. Dodatkowe obciążenie Do każdego otworu mocowana będzie śruba M6 z 25mm gwintem, 6 podkładek poszerzanych i nakrętka.

4.1. Orientacyjna masa poszczególnych elementów

- Nakrętka M6: $\sim 2\,\mathrm{g}$,
- 6 podkładek M6 powiększanych około $1.5\,\mathrm{g}$ na sztukę:

$$6 \times 1.5\,\mathrm{g} = 9\,\mathrm{g},$$

- Śruba: $\sim (6 - 7)\,\mathrm{g}$ – w zależności od dokładnej klasy wytrzymałości itp.

5. Obliczenie momentu bezwładności jednej śruby (2-segmentowy model)

Zastosowano model mas punktowych (lumped mass). Podzielono masę na 2–3 charakterystyczne fragmenty, przypisując każdemu promień r (od osi) i sumujemy mr^2.

5.1. Nakrętka Masa:

$$m_{\mathrm{nut}} = 0.002\,\mathrm{kg},$$

Promień:

$$r_{\mathrm{nut}} = 0.023\,\mathrm{m}.$$

Moment bezwładności:

$$I_{\mathrm{nut}} = m_{\mathrm{nut}}\, r_{\mathrm{nut}}^2 = 0.002 \times (0.023)^2 = 0.002 \times 0.000529 \approx 1.058 \times 10^{-6} \approx 1.06 \times 10^{-6}\,\mathrm{kg\,m^2}.$$

5.2. Łeb śruby + 6 podkładek + pozostały gwint Masa łącznie:

$$m_{\text{ext}} = 0.017 - 0.002 = 0.015\,\text{kg},$$

Promień:

$$r_{\text{ext}} = 0.026\,\text{m}.$$

Moment bezwładności:

$$I_{\text{ext}} = m_{\text{ext}}\, r_{\text{ext}}^2 = 0.015 \times (0.026)^2 = 0.015 \times 0.000676 = 1.014 \times 10^{-5} \approx 1.01 \times 10^{-5}\,\text{kg}\,\text{m}^2.$$

5.3. Suma dla jednej śruby

$$I_{\text{1screw}} = I_{\text{nut}} + I_{\text{ext}} \approx 1.06 \times 10^{-6} + 1.01 \times 10^{-5} = 1.116 \times 10^{-5} \approx 1.12 \times 10^{-5}\,\text{kg}\,\text{m}^2.$$

6. Osiem śrub (rozmieszczonych równomiernie)

Moment bezwładności dla ośmiu śrub:

$$I_{\text{8screws}} = 8 \times I_{\text{1screw}} = 8 \times 1.12 \times 10^{-5} = 8.96 \times 10^{-5} \approx 9.0 \times 10^{-5}\,\text{kg}\,\text{m}^2.$$

7. Dodanie do momentu bezwładności korpusu wykonanego z PLA

Z wcześniejszych wyliczeń:

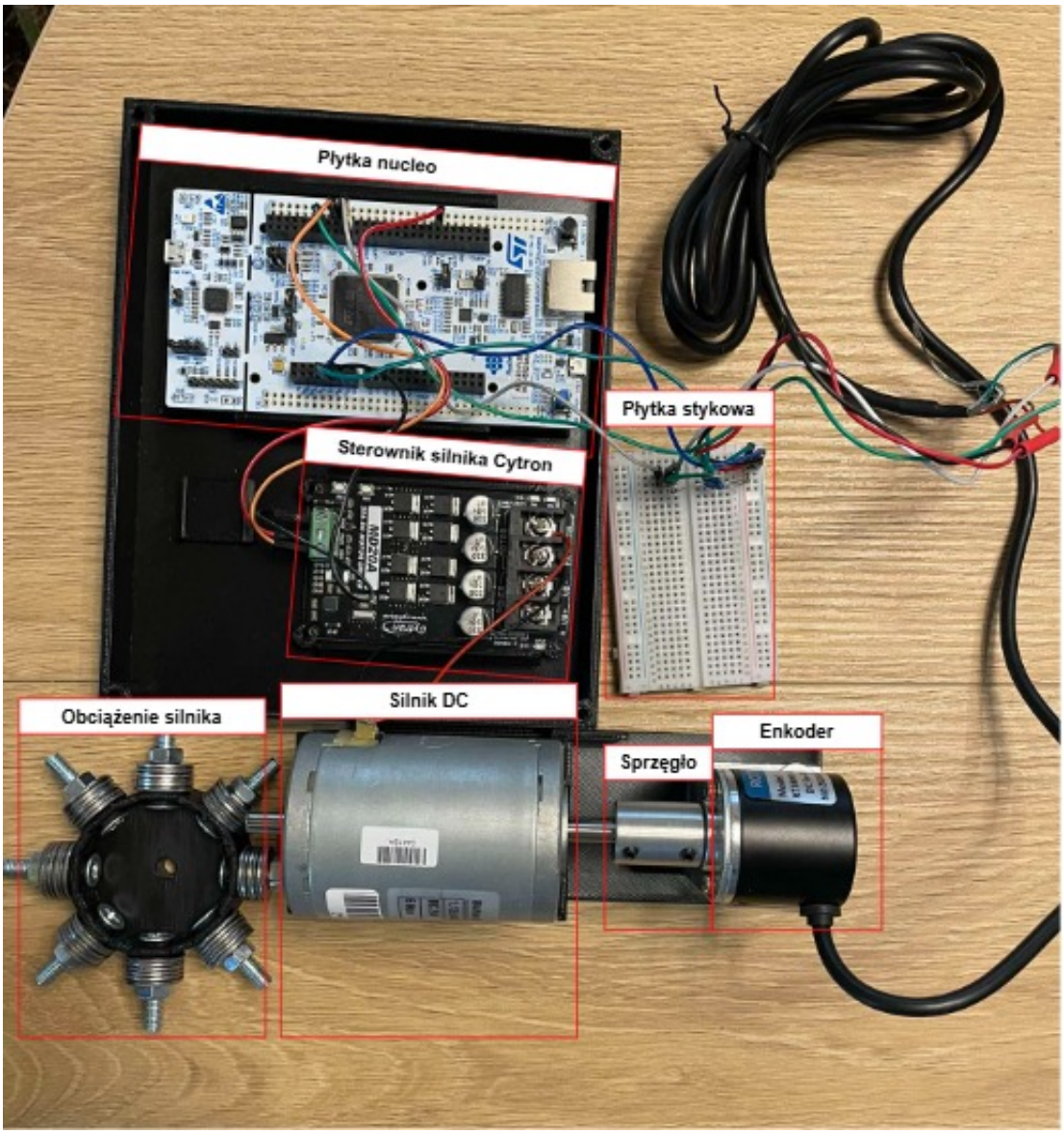
$$I_{\text{korpus}} \approx 1.3 \times 10^{-5}\,\text{kg}\,\text{m}^2.$$

Całkowity moment bezwładności:

$$I_{\text{total}} = I_{\text{korpus}} + I_{\text{8screws}} = 1.3 \times 10^{-5} + 9.0 \times 10^{-5} = 1.03 \times 10^{-4} \approx 1.0 \times 10^{-4}\,\text{kg}\,\text{m}^2.$$

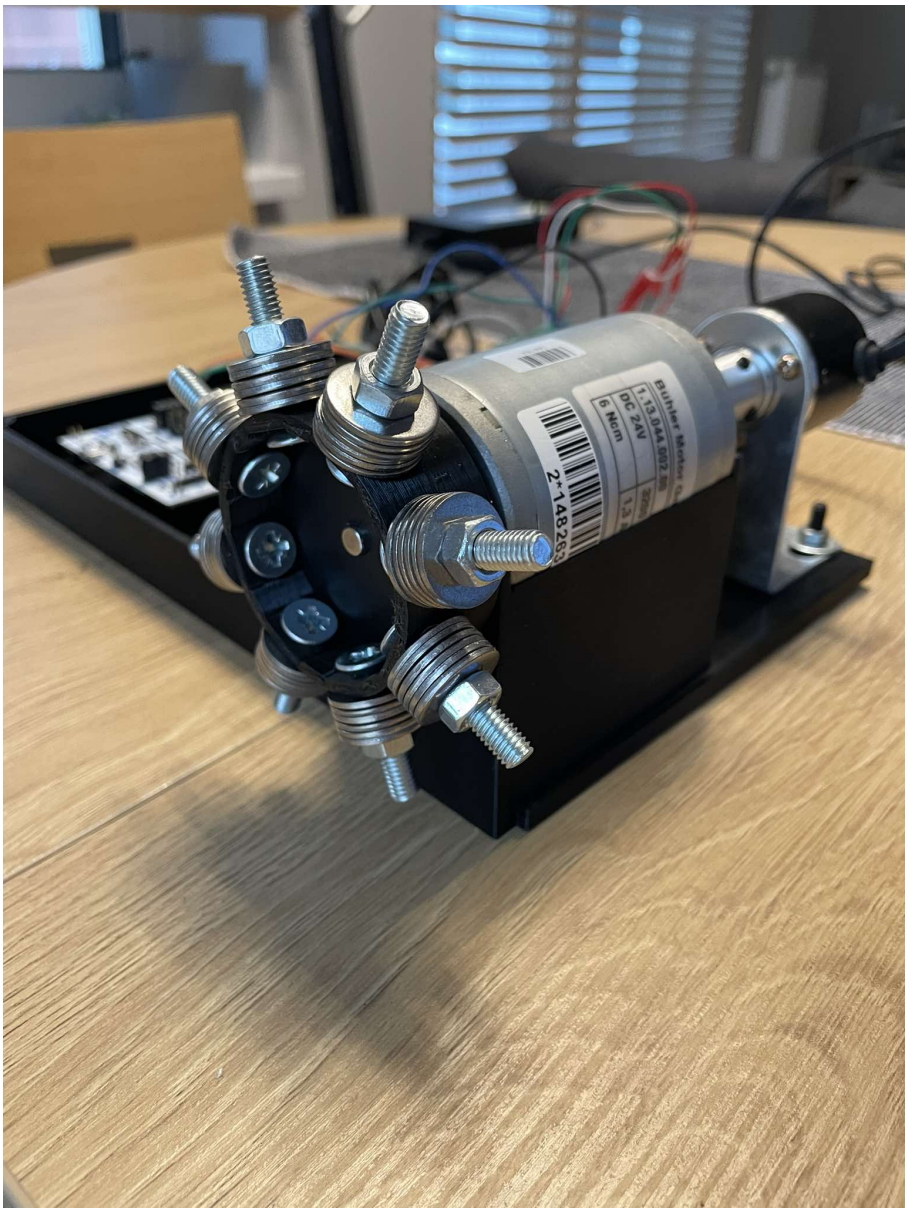
3.4 Złożenie stanowiska

Elektronika została ustawiona na głównej podstawce. Istnieje możliwość zamontowania wykorzystywanych płytek przy pomocy śrub. Silnik wraz z enkoderem zamontowano na bocznej podstawce. Połączenie silnika z enkoderem zrealizowano za pomocą sztywnego sprzęgła zaciskającego się na wale silnika i wyżłobieniu w wale enkodera. Przygotowane moduły stanowiska zaprezentowano na zdjęciu 3.7.



Rysunek 3.7: Zdjęcie z góry prezentujące elementy stanowiska

W stanowisku oprócz wymienianych wcześniej elementów zastosowano dodatkowo płytkę stykową. Pozwalała ona na szybkie dokonywanie zmian w połączeniach i diagnozę usterek powodujących nieprawidłowe działanie układu. Obciążenie silnika jest realizowane poprzez montaż zestawu śrub i podkładek na wydrukowany model przestosowany do mocowania na wale silnika. Zamontowane obciążenie pokazano na zdjęciu 3.8.



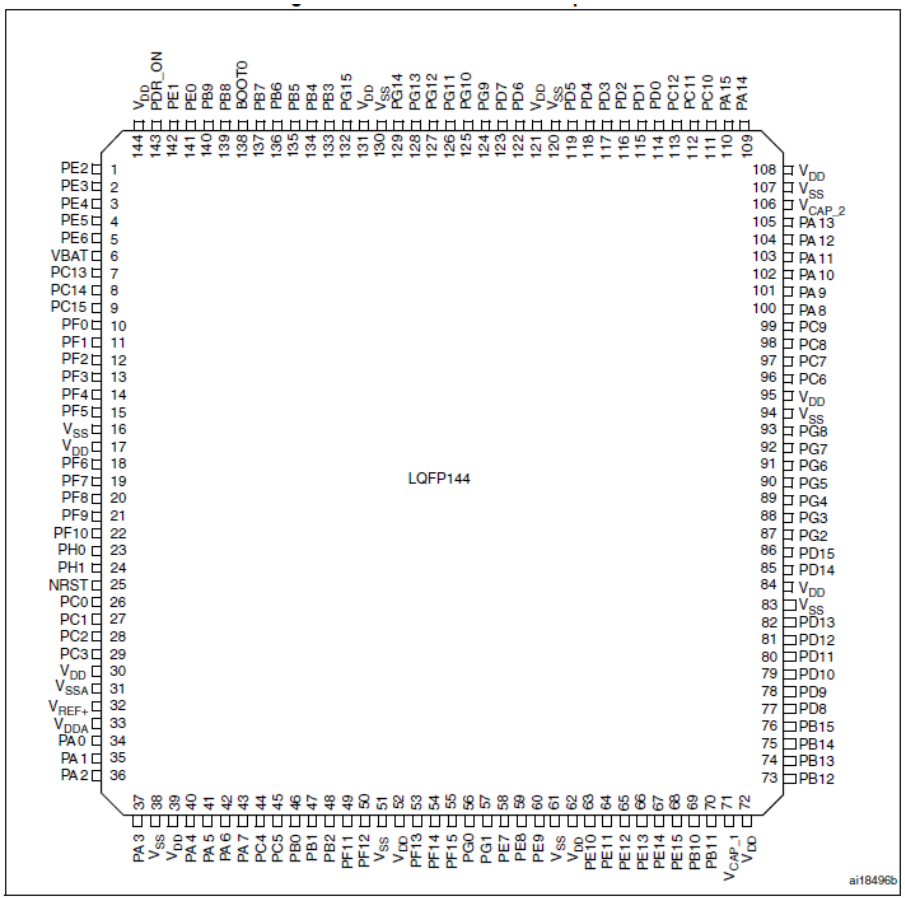
Rysunek 3.8: Zdjęcie prezentujące zastosowane obciążenie silnika

Ostatecznie zmieniono układ montowanego dodatkowego obciążenia - główki śrub znajdują się wewnątrz kołnierza a nakrętki na zewnątrz. Powoduje to jedynie marginalną zmianę całkowitego momentu bezwładności.

Rozdział 4

Program do obsługi płytki nucelo

Do konfiguracji mikroprocesora i napisania kodu sterującego działaniem silnika DC wykorzystano programi STM32Cube IDE i STM32Cube MX. MX służy do konfiguracji ustawień mikroprocesora i pinoutu. Istnieje możliwość zaimportowania ustawień z ogólnie dostępnych modeli płytek co znacznie przyśpiesza pracę. W konfigurcji użyto modelu płytki wykorzystywanej w projekcie. Ekran startowy MX wyświetla model mikroprocesora wraz z dostępnymi wyjściami; widok ten jest odbiciem danych w nocie katalogowej. Fragment noty przedstawiono na rysunku 4.1.

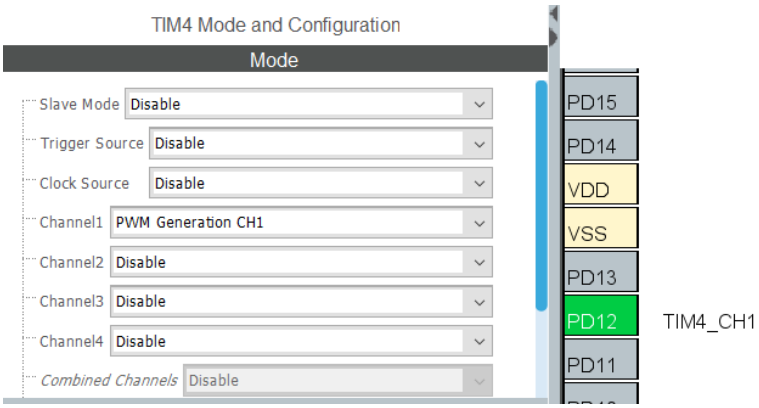


Rysunek 4.1: Strona z noty katalogowej prezentująca układ pinów wyjsciowych [25]

Po wykonaniu odpowiedniej konfiguracji, MX generuje wstępny kod z inicjalizacją wszystkich wykorzystywanych peryferiów do dalszej obróki w IDE.

4.1 Generowanie PWM

Do sterowania silnikiem DC wybrano metodę opartą na generowaniu odpowiedniego sygnału PWM.



Rysunek 4.2: Skonfigurowanie TIM4 do generowania sygnału PWM

W tym widoku w CubeMX ustala się tryb pracy TIM4. Na kanale 1 (Channel1) jest włączona funkcja PWM Generation. Dzięki temu, TIM4 wygeneruje sygnał PWM na wyjściu fizycznego pinu przypisanego do TIM4_CH1 (w tym wypadku PD12).

```
3404 static void MX_TIM4_Init(void) {
3405
3406     /* USER CODE BEGIN TIM4_Init 0 */
3407
3408     /* USER CODE END TIM4_Init 0 */
3409
3410     TIM_MasterConfigTypeDef sMasterConfig = { 0 };
3411     TIM_OC_InitTypeDef sConfigOC = { 0 };
3412
3413     /* USER CODE BEGIN TIM4_Init 1 */
3414
3415     /* USER CODE END TIM4_Init 1 */
3416     htim4.Instance = TIM4;
3417     htim4.Init.Prescaler = 0;
3418     htim4.Init.CounterMode = TIM_COUNTERMODE_UP;
3419     htim4.Init.Period = 8399;
3420     htim4.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
3421     htim4.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
3422     if (HAL_TIM_PWM_Init(&htim4) != HAL_OK) {
3423         Error_Handler();
3424     }
3425     sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
3426     sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
3427     if (HAL_TIMEx_MasterConfigSynchronization(&htim4, &sMasterConfig)
3428             != HAL_OK) {
3429         Error_Handler();
3430     }
3431     sConfigOC.OCMode = TIM_OCMODE_PWM1;
3432     sConfigOC.Pulse = 4199;
3433     sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
3434     sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
3435     if (HAL_TIM_PWM_ConfigChannel(&htim4, &sConfigOC, TIM_CHANNEL_1)
3436             != HAL_OK) {
3437         Error_Handler();
3438     }
3439     /* USER CODE BEGIN TIM4_Init 2 */
3440
3441     /* USER CODE END TIM4_Init 2 */
3442     HAL_TIM_MspPostInit(&htim4);
3443
3444 }
```

Rysunek 4.3: Ustawienia TIM4

Po skonfigurowaniu widoku w MX, wygenerowano kod ustawiający działanie `TIM4`. Najważ-

niejsze elementy tego kodu to:

- `htim4.Init.Period = 8399;` - maksymalna wartość licznika wynosi 8399. Gdy licznik osiąga tę wartość, zostaje wyzerowany i zaczyna liczyć od nowa.
- `sConfigOC.OCMode = TIM_OCMODE_PWM1;` - konfiguruje kanał 1 timera w trybie `PWM mode 1`.
- Parametr `Pulse` - ustawia domyślne wypełnienie.

```
3115 void setPWMPercent(uint16_t duty_percent) {
3116     // Ograniczenie wartosci do 1000
3117     if (duty_percent > 1000)
3118         duty_percent = 1000;
3119
3120     // Obliczenie wypełnienia na podstawie wartości procentowej
3121     uint32_t pulse_length = (htim4.Init.Period + 1) * duty_percent / 1000;
3122     __HAL_TIM_SET_COMPARE(&htim4, TIM_CHANNEL_1, pulse_length);
3123 }
```

Rysunek 4.4: Obliczanie wypełnienia PWM

Funkcja przyjmuje wartość w przedziale 0–1000. W `setPWMPercent`, wypełnienie PWM (duty cycle) jest wyliczane przy użyciu formuły:

$$\text{pulse\_length} = \frac{(\texttt{htim4.Init.Period} + 1) \times \texttt{duty\_percent}}{1000}$$

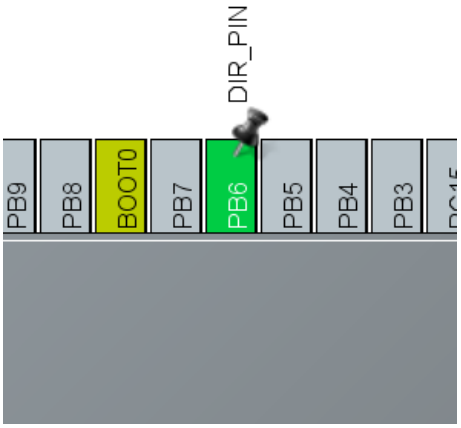
Gdzie:

`htim4.Init.Period` określa maksymalną wartość, do której licznik timera zlicza przed zresetowaniem. W tym przypadku `htim4.Init.Period = 8399`, co oznacza, że timer liczy od 0 do 8399, co daje łącznie 8400 kroków.

`duty_percent` to wartość wejściowa określająca wypełnienie PWM w skali od 0 do 1000. Wartość `0` oznacza 0% wypełnienia (sygnał stale w stanie niskim), wartość `500` oznacza 50% wypełnienia, a wartość `1000` oznacza 100% wypełnienia (sygnał stale w stanie wysokim).

`pulse_length` to wynik obliczeń według podanej formuły. Wynik ten określa wartość rejestru CCR1 (Compare Register). Timer ustawia wyjście w stanie wysokim, gdy aktualna wartość licznika `CNT` jest mniejsza niż `pulse_length`.

4.2 Ustawienie kierunku obrotów silnika



Rysunek 4.5: Ustawienie pinu zmieniającego kierunek obrotu

W stosowanym sterowniku silnika Cytron MD20A jest miejsce na sygnał sterujący kierunkiem obrotów silnika. Pin PB6 został skonfigurowany jako wyjście GPIO.

```
26 #define DIR_PIN GPIO_PIN_6  // Ustawiony na PB6
27 #define DIR_GPIO_PORT GPIOB  // Port dla PB6 to GPIOB
```

Rysunek 4.6: Zdefiniowanie DIR_PIN

W kodzie pin PB6 jest zdefiniowany jako `DIR_PIN` .

```
3125 void setDirection(uint8_t direction) {
3126     // Ustawia kierunek obrotu na pinie DIR_PIN (PB6)
3127     HAL_GPIO_WritePin(DIR_GPIO_PORT, DIR_PIN,
3128             direction ? GPIO_PIN_SET : GPIO_PIN_RESET);
3129 }
```

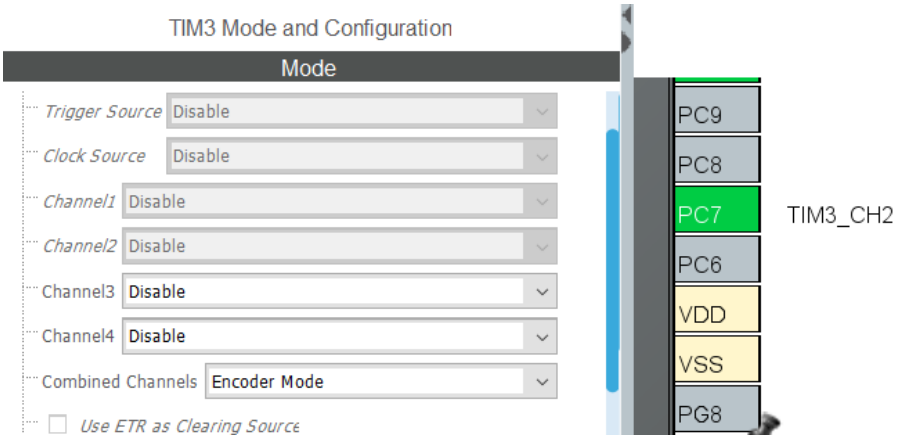
Rysunek 4.7: Funkcja setDirection

Parametr `direction` typu `uint8_t` (wartość logiczna 0 lub 1) decyduje o kierunku obrotów silnika:

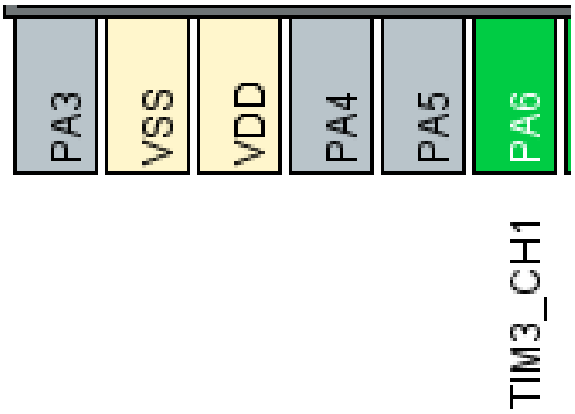
- `0` - resetuje stan pinu (`GPIO_PIN_RESET`).
- `1` - ustawia stan wysoki na pinie (`GPIO_PIN_SET`).

Użycie operatora trójargumentowego (`? :`) pozwala na zapisanie logicznej zależności w jednej linii. `HAL_GPIO_WritePin(...)` to funkcja z biblioteki HAL (Hardware Abstraction Layer) używana do ustawienia stanu pinu GPIO.

4.3 Odczyt enkodera



Rysunek 4.8: Konfiguracja TIM3 na Encoder Mode



Rysunek 4.9: Pin Channel1

Pin PA6 jest ustawiony jako TIM3_CH1 (kanał 1 timera TIM3), wykorzystywany jako jedno z wejść dla sygnału enkodera. Pełni rolę detektora zboczy sygnału przychodzącego z enkodera na kanale A. Pin PC7 został ustawiony jako TIM3_CH2 (kanał 2 timera TIM3). Został zastosowany jako drugie wejście dla sygnału enkodera (kanał B). W połączeniu z kanałem CH1 pozwala na określenie kierunku obrotów enkodera (poprzez analizę faz sygnałów A i B). Timer TIM3 działa w trybie enkodera (Encoder Mode), co pozwala liczyć impulsy generowane przez enkoder obrotowy (ustalać pozycję enkodera).

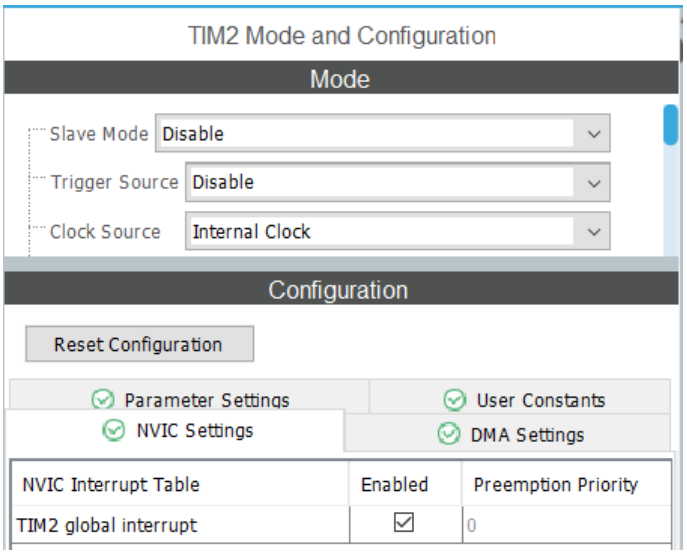
```
3357 static void MX_TIM3_Init(void) {
3358
3359     /* USER CODE BEGIN TIM3_Init 0 */
3360
3361     /* USER CODE END TIM3_Init 0 */
3362
3363     TIM_Encoder_InitTypeDef sConfig = { 0 };
3364     TIM_MasterConfigTypeDef sMasterConfig = { 0 };
3365
3366     /* USER CODE BEGIN TIM3_Init 1 */
3367
3368     /* USER CODE END TIM3_Init 1 */
3369     htim3.Instance = TIM3;
3370     htim3.Init.Prescaler = 0;
3371     htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
3372     htim3.Init.Period = 65535;
3373     htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
3374     htim3.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
3375     sConfig.EncoderMode = TIM_ENCODERMODE_TI1;
3376     sConfig.IC1Polarity = TIM_ICPOLARITY_RISING;
3377     sConfig.IC1Selection = TIM_ICSELECTION_DIRECTTI;
3378     sConfig.IC1Prescaler = TIM_ICPSC_DIV1;
3379     sConfig.IC1Filter = 0;
3380     sConfig.IC2Polarity = TIM_ICPOLARITY_RISING;
3381     sConfig.IC2Selection = TIM_ICSELECTION_DIRECTTI;
3382     sConfig.IC2Prescaler = TIM_ICPSC_DIV1;
3383     sConfig.IC2Filter = 0;
3384     if (HAL_TIM_Encoder_Init(&htim3, &sConfig) != HAL_OK) {
3385         Error_Handler();
3386     }
3387     sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
3388     sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
3389     if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig)
3390             != HAL_OK) {
3391         Error_Handler();
3392     }
3393     /* USER CODE BEGIN TIM3_Init 2 */
3394
3395     /* USER CODE END TIM3_Init 2 */
3396
3397 }
```

Rysunek 4.10: Konfiguracja TIM3 w kodzie

Najważniejsze parametry timera to:

- `CounterMode = TIM_COUNTERMODE_UP` - Licznik pracuje w trybie zliczania w górę (od 0 do wartości `Period`).
- `IC1Polarity = TIM_ICPOLARITY_RISING` - Kanał 1 wykrywa zbocza narastające sygnału A.
- `IC2Polarity = TIM_ICPOLARITY_RISING` - Kanał 2 wykrywa zbocza narastające sygnału B.
- Filtr wejściowy (`IC1Filter`, `IC2Filter`) = **0** - oznacza brak dodatkowego filtrowania sygnałów wejściowych.

4.4 Przerwania



Rysunek 4.11: Konfiguracja TIM2

Clock Source = Internal Clock - timer korzysta z wewnętrznego zegara mikrokontrolera (APB1=84MHz). TIM2 global interrupt = Enabled - przerwanie timera TIM2 jest włączone, co pozwala obsługiwać jego zdarzenia w funkcji callback (`HAL_TIM_PeriodElapsedCallback`). Preemption Priority = 0 - ustawiono najwyższy priorytet przerwania (wartość 0 oznacza najwyższy poziom w STM32).

```
3315 static void MX_TIM2_Init(void) {
3316
3317     /* USER CODE BEGIN TIM2_Init 0 */
3318
3319     /* USER CODE END TIM2_Init 0 */
3320
3321     TIM_ClockConfigTypeDef sClockSourceConfig = { 0 };
3322     TIM_MasterConfigTypeDef sMasterConfig = { 0 };
3323
3324     /* USER CODE BEGIN TIM2_Init 1 */
3325
3326     /* USER CODE END TIM2_Init 1 */
3327     htim2.Instance = TIM2;
3328     htim2.Init.Prescaler = 1680;
3329     htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
3330     htim2.Init.Period = 99;
3331     htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
3332     htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
3333     if (HAL_TIM_Base_Init(&htim2) != HAL_OK) {
3334         Error_Handler();
3335     }
3336     sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
3337     if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK) {
3338         Error_Handler();
3339     }
3340     sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
3341     sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
3342     if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig)
3343             != HAL_OK) {
3344         Error_Handler();
3345     }
```

Rysunek 4.12: Konfiguracja TIM2 w kodzie

`Prescaler = 1680` - zmniejsza częstotliwość taktowania timera. Dla zegara `APB1 = 84 MHz`, częstotliwość zliczania timera wynosi:

$$f_{\text{timer}} = \frac{f_{\text{APB1}}}{\text{Prescaler} + 1} = \frac{84\,\text{MHz}}{1680} = 50\,\text{kHz}.$$

Period = `99` - timer odlicza od 0 do 99, co oznacza, że przerwanie występuje co 100 kroków. Przy częstotliwości zliczania 50 kHz przerwania występują co:

$$T_{\text{przerwania}} = \frac{1}{f_{\text{timer}} \times (\text{Period} + 1)} = \frac{1}{50\,\text{kHz} \times 100} = 2\,\text{ms}.$$

CounterMode = TIM_COUNTERMODE_UP - timer liczy w górę (od 0 do wartości określonej przez `Period`).

```
3566 void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
3567     if (htim == &htim2) {  // Przerwanie timera TIM2
3568         // Odczytaj aktualną pozycję enkodera
3569         current_position = __HAL_TIM_GET_COUNTER(&htim3);
3570         __HAL_TIM_SET_COUNTER(&htim3, 0);
3571         actual_speed = (current_position)*20;
3572         if(i_tab<3000){
3573             speed_tab[i_tab] = actual_speed;
3574             pwm = (tab_u[i_tab]*1000)/24;
3575             setPWMPercent(pwm);
3576         }
3577         i_tab = i_tab+1;
3578         if(i_tab>3000 && end == 0){
3579             i_tab=3000;
3580             end = 1;
3581             printf("end: %d\r\n", actual_speed);
3582             setPWMPercent(0);
3583         }
3584     }
3585 }
```

Rysunek 4.13: Przerwanie

W warunku `if` następuje sprawdzenie, czy przerwanie pochodzi od `TIM2`. `__HAL_TIM_GET_COUNTER` pobiera aktualną wartość licznika `TIM3`, który zlicza impulsy z enkodera. Prędkość jest obliczana jako liczba impulsów (`current_position`) pomnożona przez skalę (w tym przypadku 20). Wartość skali zależy od rozdzielczości enkodera i okresu przerwania.

`speed_tab[i_tab] = actual_speed` zapisuje obliczoną prędkość do tablicy `speed_tab`, w której przechowywane są dane pomiarowe.

Funkcja `setPWMPercent(pwm)` ustawia wypełnienie sygnału PWM w zależności od wartości przekazywanej w `tab_u`.

Podsumowując, callback `HAL_TIM_PeriodElapsedCallback` integruje odczyt enkodera i sterowanie PWM, zapewniając płynne działanie systemu napędowego.

4.5 Główny kod

```
91  uint16_t speed_tab [3000];
92  float tab_u [3000] = {19.72112113150422,
93          24.0,
94          24.0,
95          24.0,
96          24.0,
97          24.0,
98          24.0,
99          24.0,
100         24.0,
101         24.0,
102         24.0,
103         24.0,
104         24.0,
```

Rysunek 4.14: Przekazywana tablica napięć

Prezentowany kod do obsługi stm jest przystosowany do sprawdzenia działania SSN generującego wartości napięcia przy użyciu realnego układu pomiarowego. Przekazywana jest tablica 3000 napięć do funkcji sterowania PWM (za pomocą `setPWMPercent`) a wartości w niej zawarte są przekształcane na odpowiednie wypełnienie sygnału PWM.

```
3158    /* Initialize all configured peripherals */
3159    MX_GPIO_Init();
3160    MX_ETH_Init();
3161    MX_USART3_UART_Init();
3162    MX_USB_OTG_FS_PCD_Init();
3163    MX_TIM3_Init();
3164    MX_TIM4_Init();
3165    MX_TIM2_Init();
3166    /* USER CODE BEGIN 2 */
3167
3168    HAL_NVIC_EnableIRQ(TIM2_IRQn);          // Włącz TIM2 IRQ
3169    __HAL_TIM_SET_COUNTER(&htim3, 0);
3170
3171    // Startujemy timer bazowy TIM2 z przerwaniami
3172    HAL_TIM_Base_Start_IT(&htim2);
3173    if (HAL_TIM_Base_Start_IT(&htim2) != HAL_OK) {
3174        printf("Error: Could not start TIM2 with interrupts!\r\n");
3175    }
3176
3177    // Startujemy timer w trybie enkodera
3178    HAL_TIM_Encoder_Start(&htim3, TIM_CHANNEL_ALL);
3179
3180
3181    // Startujemy PWM na kanale 1 timera 4
3182    HAL_TIM_PWM_Start(&htim4, TIM_CHANNEL_1);
3183
3184    // Ustawienie początkowego wypełnienia PWM na 10%
3185    setPWMPercent(0);
```

Rysunek 4.15: Inicjalizowanie peryferiów

W `main` inicjalizowane są peryferia:

- `HAL_TIM_Base_Start_IT(&htim2)` - uruchamia timer bazowy `TIM2` w trybie z przerwaniami.
- `HAL_TIM_Encoder_Start(&htim3, TIM_CHANNEL_ALL)` - załącza timer `TIM3` w trybie enkodera.

- `HAL_TIM_PWM_Start(&htim4, TIM_CHANNEL_1)` - uruchamia generowanie sygnału PWM na kanale 1 timera `TIM4`.

Dodatkowo początkowe wypełnienie PWM ustawiono na 0%, aby silnik pozostał nieruchomy do czasu podania napięcia sterującego.

```
3193    while (1) {
3194        //e_current = 1500 - actual_speed;
3195        //e_sum = (e_sum+e_current)*dt;
3196        //e_d = (e_current - e_last)/dt;
3197        //pwm = kp*e_current + ki*e_sum + kd*e_d;
3198        //printf("pwm: %d\r\n", pwm);
3199        //setPWMPercent(pwm);
3200        setDirection(1);    // Ustaw kierunek na 1 (0 oznacza odwrotny kierunek)
3201        if(end==1){
3202            HAL_Delay(5000);
3203            for(int i =0; i<3000; i++){
3204
3205                printf("%d,", speed_tab[i]);
3206                HAL_Delay(10);
3207
3208                //end = 0;
3209            }
3210        }
```

Rysunek 4.16: Pętla while

Pętla `while(1)` w tym fragmencie odpowiada za główne działanie programu i jest wykonywana w nieskończoność. Zakomentowany kod dotyczy adaptacji programu do sterowania silnikiem przy pomocy nastaw PID, jednak finalnie nie został użyty.

Funkcja `setDirection(1)` ustawia pin sterujący (`DIR_PIN`) w stan wysoki, co przekłada się na ustawienie jednego z kierunków obrotu silnika.

Warunek `if (end == 1)` sprawdza flagę `end`, która sygnalizuje zakończenie zbierania danych pomiarowych.

Zagnieżdżona pętla `for` wypisuje wartości z tablicy `speed_tab` (zmierzone prędkości) na UART.

Rozdział 5

Zastosowane algorytmy

5.1 Algorytm genetyczny sterujący nastawą regulatora PID

Do poprawnego zadziałania algorytmu genetycznego generującego nastawy dla wszystkich członów regulatora PID potrzeba modelu silnika DC i regulatora PID przy pomocy których można przeprowadzić symulacje dla rozpatrywanych wartości członów. Kody napisano w pythonie, w środowisku pycharm. Skorzystano z takich bibliotek jak: numpy, matplotlib i json.

5.1.1 Model silnika

```
4   class DCMotor:  3 usages
5       def __init__(self, Ua=24, Ra=4.3, La=0.0015, k=0.05, J=1.03e-5, m_op=0.06, dt=0.001):
6
7
8
9           self.Ua = Ua  # Napięcie zasilania
10          self.Ra = Ra  # Rezystancja w obwodzie twornika
11          self.La = La  # Indukcyjność w obwodzie twornika
12          self.k = k  # Stała momentu mechanicznego
13          self.J = J  # Moment bezwładności wału silnika
14          self.m_op = m_op  # Moment oporu
15          self.dt = dt  # Krok czasowy symulacji
16          self.I_max = 3.9  # Maksymalny prąd twornika [A]
17          self.reset()  # Inicjalizacja stanu silnika
```

Rysunek 5.1: Utworzenie klasy DCMotor

Inicjalizacja klasy DCMotor która odpowiada za matematyczną symulację działania silnika prądu stałego. Parametry zostały wzięte z karty katalogowej rzeczywistego silnika DC zastosowanego w stanowisku pomiarowym.

```
36      def step(self):  1 usage
37
38          # Równanie opisujące prąd twornika
39          self.ia_dot = (self.Ua - self.k * self.w - self.ia * self.Ra) / self.La
40          self.ia += (self.ia_dot + self.ia_dot_prev) / 2 * self.dt  # Całkowanie metodą trapezów
41          self.ia = np.clip(self.ia, -self.I_max, self.I_max)  # Ograniczenie prądu
42          self.ia_dot_prev = self.ia_dot  # Aktualizacja wartości poprzedniej
43
44          # Równanie momentu na wale: przyspieszenie kątowe
45          self.w_dot = (self.k * self.ia - self.m_op) / self.J
46          self.w_dot = np.clip(self.w_dot, -1e4,  a_max: 1e4)  # Ograniczenie przyspieszenia (przykładowe wartości)
47
48          # Aktualizacja prędkości kątowej
49          self.w += (self.w_dot + self.w_dot_prev) / 2 * self.dt
50          max_rpm = 5000  # Maksymalna prędkość w RPM
51          self.w = np.clip(self.w, -max_rpm * 2 * np.pi / 60, max_rpm * 2 * np.pi / 60)
52
53          self.w_dot_prev = self.w_dot  # Aktualizacja wartości poprzedniej
54
55          # Przeliczenie prędkości kątowej na obr/min (RPM)
56          self.predkosc = self.w * 60 / (2 * np.pi)
```

Rysunek 5.2: Funkcja step

Funkcja step odpowiada za jeden krok czasowy w symulacji. W trakcie jednego kroku zostaje wyliczony prąd twornika $\dot{i_a}$ (linijka 37) przy pomocy równania obwodu elektrycznego silnika DC:

$$\dot{i_a} = \frac{U_a - k \cdot \omega - i_a \cdot R_a}{L_a}.$$

U_a – napięcie zasilania,
$k \cdot \omega$ – siła przeciwelektromotoryczna (back-EMF),
$i_a \cdot R_a$ – spadek napięcia na rezystancji twornika,
L_a – indukcyjność twornika.

Następnie zachodzi aktualizacja wartości prądu $\dot{i_a}$ metodą trapezów (linijka 38). Polega ona na uwzględnieniu zarówno bieżącej pochodnej prądu, jak i poprzedniej ($\dot{i}_a$ dla $\dot{i}_{a,\text{prev}}$) w celu zmniejszenia błędów numerycznych:

$$i_a \leftarrow i_a + \frac{\Delta t}{2}\left(\dot{i}_a(t) + \dot{i}_a(t - \Delta t)\right).$$

Zastosowane ograniczenie prądu zapobiega przyjmowaniu nierealistycznych wartości które mogłyby zdestabilizować model. W linijce 40 zachodzi zapisanie bieżącej wartości prądu aby móc ją wykorzystać w następnym kroku. Równanie zawarte w 43 linijce:

$$\dot{\omega} = \frac{k \cdot i_a - m_{op}}{J}$$

wynika z drugiej zasady dynamiki dla ruchu obrotowego, gdzie:

$k \cdot i_a$ – moment elektromagnetyczny,
m_{op} – moment obciążenia (oporu),
J – moment bezwładności wału silnika.

Aby symulacja pozostała numerycznie stabilna, przyspieszenie $\dot{\omega}$ przystosowuje się do zakresu $[-10^4, 10^4]$. Z kolei prędkość ω ogranicza się do maksymalnej wartości odpowiadającej ± 5000 RPM. Aktualizacja prędkości kątowej jest wykonywana również przy pomocy metody trapezów:

$$\omega \leftarrow \omega + \frac{\Delta t}{2}\left(\dot{\omega}(t) + \dot{\omega}(t - \Delta t)\right).$$

Dzięki powyższemu kodowi, w każdym wywołaniu `step()` silnik DC jest aktualizowany w czasie o jeden krok, uwzględniając kluczowe zjawiska elektryczne i mechaniczne zgodnie z podstawowymi równaniami fizycznymi.

5.1.2 Model regulatora PID

Podstawową postać równania PID można zapisać jako:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{d}{dt} e(t),$$

gdzie:

$e(t)$ – bieżący błąd (wartość zadana minus wartość aktualna),

K_p, K_i, K_d – odpowiednio wzmocnienia członu proporcjonalnego, całkującego i różniczkującego.

W implementacjach dykretnych (takich jak np. tworzenie modelu w pythonie) zamieniamy całkowanie na sumowanie błędów w krótkich krokach czasowych Δt, a różniczkowanie na różnicę błędów w kolejnych krokach.

```
3   class PIDController:  3 usages
25      def calculate(self, predkosc_aktualna, dt=0.001):  1 usage
26
27          # Obliczanie bieżącego błędu
28          e_current = self.predkosc_zadana - predkosc_aktualna
29
30          # Człon całkujący z uwzględnieniem dt (anti-windup)
31          self.e_sum += e_current * dt
32          if self.ki > 0:
33              if self.e_sum * self.ki > self.u_max:
34                  self.e_sum = self.u_max / self.ki
35              elif self.e_sum * self.ki < self.u_min:
36                  self.e_sum = self.u_min / self.ki
37          # Człon różniczkujący na podstawie błędu, nie prędkości
38          de = (e_current - self.e_last) / dt
39
40          # Prawo sterowania PID
41          u = self.kp * e_current + self.ki * self.e_sum - self.kd * de
42
43          # Ograniczenie napięcia sterującego
44          u = np.clip(u, self.u_min, self.u_max)
45
46          # Aktualizacja poprzedniego błędu
47          self.e_last = e_current
48
49          if self.debug:
50              print(f"PID: e={e_current:.2f}, e_sum={self.e_sum:.2f}, de={de:.2f}, u={u:.2f}")
51
52          return u
```

Rysunek 5.3: Metoda calculate z klasy PIDControler

Linijki 29-34 odpowiedzialne są za człon całkujący i mechanizm anti-windup. Polega on na tym, że jeżeli wartość całki (pomnożona przez wzmocnienie całkujące k_i przekracza dopuszczalne wyjście regulatora, to ograniczamy ją do takiego poziomu, by po przemnożeniu przez k_i dawała co najwyżej $u_{\max}$ (lub co najmniej $u_{\min}$).

W praktyce oznacza to, że człon całkujący nie będzie się już dalej zwiększał (ani zmniejszał) w sytuacji, gdy regulator i tak jest na granicy sterowania - napięcie wyszło poza dopuszczalny zakres. Dzięki temu, gdy wyjście przestaje być nasycone, integralny licznik błędu (`e_sum`) nie jest sztucznie powiększany, co pozwala znacznie szybciej i łagodniej powrócić do poprawnego sterowania. Równanie prawa sterowania PID ma postać:

$$u = K_p \cdot e_c + K_i \cdot e_s - K_d \cdot \frac{\Delta e}{\Delta t}$$

Gdzie:

- e_c - `e_current`, aktualny błąd
- e_s - `e_sum`, skumulowany błąd

W zależności od przyjętej definicji błędu i wymaganej reakcji układu, człon różniczkujący może działać jako element tłumiący (zmniejszający sygnał sterujący) lub wzmacniający.

W przedstawionym przypadku stosowana jest pierwsza definicja błędu:

$$e(t) = r - y$$

Gdzie:

- r – wartość zadana,
- y – wartość aktualna.

Kod ograniczający sygnał sterujący (linijka 43) zapobiega wyjściu napięcia sterującego poza dopuszczalny zakres $-24\,\mathrm{V}, +24\,\mathrm{V}$. Chroni to układ przed zbyt dużym sygnałem na wyjściu, który w rzeczywistych zastosowaniach nie byłby fizycznie możliwy lub bezpieczny.

Gdy `self.debug` jest włączone, program wypisuje w terminalu wartości poszczególnych składników regulatora (błąd, sumę błędu, pochodną błędu i samo napięcie sterujące). Pozwala to łatwo monitorować działanie PID w czasie.

5.1.3 Algorytm genetyczny

Konstruktor klasy `GeneticAlgorithm` jest odpowiedzialny za inicjalizację parametrów algorytmu oraz przygotowanie pierwszej populacji osobników.

```
9   class GeneticAlgorithm:  2 usages
10      def __init__(self, pop_size=100, generations=25, mutation_rate=0.4,
11                   kp_range=(0, 1), ki_range=(0, 1), kd_range=(0, 1)):
12          """
13          Konstruktor algorytmu genetycznego.
14
15          Parametry:
16          - pop_size: liczba osobników w populacji
17          - generations: liczba pokoleń ewolucji
18          - mutation_rate: współczynnik mutacji
19          - kp_range, ki_range, kd_range: zakresy dozwolonych wartości Kp, Ki, Kd
20
21          """
22          self.pop_size = pop_size
23          self.generations = generations
24          self.mutation_rate = mutation_rate
25          self.kp_range = kp_range
26          self.ki_range = ki_range
27          self.kd_range = kd_range
28
29          # Inicjalizacja populacji losowymi wartościami [Kp, Ki, Kd]
30          self.population = np.random.uniform(
31              low: [kp_range[0], ki_range[0], kd_range[0]],
32              high: [kp_range[1], ki_range[1], kd_range[1]],
33              size: (pop_size, 3)
34          )
35
36          # Lista do zapisywania najlepszych wyników w kolejnych pokoleniach
37          self.history = []
38
39      def fitness(self, individual, predkosc_zadana=1500, moment=1.03e-6):
```

Rysunek 5.4: Konstruktor algorytmu genetycznego

Populacja jest generowana losowo za pomocą funkcji `np.random.uniform`, która tworzy macierz o wymiarach $(\text{pop\_size}, 3)$, gdzie każda kolumna odpowiada K_p, K_i, K_d. Zakresy wartości są określone przez argumenty `kp_range`, `ki_range`, `kd_range`. Wzór losowania:

$$K_x = \text{rand}(l_x, h_x), \quad x \in \{p, i, d\},$$

Gdzie:

- l_x – dolna granica zakresu dla parametru K_x,
- h_x – górna granica zakresu dla parametru K_x,
- rand – funkcja losująca liczby z przedziału $[0, 1)$.

```
237         # Tworzymy nową populację: najlepsi + potomkowie
238         new_population = []
239         while len(new_population) < self.pop_size - len(best_individuals):
240             indices = np.random.choice(len(best_individuals), size: 2, replace=False)
241             parent1, parent2 = best_individuals[indices[0]], best_individuals[indices[1]]
242 
243             child = (parent1 + parent2) / 2
244 
245             # Mutacja
246             mutation = np.random.uniform(-0.2, high: 0.2, child.shape) * self.mutation_rate
247             child += mutation
248 
249             # Ograniczanie do przedziałów
250             child = np.clip(
251                 child,
252                 a_min: [self.kp_range[0], self.ki_range[0], self.kd_range[0]],
253                 a_max: [self.kp_range[1], self.ki_range[1], self.kd_range[1]]
254             )
255 
256             # Wymuszamy kd != 0
257             if child[2] <= 1e-6:
258                 child[2] = np.random.uniform(self.kd_range[0] + 1e-3, self.kd_range[1])
259 
260             new_population.append(child)
261 
262         self.population = np.vstack((best_individuals, new_population))
```

Rysunek 5.5: Tworzenie nowej populacji

Zastosowano selekcję elitarną. Najlepsze osobniki z poprzedniej generacji są wybierane na podstawie ich wartości funkcji oceny (fitness). Dwóch rodziców jest losowo wybieranych spośród najlepszych jednostek (`best_individuals`). Potomkowie są generowani jako średnia arytmetyczna parametrów rodziców (linijka 242):

$$c = \frac{p_1 + p_2}{2}$$

Gdzie:

- c – `child`, potomek,
- p_1 – `parent1`, pierwszy rodzic,
- p_2 – `parent2`, drugi rodzic.

Krzyżowanie zapewnia, że potomkowie dziedziczą cechy obu rodziców, co pozwala na poprawną eksplorację przestrzeni parametrów. Potomkowie są modyfikowani za pomocą mutacji, która polega na dodaniu losowej wartości z rozkładu równomiernego:

$$m = \text{rand}(-0.2, 0.2) \cdot r_m$$

Gdzie:

- m - `mutation`, wartość mutacji,
- rand$(-0.2, 0.2)$ - losowa liczba z zakresu $[-0.2, 0.2]$,
- r_m - `mutation_rate`, współczynnik mutacji.

Mutacja wprowadza różnorodność do populacji, co zapobiega utknięciu algorytmu w lokalnym minimum. Parametry potomków są ograniczane do dozwolonych zakresów za pomocą funkcji `np.clip`, aby uniknąć generowania wartości, które mogą prowadzić do niestabilności regulatora. Prezentowany algorytm ma tendencję do ustawiania wartości K_d na 0. Aby temu zapobiec, wymusza się $K_d > 0$. Jeśli K_d jest zbyt małe, zostaje przypisane nowe losowe minimalne K_d.

```
173             # Obliczanie fitness
174             fitness_score = 0.4 * settling_time + 0.4 * overshoot_ratio + 0.2 * abs(rpm - achieved_speed) / rpm
175
176             # Dodatkowe kary
177             if overshoot > 2 * rpm:
178                 fitness_score += 500  # duże przeregulowanie
179             if achieved_speed < 0:
180                 fitness_score += 1000  # prędkość ujemna
181
182             # Kara za oscylacje: *500
183             osc_penalty = (osc_penalty_accum / (rpm ** 2 * 3000)) * 500
184             fitness_score += osc_penalty
185
186             # Kara za sumaryczny błąd (średni w czasie)
187             avg_error = error_penalty / 3000
188             fitness_score += (avg_error / rpm) * 50
```

Rysunek 5.6: Obliczanie funkcji oceny

Funkcja oceny (`fitness_score`) w algorytmie genetycznym umożliwia porównanie jakości sprawdzanych zestawów parametrów PID. Czas ustalania (`settling_time`) mierzy, ile czasu system potrzebuje, aby prędkość weszła w akceptowalny zakres wokół wartości zadanej ($\pm 5\%$) i pozostała w nim do końca symulacji. Jeśli `current_speed` nigdy nie spełni tego warunku, zakładany jest maksymalny czas symulacji ($t_s = 3\,\mathrm{s}$), co oznacza, że system nie zdążył się ustabilizować w przewidzianym czasie. W algorytmie genetycznym krótszy czas ustalania skutkuje mniejszą wartością `fitness_score`, co oznacza lepsze parametry PID.

```
92              # Maksymalne przeregulowanie
93              if current_speed > predkosc_zadana:
94                  max_overshoot = max(max_overshoot, current_speed - predkosc_zadana)
95
96              # Czas ustalenia: gdy prędkość wejdzie w ±5% wartości zadanej
97              if abs(current_speed - predkosc_zadana) < 0.05 * predkosc_zadana and settling_time is None:
98                  settling_time = t * motor.dt
99
100             achieved_speed = current_speed
101
102             # Kara za oscylacje: kwadrat różnic kolejnych prędkości
103             if prev_speed is not None:
104                 delta_speed = current_speed - prev_speed
105                 osc_penalty_accum += delta_speed ** 2
106             prev_speed = current_speed
107
108             # Kara za sumaryczny błąd
109             error = abs(current_speed - predkosc_zadana)
110             error_penalty += error
```

Rysunek 5.7: Wprowadzone kary

Przeregulowanie (`max_overshoot`) to największe przekroczenie wartości zadanej (`predkosc_zadana`) przez prędkość obrotową w trakcie symulacji.

Wzór:

$$\text{MO} = \max(\text{MO}, v_c - v_r) \quad \text{dla } v_c > v_r.$$

Gdzie:

- MO - `max_overshoot`, maksymalne przeregulowanie,
- v_c - `current_speed`, aktualna prędkość,
- v_r - `predkosc_zadana`, prędkość zadana.

Kary są wprowadzane do funkcji oceny, aby zredukować niepożądane zachowania układu i poprawić jego stabilność.

Dla dużego przeregulowania, jeśli `overshoot` $> 2 \cdot$ `rpm`, do `fitness_score` dodawana jest kara o wartości 500. Ma to na celu zapobieganie wybieraniu parametrów PID, które powodują niestabilność układu.

Jeśli prędkość obrotowa (`achieved_speed`) osiągnie wartość ujemną (co jest fizycznie niemożliwe dla systemu), wprowadzana jest kara w wysokości 1000. Takie podejście eliminuje konfiguracje prowadzące do nielogicznych wyników.

Oscylacje mierzą zmienność prędkości obrotowej w czasie. Są one określane jako suma kwadratów różnic pomiędzy kolejnymi wartościami `current_speed` w trakcie symulacji:

$$P = \sum (v_c - v_p)^2$$

Gdzie:

- P - `osc_penalty_accum`, skumulowana kara za oscylacje,
- v_c - `current_speed`, aktualna prędkość,
- v_p - `prev_speed`, poprzednia prędkość.

Im większa wartość `P`, tym większe wahania prędkości.

Sumaryczny błąd (`error_penalty`) mierzy całkowity uchyb pomiędzy prędkością aktualną a prędkością zadaną w trakcie całej symulacji:

$$E = \sum |v_r - v_c|$$

Gdzie:

- E - `error_penalty`, skumulowana kara za błąd,
- v_r - `predkosc_zadana`, prędkość zadana,
- v_c - `current_speed`, aktualna prędkość.

Niższa wartość `error_penalty` wskazuje na lepsze dopasowanie dynamiki układu do wartości zadanej, co świadczy o wyższej jakości regulacji.

Funkcja oceny stanowi kombinację ważoną wyżej opisanych metryk oraz kar. Wyrażenie funkcji oceny przedstawia się następująco:

$$F = 0.4 \cdot T_s + 0.4 \cdot O_r + 0.2 \cdot \frac{|N - v_a|}{N} + P_k$$

Gdzie:

- F - `fitness_score`, wynik funkcji dopasowania,
- T_s - `settling_time`, czas ustalania (ważony 40%),
- O_r - `overshoot_ratio`, przeregulowanie względne (ważone 40%),
- $\frac{|N-v_a|}{N}$ - ostateczny błąd względny prędkości (ważony 20%),
- N - `rpm`, zadana liczba obrotów na minutę,
- v_a - `achieved_speed`, osiągnięta prędkość,
- P_k - `kary`, suma wszystkich kar.

```
225         # Re-inicjalizacja przy stagnacji
226         if stagnation_counter >= 4 and generation < self.generations - 2:
227             print(f"Stagnacja w generacji {generation + 1}, losowa re-inicjalizacja populacji.")
228             random_individuals = np.random.uniform(
229                 low: [self.kp_range[0], self.ki_range[0], self.kd_range[0]],
230                 high: [self.kp_range[1], self.ki_range[1], self.kd_range[1]],
231                 size: (self.pop_size - len(best_individuals), 3)
232             )
233             self.population = np.vstack((best_individuals, random_individuals))
234             stagnation_counter = 0
235             continue
```

Rysunek 5.8: Mechanizm reinicjalizacji populacji w przypadku stagnacji

W algorytmie zastosowano mechanizm detekcji stagnacji. Działa w sytuacji, gdy przez kilka kolejnych pokoleń nie następuje istotna poprawa wartości funkcji oceny, co wskazuje na brak postępu w ewolucji populacji.

Stagnacja jest wykrywana przez porównanie najlepszego wyniku funkcji oceny w bieżącej generacji z wynikiem z poprzedniej generacji. Jeśli różnica między nimi jest mniejsza od zadanej tolerancji (10^{-6}) przez określoną liczbę pokoleń (4 pokolenia), uznaje się, że populacja znajduje się w stanie stagnacji.

W przypadku wykrycia stagnacji algorytm przeprowadza częściową reinicjalizację populacji. Najlepszych N osobników z bieżącej populacji (np. 15 najlepszych) pozostawia się bez zmian, ponieważ są one uznawane za potencjalnie dobre rozwiązania. Reszta populacji (`pop_size` $- N$) jest losowo generowana w zadanych przedziałach wartości parametrów K_p, K_i oraz K_d, co wprowadza nową różnorodność do algorytmu. Dzięki temu możliwe jest dalsze szukanie precyzyjniejszych nastaw członów regulatora. Po przeprowadzeniu reinicjalizacji licznik stagnacji zostaje wyzerowany, co pozwala na dalsze monitorowanie postępu w kolejnych pokoleniach.

5.1.4 Symulacja

```
79      # Symulacja 3 s, 3000 kroków
80      for t in range(3000):
81          Ua = pid.calculate(motor.predkosc, dt=motor.dt)
82          motor.Ua = Ua
83          motor.step()
84
85          # Rejestracja napięcia w tablicy voltage_data
86          voltage_data.append(Ua)
87
88          current_speed = motor.predkosc
89          speed_data.append(current_speed)
90          t_data.append(t * motor.dt)
```

Rysunek 5.9: Przeprowadzenie symulacji

Symulacja jest przeprowadzana w iteracjach krokowych z krokiem czasowym $dt = 0.001$ przez 3000 kroków (3 s). W każdej iteracji napięcie sterujące `Ua` jest obliczane przez regulator PID na podstawie aktualnej prędkości silnika, zgodnie z wywołaniem `pid.calculate`(`motor.predkosc`, $dt =$ `motor.dt`). Obliczone napięcie jest następnie przypisywane do wejścia zasilania silnika jako `motor.Ua`.

Po obliczeniu napięcia sterującego model silnika jest aktualizowany za pomocą metody `motor.step()`, co powoduje zmianę zmiennych stanu, takich jak prędkość obrotowa i prąd. W trakcie każdej iteracji obliczone wartości są rejestrowane w odpowiednich listach: napięcie `Ua` w `voltage_data` (dane potrzebne do wytrenowania sztucznej sieci neuronowej potrafiącej bezpośrednio sterować silnikiem DC bez pomocy regulatora PID), prędkość obrotowa silnika w `speed_data`, a aktualny czas ($t \cdot dt$) w `t_data`.

Symulacja trwa do momentu osiągnięcia $t = 3$ s. Po zakończeniu wszystkie zebrane dane, takie jak przebieg prędkości, napięcia i czas, są dostępne do dalszej analizy.

```
113     results = load_results()
114     predkosci = [500, 1000, 1500, 2000]
115     momenty_bezwladnosci = [1.03e-5, 5.03e-5, 9.03e-5, 13.03e-5, 13.03e-4]
```

Rysunek 5.10: Zestaw parametrów sprawdzany podczas symulacji

Tabela 5.1: Kombinacje prędkości i momentów bezwładności testowane w trakcie symulacji

Prędkość (RPM)	**Moment bezwładności (kg·m²)**
500	1.03×10^{-5}
500	5.03×10^{-5}
500	9.03×10^{-5}
500	1.30×10^{-4}
500	1.30×10^{-3}
1000	1.03×10^{-5}
1000	5.03×10^{-5}
1000	9.03×10^{-5}
1000	1.30×10^{-4}
1000	1.30×10^{-3}
1500	1.03×10^{-5}
1500	5.03×10^{-5}
1500	9.03×10^{-5}
1500	1.30×10^{-4}
1500	1.30×10^{-3}
2000	1.03×10^{-5}
2000	5.03×10^{-5}
2000	9.03×10^{-5}
2000	1.30×10^{-4}
2000	1.30×10^{-3}

Po przeprowadzeniu symulacji wyniki zapisywane są w pliku results.json który zawiera dane między innymi dotyczące oczekiwanej prędkości obrotowej w danym przykładzie, momencie obciążenia, parametrze fitness i nastawach członów PID.

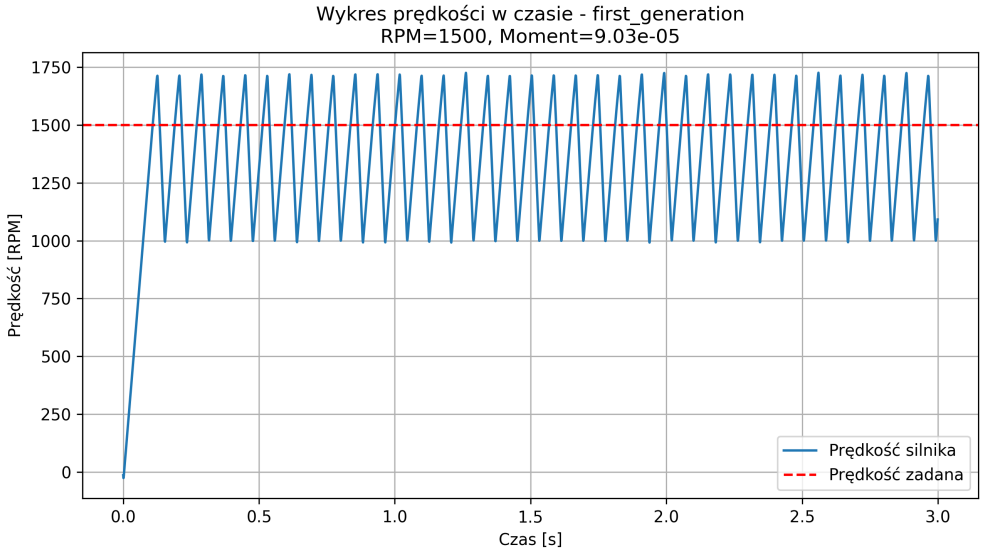

```
"1500_9.03e-05": {
  "rpm": 1500,
  "moment_bezwladnosci": 9.03e-05,
  "kp": 0.39590759032226563,
  "ki": 0.5655928252090571,
  "kd": 0.0004539305912765281,
  "fitness": 1.2031147370566253,
  "additional_data": {
    "time_to_settle": 0.10400000000000001,
    "overshoot": 35.91727429462753,
    "achieved_speed": 1500.4646740455873
  }
},
```


Rysunek 5.11: format zapisu otrzymanych danych w pliku json

W trakcie symulacji generowane są również pliki voltage_data które oprócz danych występujących w results.json zawierają jeszcze serię 3000 nastaw napięcia wynikających z każdego kroku w pętli symulacji trwającej 3 sekundy. Dane te będą użyte do trenowania SSN sterującego bezpośrednio nastawą silnika DC.

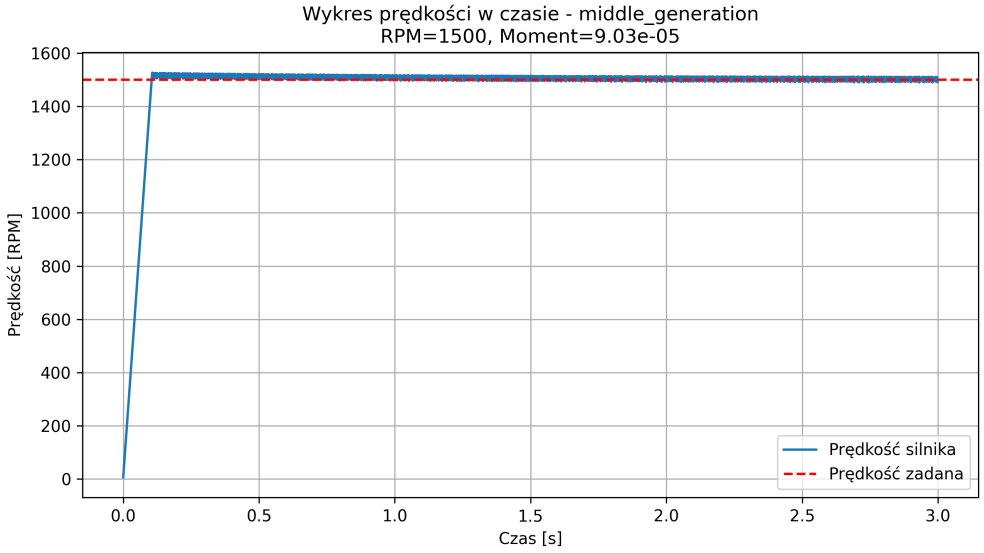
5.1.5 Otrzymane wyniki

Wyniki dla przykładowego scenariusza symulacji (RPM = 1500, moment = $9.03 \times 10^{-5} \mathrm{kg} \cdot \mathrm{m}^2$, $\mathrm{pop\_size} = 100, \mathrm{generations} = 25, \mathrm{mutation\_rate} = 0.05$) wyglądają następująco:



Rysunek 5.12: Wykres prędkości przy zastosowaniu parametrów PID uzyskanych w pierwszej generacji AG

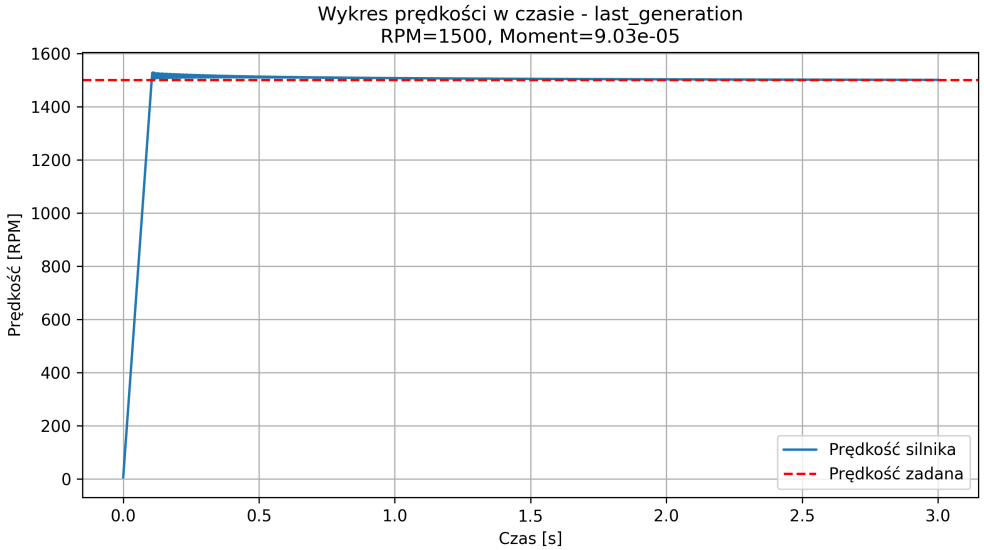
W pierwszej, losowo wygenerowanej generacji można zauważyć wyraźne oscylacje oraz znaczące przeregulowanie w stosunku do prędkości zadanej wynoszącej 1500 RPM. System nie zdołał osiągnąć stabilności w akceptowalnym czasie; po wykresie widać że nie był nawet tego bliski. Maksymalne przeregulowanie przekraczało 250 RPM, co wskazuje na przypadkowy dobór parametrów PID, charakterystyczny dla początkowej fazy ewolucji algorytmu genetycznego.



Rysunek 5.13: Wykres prędkości przy zastosowaniu parametrów PID uzyskanych w środkowej generacji AG

W środkowej generacji układ osiąga znacznie lepsze właściwości dynamiczne w porównaniu z wcześniej prezentowanym. Oscylacje zostały praktycznie wyeliminowane, a czas ustalania wynosi około 0,2 sekundy. Po osiągnięciu prędkości zadanej, system utrzymuje stabilność z minimalnym

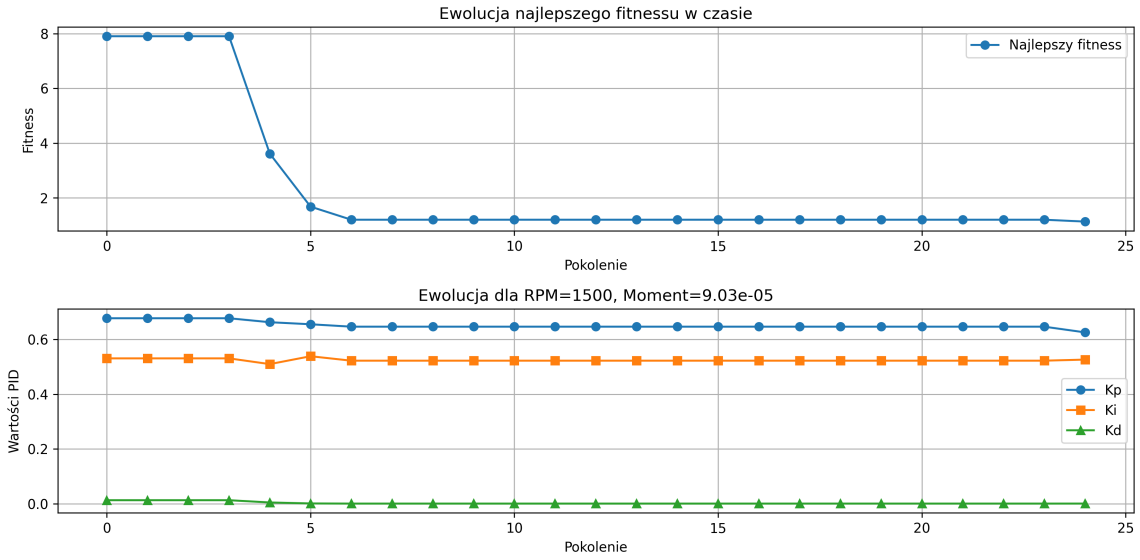
odchyleniem. Wyraźnie widać, że parametry PID są znacznie lepiej dopasowane, co przekłada się na wyższą precyzję.



Rysunek 5.14: Wykres prędkości przy zastosowaniu parametrów PID uzyskanych w ostatniej generacji AG

W ostatniej generacji, układ osiąga najwyższą stabilność w porównaniu do wcześniejszych etapów ewolucji. Prędkość silnika szybko zbliża się do wartości zadanej, czas ustalania dalej wynosi około 0,2 sekundy. Brak widocznych oscylacji po osiągnięciu wartości zadanej wskazuje na skuteczne dobranie parametrów regulatora PID.

System wykazuje wyjątkowo dobrą zgodność z wartością zadaną, co oznacza, że odchylenie prędkości w czasie całej symulacji jest praktycznie zerowe. Wynik ten sugeruje, że ewolucja algorytmu genetycznego skutecznie doprowadziła do znalezienia optymalnego zestawu parametrów PID, eliminując zarówno przeregulowanie, jak i nadmierne oscylacje.



Rysunek 5.15: Wykres przedstawiający ewolucję parametrów i ich wpływ na funkcję oceny

Górny wykres na rysunku 5.15 przedstawia ewolucję najlepszego fitnessu w czasie dla najlepszego osobnika w kolejnych generacjach. Na początku widoczna jest stosunkowo wysoka wartość

fitnessu, wynosząca około 8. Mimo że oznacza to, że początkowe parametry PID powodowały przeregulowanie, wydłużony czas ustalania oraz kary za oscylacje i błędy, wynik ten nie jest aż tak zły jak na całkowicie losowo dobrane parametry. Początkowa wartość fitness mogła być znacznie wyższa, co świadczy o umiarkowanie korzystnym zestawie startowym.

W pierwszych pięciu pokoleniach można zaobserwować dynamiczną poprawę fitnessu, co wskazuje na efektywne działanie algorytmu genetycznego w kierunku minimalizacji błędów i dostosowania parametrów PID. Po około szóstej generacji wartość fitness stabilizuje się na poziomie bliskim 0, co oznacza, że algorytm osiągnął optymalne parametry. Dzięki temu udało się zminimalizować czas ustalania, przeregulowanie oraz oscylacje, a kolejne generacje nie przynosiły już znaczących ulepszeń.

Dolny wykres ilustruje ewolucję wartości współczynników K_p, K_i i K_d w czasie. Parametr K_p początkowo ulegał delikatnym wahaniom, aby po kilku generacjach ustabilizować się w okolicach 0.6. Taka wartość zapewnia szybką reakcję układu na zmiany błędu. Wartość K_i ustabilizowała się na poziomie około 0.45, co wskazuje na odpowiednią korektę długoterminowych uchybów. Z kolei wartość K_d pozostała bardzo niska, bliska 0, co sugeruje, że tłumienie oscylacji odbywa się głównie dzięki dobrze dobranym wartościom K_p i K_i, bez potrzeby większego udziału członu różniczkującego.

Wnioski płynące z obu wykresów wskazują, że algorytm genetyczny skutecznie zoptymalizował parametry PID już w ciągu pierwszych 5–6 generacji. Po tym etapie ewolucja osiągnęła zbliżenie do globalnego minimum funkcji oceny, co wyjaśnia brak istotnych zmian w kolejnych pokoleniach.

5.2 Sztuczne sieci neuronowe dobierające nastawy PID

Do przeprowadzenia uczenia stworzono powiększony zbiór przypadków testowych dla algorytmu genetycznego. W ten sposób uzyskano 418 wyników symulacji które będą wystarczającą liczbą aby SSN potrafiła odpowiednio dopasowywać nastawy członów regulatora PID do danych przypadków. W celu automatycznego wyznaczania nastaw regulatora PID w zależności od parametrów wejściowych, wykorzystano sieć neuronową typu MLP.

5.2.1 Callback

Program zaczyna się od przygotowania callbacku, czyli specjalnego mechanizmu w bibliotece Keras, pozwalającemu wywoływać dodatkowe operacje w trakcie trenowania modelu.

```
17  class GradientPlotCallback(tf.keras.callbacks.Callback):  1 usage
18
19      def __init__(self, my_model, X_sample, Y_sample, outdir="plots/gradients"):
20          super().__init__()
21          self.my_model = my_model
22          self.X_sample = tf.constant(X_sample, dtype=tf.float32)
23          self.Y_sample = tf.constant(Y_sample, dtype=tf.float32)
24          self.outdir = outdir
25          os.makedirs(self.outdir, exist_ok=True)
26
27      def on_epoch_end(self, epoch, logs=None):
28          # co 5 epok + epoka 0
29          if epoch == 0 or (epoch % 5 == 0):
30              self.plot_gradient_for_epoch(epoch)
31
32      def plot_gradient_for_epoch(self, epoch):  1 usage
33          with tf.GradientTape() as tape:
34              preds = self.my_model(self.X_sample, training=True)
35              loss_value = self.my_model.compiled_loss(
36                  self.Y_sample,
37                  preds,
38                  regularization_losses=self.my_model.losses
39              )
```

Rysunek 5.16: Callback do uzyskiwania gradientów

Przedstawiona wyżej klasa jest odpowiedzialna za obliczanie i wizualizację parametrów gradientów w trakcie trenowania sieci neuronowej w środowisku Keras. Uzyskiwane parametry to kernele i biasy:

- **Kernels (Wagi)** - są to macierze wartości o wielowymiarowe strukturze, reprezentują współczynniki wag w sieci neuronowej. Odpowiadają za modelowanie wpływu wejść na wyjście poprzez obliczanie liniowej kombinacji cech wejściowych. Definiują one zależności pomiędzy warstwami sieci neuronowej, co umożliwia odwzorowanie złożonych relacji między danymi wejściowymi a wyjściowymi.
- **Biases (Przesunięcia)** - wektory wartości jednowymiarowe, dodawane do wyniku obliczonej liniowej kombinacji wag i wejść. Biases przesuwają wynik tej kombinacji, co pozwala funkcji aktywacji operować w szerszym zakresie wartości. Dzięki temu sieć neuronowa zyskuje większą elastyczność w aproksymacji nieliniowych zależności między danymi wejściowymi a wyjściowymi.

Oba te parametry są kluczowe dla optymalizacji sieci w procesie uczenia, ponieważ umożliwiają dokładne dopasowanie modelu do zbioru danych, minimalizując funkcję straty.

Klasa umożliwia cykliczną (w tym przypadku co 5 epok zaczynając od zerowej) analizę wartości gradientów względem parametrów modelu. Na podstawie intensywności aktualizacji w poszczególnych warstwach (normy L2 gradientu- miara wielkości gradientu w matematycznej przestrzeni) można ocenić, czy w danej warstwie zachodzi efektywne uczenie lub czy gradienty nie ulegają zanikaniu bądź eksplodowaniu. Funkcja plot_gradient_for_epoch realizuje rysowanie wykresów słupkowych przedstawiających normy L2, co ułatwia inspekcję i ewentualne dostrajanie hiperparametrów (learning rate, regularizację, itp.).

5.2.2 Wczytanie danych z pliku json

```
116  def load_data_from_json(json_filename):  1 usage
117
118      with open(json_filename, "r") as f:
119          data = json.load(f)
120
121      X_list = []
122      Y_list = []
123      W_list = []
124
125      for key, val in data.items():
126          rpm = val.get("rpm", None)
127          mom = val.get("moment_bezwladnosci", None)
128          kp = val.get("kp", None)
129          ki = val.get("ki", None)
130          kd = val.get("kd", None)
131          fit = val.get("fitness", None)
132
133          if None in [rpm, mom, kp, ki, kd, fit]:
134              continue
135
136          X_list.append([rpm, mom])
137          Y_list.append([kp, ki, kd])
138
139          w_raw = 1.0 / (fit + 1.0)
140          w = min(w_raw, 50.0)
141          W_list.append(w)
142
143      X_array = np.array(X_list, dtype=np.float32)
144      Y_array = np.array(Y_list, dtype=np.float32)
145      W_array = np.array(W_list, dtype=np.float32)
146
147      return X_array, Y_array, W_array
```

Rysunek 5.17: Wczytanie danych z pliku JSON

Funkcja load_data_from_json służy do wczytywania danych z pliku JSON i przekształcania ich w zestaw danych gotowy do użycia w procesie uczenia modelu. Tworzone są trzy listy:

- `X_list` - przechowuje wejściowe dane modelu (wartości `rpm` i `moment_bezwladnosci`).
- `Y_list` - przechowuje wyjściowe dane modelu (parametry PID: `kp`, `ki`, `kd`).
- `W_list` - przechowuje wagi próbek.

Kod w linijkach 139-141 służy do wyliczenia wagi próbek na podstawie przekazywanej wartości fitness. Wyższą wagę uzyskują wartości z mniejszym `fitness`, co skłania SSN do skupienia się na próbkach z dokładniejszym wynikiem osiągniętym w przeprowadzanych sumulacjach w algorytmie genetycznym.

5.2.3 Model MLP

```python
def build_mlp_model(input_dim=2, hidden_units=128):  1 usage

    model = Sequential()
    model.add(Dense(hidden_units, activation='relu', input_shape=(input_dim,)))
    model.add(BatchNormalization())
    model.add(Dropout(0.2))

    model.add(Dense(hidden_units, activation='relu'))
    model.add(BatchNormalization())
    model.add(Dropout(0.2))

    model.add(Dense(hidden_units, activation='relu'))
    model.add(BatchNormalization())
    model.add(Dropout(0.2))

    model.add(Dense(3, activation='sigmoid'))

    model.compile(optimizer=Adam(learning_rate=0.0005), loss='mse')
    return model
```

Rysunek 5.18: Struktura modelu MLP

Funkcja `build_mlp_model` definiuje i zwraca wielowarstwowy perceptron zaprojektowany do przewidywania parametrów PID.

Parametr `input_dim` określa liczbę cech wejściowych (w tym przypadku 2 – `rpm` i `moment_bezwładności`), natomiast `hidden_units` ustala liczbę neuronów w każdej warstwie ukrytej. Można wyróżnić 3 warstwy ukryte, każda jest w pełni połączona z `hidden_units` i posiada funkcję aktywacji ReLU, która wprowadza nieliniowość, usuwając wartości ujemne. Dodatkowo w każdej znajduje się `BatchNormalization()` - normalizuje dane wyjściowe z poprzedniej warstwy, co stabilizuje proces uczenia i przyspiesza minimalizację funkcji kosztu. `Dropout(0.2)`: Wprowadza regularizację poprzez losowe wyłączanie 20% neuronów podczas uczenia, co zmniejsza ryzyko przeuczenia. Warstwa wyjściowa posiada 3 neuronami odpowiadające parametrom PID. Wyrażenie activation=’sigmoid’ ogranicza wartości wyjściowe do zakresu [0, 1], co jest zgodne z normalizowanymi parametrami regulatora. Użyto optymalizatora Adam (Adaptive Moment Estimation) z ustaloną prędkością uczenia (0.0005). Zastosowana funkcja straty MSE (Mean Squared Error) minimalizuje średni błąd kwadratowy między przewidywanymi a rzeczywistymi wartościami parametrów PID.

5.2.4 Normalizacja danych

```python
#Normalizacja X (rpm, moment) do [0,1]
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)

#Podział na train i test
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test, W_train, W_test = \
    train_test_split( *arrays: X_scaled, Y, W, test_size=0.2, random_state=42)
print(f"Trening na {len(X_train)} przykładach, test na {len(X_test)}.")

# 4. Budowa modelu (3 warstwy, 128 neurony, sigmoid)
model = build_mlp_model(input_dim=X_train.shape[1], hidden_units=128)
```

Rysunek 5.19: Normalizacja danych wejściowych

Wywołanie `MinMaxScaler()` powoduje liniowe skalowanie wartości wejściowych do zadanego zakresu (domyślnie jest to [0, 1]). Funkcja `fit_transform(X)` oblicza minimalną i maksymalną

wartość dla każdej cechy, a następnie przekształca dane. Normalizacja danych to bardzo ważny etap przygotowania próbek, bez tego mogą pojawić się problemy wynikające z różnej skali cech. Następnie zachodzi podział danych- 80% próbek przeznaczone jest na dane treningowe, pozostałem 20% na przypadki testowe. Wyrażenie `random_state=42` to ustalony stan losowości, aby podział był powtarzalny. W linijce 253 występuje zainicjowanie odpowiedniej architektury MLP.

5.2.5 Trening modelu

```
273    #Trening z wagami
274    history = model.fit(
275        X_train, Y_train,
276        sample_weight=W_train,
277        validation_split=0.2,
278        epochs=100,
279        batch_size=16,
280        verbose=1,
281        callbacks=[grad_plot_cb, mse_history_cb, early_stop, reduce_lr]
282    )
```

Rysunek 5.20: Trenowanie modelu

Występująca w 274 linii kodu funkcja `model.fit` pochodzi z biblioteki Keras i odpowiada za wykonanie procesu uczenia modelu na dostarczonych danych treningowych. Argumenty tej funkcji kontrolują sposób treningu, konfigurację danych, wagi próbek oraz callbacki:

- `X_train` to dane wejściowe, a `Y_train` to dane wyjściowe (docelowe wartości PID). Są to dane treningowe, na których model uczy się odwzorowania wejść na wyjścia.
- `sample_weight=W_train` wprowadza wagi próbek, które modyfikują wpływ poszczególnych przykładów na funkcję kosztu. Próbki o większej wadze mają większy wpływ na uczenie.
- `validation_split=0.2` automatycznie dzieli dane treningowe na 80% próbek do treningu i 20% do walidacji. Walidacja monitoruje, czy model poprawia się na danych niewidocznych w danej iteracji treningu.
- `epochs=100` określa liczbę pełnych przebiegów po danych treningowych (epok). W każdej epoce model aktualizuje wagi na podstawie całego zbioru treningowego.
- `batch_size=16` definiuje liczbę próbek przetwarzanych równocześnie w każdej iteracji. Ma wpływ na wydajność treningu i stabilność gradientów (małe `batch_size` zapewnia większą stabilność, ale wolniejsze działanie).
- `verbose=1` ustawia tryb wyświetlania logów treningu w konsoli. Wartość `1` oznacza szczegółowe informacje o przebiegu każdej epoki.
- `callbacks` to lista obiektów typu `callback`, które modyfikują lub monitorują proces uczenia. W tym kodzie zastosowano cztery callbacki:
  - `grad_plot_cb`: Tworzy wykresy norm L2 gradientów dla kerneli i biasów co 5 epok.
  - `mse_history_cb`: Zapisuje wartości MSE (błędu średniokwadratowego) dla każdej epoki do pliku JSON.
  - `early_stop`: Monitoruje `val_loss` i zatrzymuje uczenie, gdy brak poprawy przez 10 kolejnych epok.

– `reduce_lr`: Redukuje współczynnik uczenia (`learning rate`) o połowę, gdy `val_loss` nie poprawia się przez 5 kolejnych epok.

```
287    #Ocena na zbiorze testowym
288    test_loss = model.evaluate(X_test, Y_test, sample_weight=W_test)
289    print(f"Test MSE = {test_loss:.4f}")
```

Rysunek 5.21: Ocena modelu na zbiorze testowym

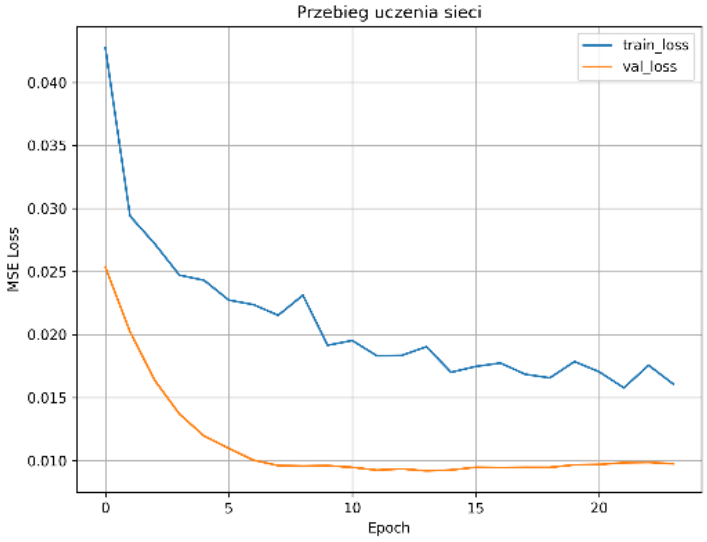
Wyrażenie `model.evaluate` sprawdza wydajność modelu na nowych danych. `X_test` i `Y_test` to dane wejściowe i wyjściowe testowe, które umożliwiają ocenę przewidywań modelu w porównaniu do rzeczywistych wartości. Kod pozwala ocenić, jak dobrze model przewiduje wartości PID dla nowych danych.

5.2.6 Działanie kodu i otrzymane wyniki

```
17/17 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 0.0176 - val_loss: 0.0099 - learning_rate: 1.2500e-04
Epoch 24/100
17/17 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 0.0145 - val_loss: 0.0098 - learning_rate: 1.2500e-04
[MSEHistoryCallback] Zapisano historię MSE do: mse_history.json
Zapisano wykres historii uczenia: plots\training_loss.png
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.0078
Test MSE = 0.0074
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
Zapisano wykres predykcji train: plots\pred_scatter_train.png
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step
Zapisano wykres predykcji test: plots\pred_scatter_test.png
```

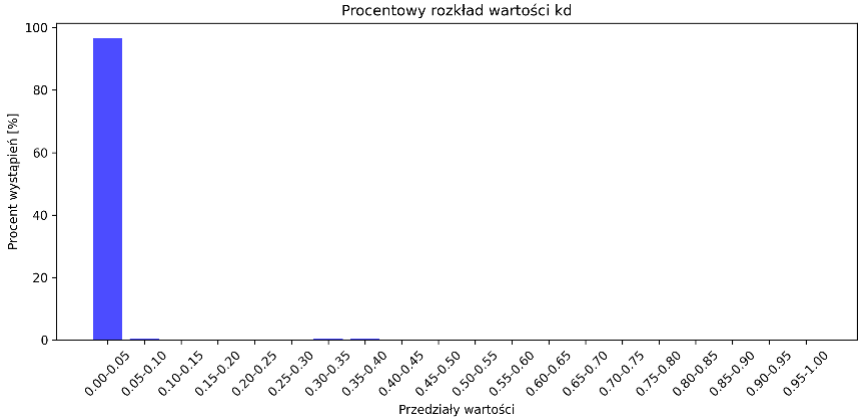
Rysunek 5.22: Zatrzymanie uczenia po 25 epokach

Uczenie zostało przerwane po 25 epokach, mimo ustawienia limitu na 100 epok, co widać na rysunku 5.22. Stało się tak dzięki zastosowaniu mechanizmu wczesnego zatrzymania (`EarlyStopping`), który monitoruje stratę walidacyjną (`val_loss`) i przerywa trening, gdy przez ustaloną liczbę epok (w tym przypadku 10) brak jest zauważalnej poprawy. W rezultacie, model unika przeuczenia (overfitting), oszczędzając czas obliczeniowy.

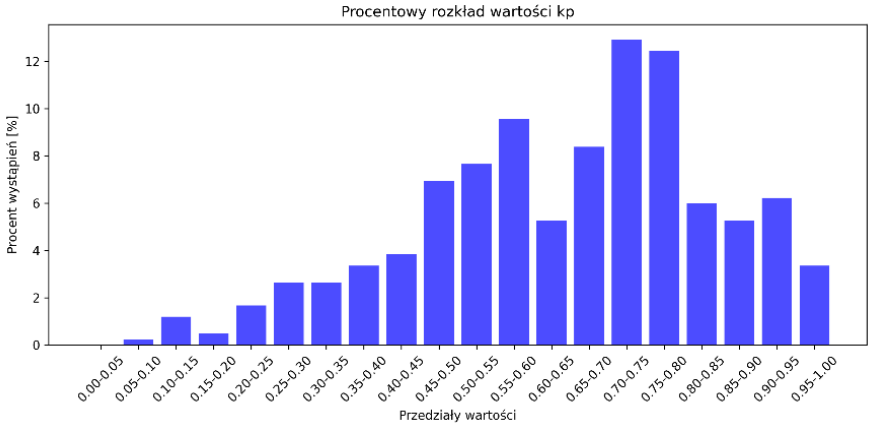


Rysunek 5.23: Wykresy uczenia

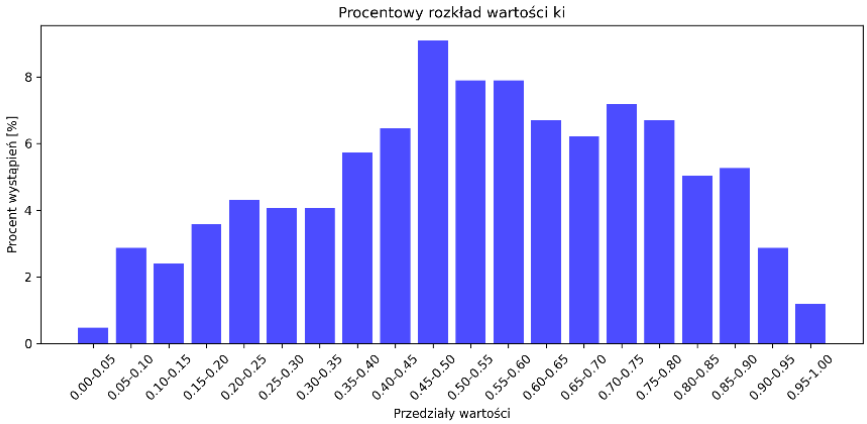
Wykres strat przedstawiony na rysunku 5.23 (`loss` i `val_loss`) pokazuje szybki spadek wartości na początku treningu, a następnie stabilizację. To sugeruje, że model osiągnął punkt, w którym dalsze uczenie nie przynosi znaczącej poprawy. Aby lepiej zrozumieć uzyskane wyniki wykonano histogramy parametrów członów PID które przedstawiono na rysunkach 5.24 - 5.26.



Rysunek 5.24: Rozkład wartości członu Kd w dostarczonych danych



Rysunek 5.25: Rozkład wartości członu Kp w dostarczonych danych

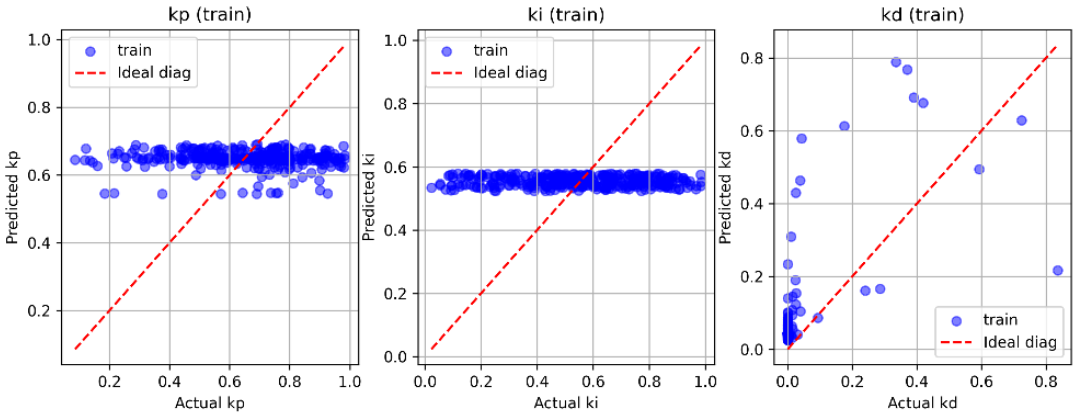


Rysunek 5.26: Rozkład wartości członu Ki w dostarczonych danych

Analiza histogramów parametrów PID występujących w dostarczonych danych wskazuje na:

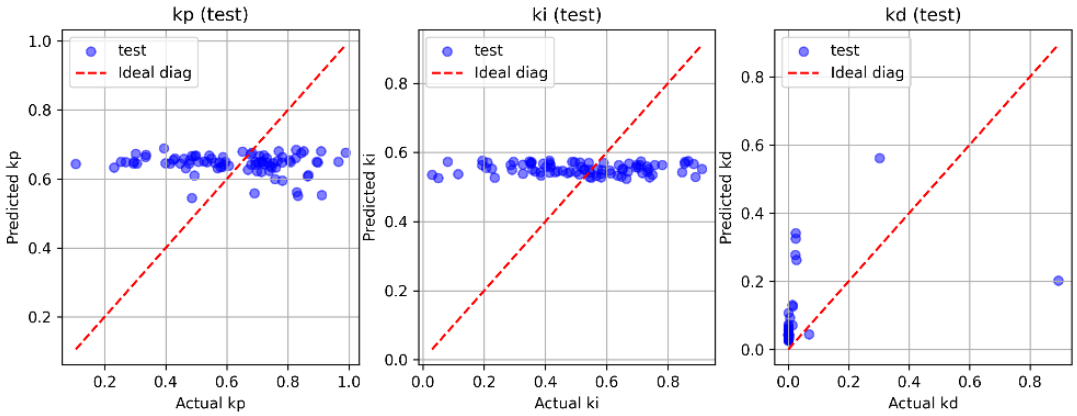
- Kp: najwięcej wartości skupia się w przedziałach 0.7 –0.75, co oznacza, że większość danych sugeruje wysoką proporcjonalność w sterowaniu.
- Ki: rozkład jest bardziej równomierny, jednak widoczne jest przesunięcie środka ciężkości w kierunku 0.45–0.6. To może sugerować umiarkowaną rolę członu całkującego w sterowaniu.
- Kd: rozkład jest wysoce skumulowany w przedziale 0–0.05. To wskazuje na minimalne wykorzystanie członu różniczkującego w danych, co utrudni modelowi dokładne odwzorowanie dla większych wartości Kd.

Uzyskane wyniki na zbiorze treningowym znacznie odbiegają od oczekiwanych. Przedstawiono je na rysunku 5.27.



Rysunek 5.27: Wyniki danych treningowych

W zbiorze treningowym model przewiduje wartości w wąskim zakresie dla wszystkich współczynników PID. Świadczy to o tym, że model był w stanie dopasować się do dominujących wzorców w danych uczących. Jednak wąskie pasma przewidywań mogą wskazywać na brak wystarczającej elastyczności modelu, co jest problemem, gdy model ma być stosowany na bardziej zróżnicowanych danych wejściowych. Zdecydowanie zabrakło ilości danych ze skrajnymi wartościami członów PID, co zmusiłoby model do opuszczenia bezpiecznego zakresu wartości. Ewaluacja na zbiorze testowym również wykazała wady trenowanego modelu. Uzyskany rozkład nastaw PID przedstawia rysunek 5.28.

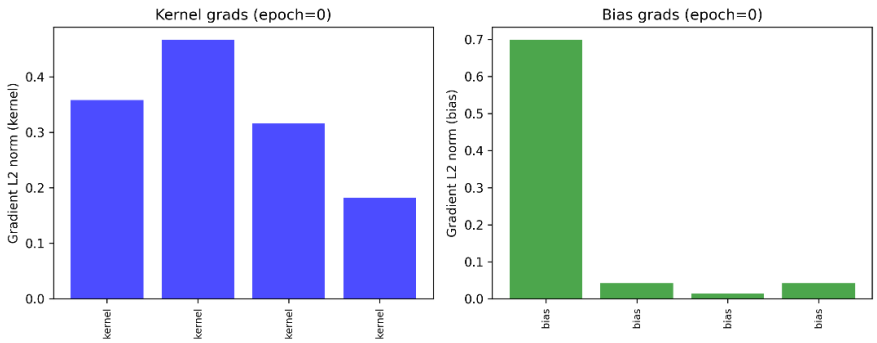


Rysunek 5.28: Wyniki danych testowych

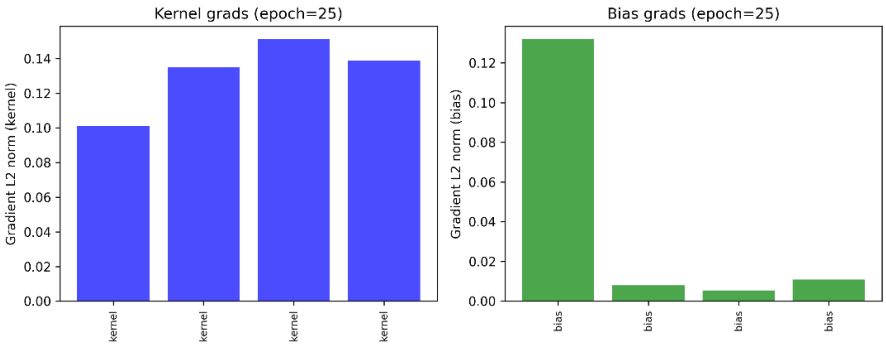
Wyniki na zbiorze testowym: Wyniki na zbiorze testowym:

- Człon proporcjonalny**:**
  - Model przewiduje wartości w wąskim zakresie, głównie między 0.6 a 0.8, zarówno na zbiorze testowym, jak i treningowym. Jest to zgodne z rozkładem danych uczących, gdzie większość przykładów również mieści się w tym przedziale.
  - Problemem jest jednak brak elastyczności modelu w przewidywaniu wartości Kp spoza tego wąskiego pasma. Oznacza to, że model uczył się głównie dopasowywania do dominującego wzorca w danych uczących, ale nie generalizuje dobrze dla przykładów o mniej reprezentowanych wartościach.
  - Wykres testowy pokazuje, że dla wartości rzeczywistych Kp poza tym zakresem model nie dostarcza dokładnych przewidywań, co może wynikać z niedostatecznej reprezentacji takich danych w zbiorze uczącym.
- Człon całkujący**:**
  - Rozkład rzeczywistych wartości Ki w danych wejściowych jest bardziej równomierny w porównaniu do Kp i Kd, co mogło ułatwić modelowi naukę odwzorowania tej zależności.
  - Pomimo tego przewidywania na zbiorze testowym nadal wykazują wąskie pasmo wyników, co wskazuje na ogólną tendencję modelu do ograniczania prognoz do bezpiecznych wartości.
- Człon różniczkujący:
  - Model wyraźnie niedoszacowuje wartości K_d, przewidując głównie bardzo małe wartości, co widać na wykresie testowym, gdzie większość punktów znajduje się blisko osi X (przewidywane $K_d \approx 0$).
  - Ten problem jest spowodowany nierównomiernym rozkładem danych uczących - jak pokazuje histogram, większość wartości K_d w danych wejściowych to bardzo małe liczby. Model nauczył się więc wzorca dominującego w danych, ale nie jest w stanie skutecznie przewidywać większych wartości, które występują rzadko.
  - Niedostateczne reprezentowanie wartości K_d w danych uczących spowodowało, że model utkwił w lokalnym minimum optymalizacji, które faworyzuje przewidywanie niskich wartości.

Zastosowane callbacki do uzyskania wartości gradientów mogą pomóc w interpretacji jak wyglądał proces uczenia sieci. Uzyskane wartości biasów i kerneli zaprezentowano na rysunkach 5.29 - 5.30.



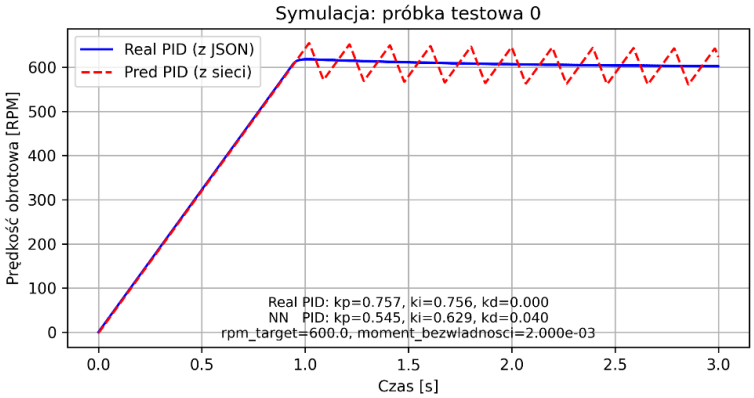
Rysunek 5.29: Wartości gradientów w epoce 0



Rysunek 5.30: Wartości gradientów w epoce 25

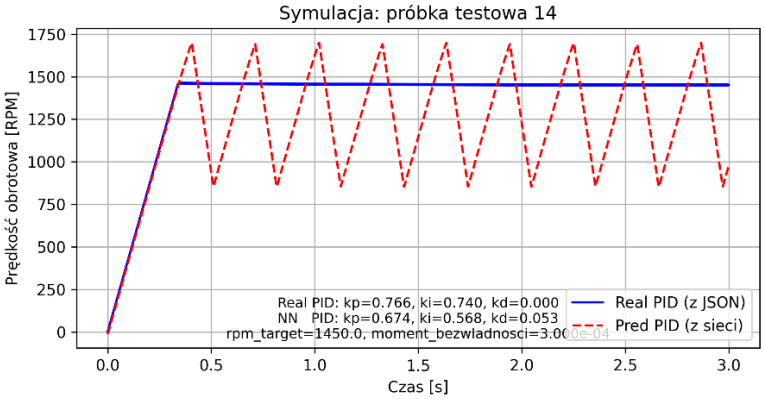
Na początku treningu normy gradientów dla kerneli i biasów były stosunkowo duże, co wskazuje na znaczną aktualizację wag w początkowych epokach. W epoce 25 gradienty stały się mniejsze, co wskazuje na spadek tempa nauki i osiągnięcie stabilizacji. Stabilizacja gradientów jest pożądana, ale jednocześnie może sugerować, że model nie odkrywa już nowych wzorców w danych.

Po przeanalizowaniu wyników na zbiorze testowym widać że bezpieczny próg który wybrał SSN i tak powoduje w większości przypadków oscylacje, tak jak na rysunku 5.31.



Rysunek 5.31: Niewielkie oscylacje przy wygenerowanych przez SSN nastawach PID

W większości uzyskanych wyników występują jednak znacznie większe oscylacje które dyskwalifikują badany model jako narzędzie do generowania poprawionych nastaw członów PID.



Rysunek 5.32: Znaczne oscylacje przy wygenerowanych przez SSN nastawach PID

5.3 Drzewo regresyjne

Do zadania dostosowania parametrów członów PID lepiej od standardowego drzewa decyzyjnego nadaje się drzewo regresyjne. Sprawdzenie działania tego prostego algorytmu uczenia maszynowego zostanie wykonane przy pomocy pliku results.json utworzonego w czasie symulacji 418 przypadków testowych w algorytmie genetycznym.

5.3.1 Wczytanie i podział danych

```
21  with open(json_filename, "r") as f:
22      data = json.load(f)
23
24  X_list, Y_list = [], []
25
26  for val in data.values():
27      rpm = val.get("rpm")
28      moment = val.get("moment_bezwladnosci")
29      kp = val.get("kp")
30      ki = val.get("ki")
31      kd = val.get("kd")
32
33      if None in [rpm, moment, kp, ki, kd]:
34          continue
35
36      X_list.append([rpm, moment])
37      Y_list.append([kp, ki, kd])
38
39  X_array = np.array(X_list, dtype=np.float32)
40  Y_array = np.array(Y_list, dtype=np.float32)
```

Rysunek 5.33: Pobranie danych z pliku json

Wczytanie danych odbywa się tak samo jak w przypadku kodu użytego w SSN generującego nastawy PID. Jedyną różnicą jest brak uzyskania zmiennej fitness która nie wniosłaby wartości do działania drzewa regresyjnego.

```
52  X_train, X_test, Y_train, Y_test = train_test_split( *arrays: X, Y, test_size=0.2, random_state=1)
```

Rysunek 5.34: Podział danych na treningowe i testowe

Uzyskane dane dzielimy na treningowe (80% zbioru) i testowe (20%).

5.3.2 Ewaluacja

```
54  # Funkcja do ewaluacji modelu z różnymi max_depth
55  def eval_model_with_max_depths(X, y):  1 usage
56      max_depths = range(1, 11)  # Testujemy głębokości od 1 do 10
57      mean_mse = []
58
59      for max_depth in max_depths:
60          model = DecisionTreeRegressor(max_depth=max_depth, random_state=3)
61          cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=3)
62          scores = cross_val_score(model, X, y, scoring='neg_mean_squared_error', cv=cv, n_jobs=-1)
63          mean_mse.append(-scores.mean())  # Przekształcamy na dodatnie MSE
64
65      return max_depths, mean_mse


66
67  # Ewaluacja modelu
68  max_depths, mean_mse = eval_model_with_max_depths(X, Y)
```

Rysunek 5.35: Ewaluacja modelu

Ewaluacja w kontekście uczenia maszynowego to proces oceny wydajności modelu. Polega na zastosowaniu modelu na nowych danych (zbiorze testowym), które nie były używane podczas treningu, w celu określenia, jak dobrze model przewiduje wyniki.

Zbyt duża głębokość drzewa prowadzi do przeuczenia, natomiast zbyt mała do niedouczenia. Optymalna głębokość minimalizuje błąd MSE, co zapewnia lepszą zdolność generalizacji. Aby wybrać najlepszą złożoność drzewa, tworzonych jest kilka modeli (w tym przypadku zmienna max_depths jest w zakresie [1:11] co wskazuje na tworzenie drzew od 1 do 10 poziomów). Linijki 60-61 odpowiadają za przeprowadzanie kroswalidacji (`RepeatedKFold`): dane są dzielone na 10 podzbiorów (`n_splits=10`), a proces jest powtarzany 3 razy (`n_repeats=3`) z różnymi podziałami. Taka forma podziału zapewnia bardziej wiarygodną ocenę modelu przez testowanie na różnych podzbiorach, znacząco zmniejsza ryzyko oceny na nietypowym zestawie danych. Wyniki kroswalidacji (`scores`) to lista błędów MSE dla każdego podziału.

5.3.3 Optymalna głębokość drzewa decyzyjnego

```
81  # Wybór optymalnej głębokości drzewa
82  best_max_depth = max_depths[np.argmin(mean_mse)]
83  print(f"Najlepsza głębokość drzewa: {best_max_depth}")
84
85  # Trening drzewa regresyjnego
86  regressor = DecisionTreeRegressor(max_depth=best_max_depth, random_state=3)
87  regressor.fit(X_train, Y_train)
```

Rysunek 5.36: Wybór optymalnej głębokości drzewa i trening

Funkcja NumPy `np.argmin(mean_mse)` zwraca indeks najmniejszej wartości w liście średnich wartości MSE. Na tej podstawie ostatecznie wybierana jest najlepsza głębokość drzewa.

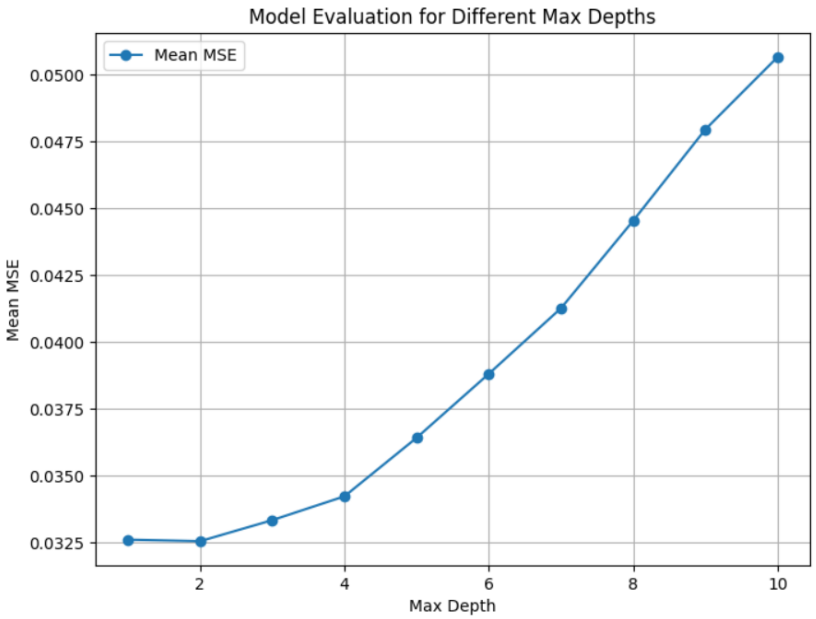
Klasa `DecisionTreeRegressor` z biblioteki scikit-learn implementuje drzewo decyzyjne do regresji. Jej argument `random_state=3` to ziarno losowości, które zapewnia powtarzalność wyników podczas tworzenia drzewa.

```
92  # Eksport drzewa do Graphviz
93  dot_data = export_graphviz(
94      regressor,
95      out_file=None,
96      feature_names=["RPM", "Moment Bezwładności"],
97      filled=True,
98      rounded=True,
99      special_characters=True
100 )
101
102 # Rysowanie drzewa i zapisywanie do pliku
103 graph = graphviz.Source(dot_data)
104 output_filename = "decision_tree"
105 graph.render(output_filename, format='pdf', cleanup=True)  # Zapisz jako PDF
```

Rysunek 5.37: Kod do przedstawienia graficznego schematu drzewa regresyjnego

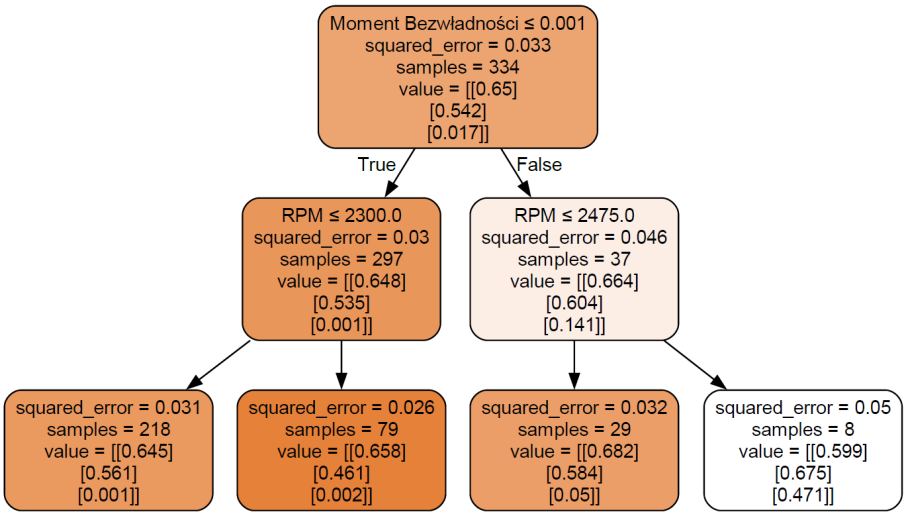
Ten fragment kodu odpowiada za generowanie wizualizacji drzewa decyzyjnego i zapisanie go w formacie PDF. Funkcja `export_graphviz` generuje opis drzewa decyzyjnego w formacie DOT (język używany przez pakiet Graphviz). `graphviz.Source(dot_data)` tworzy obiekt grafu z opisu drzewa w formacie DOT, przechowywanego w zmiennej `dot_data`. Funkcja `graph.render` zapisuje wizualizację drzewa do pliku w wybranym formacie.

5.3.4 Otrzymane Wyniki



Rysunek 5.38: Wykres wartości MSE dla różnych głębokości

Rysunek 5.38 ilustruje wykres średniego błędu kwadratowego w zależności od maksymalnej głębokości drzewa regresyjnego. Najniższy MSE (około 0.0325) wskazuje, że drzewo o głębokości równej 2 jest najlepiej dopasowane do danych. Wraz ze wzrostem głębokości drzewa (od 3 do 10) MSE znacząco rośnie, co oznacza, że model traci zdolność do generalizacji lub nie potrafi lepiej rozdzielić danych.



Rysunek 5.39: Schemat otrzymanego drzewa

Otrzymane drzewo regresyjne przewiduje wartości nastaw PID na podstawie dwóch cech wejściowych: moment bezwładności i RPM. Każdy węzeł drzewa zawiera informacje o kryterium podziału, liczbie próbek, wartości średniej (dla każdego członu PID) i błędzie kwadratowym. W korzeniu drzewa zastosowano kryterium podziału `Moment Bezwładności` ≤ 0.001. Próbki są dzielone na dwie grupy w zależności od wartości cechy `Moment Bezwładności`.

- Lewy węzeł (True): Próbki, dla których `Moment Bezwładności <= 0.001`.
- Prawy węzeł (False): Próbki, dla których `Moment Bezwładności > 0.001`.

W lewej gałęzi podział odbywa się na podstawie kryterium `RPM` ≤ 2300.0:

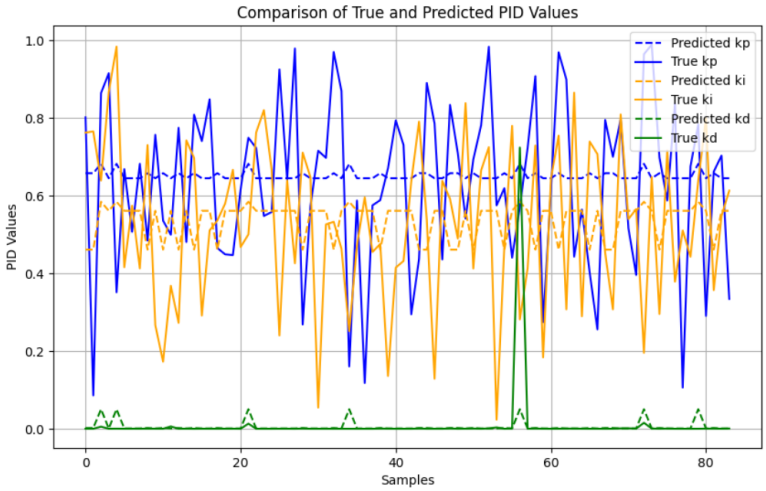
- Lewy węzeł:
  - `squared_error = 0.031`: Mniejszy błąd, co wskazuje na większą jednorodność próbek w tym podzbiorze.
  - `samples = 218`: Próbki w tej grupie.
  - `value = [[0.645], [0.561], [0.001]]`: Średnie wartości dla `kp`, `ki`, `kd`.
- Prawy węzeł (False):
  - `squared_error = 0.026`: Jeszcze mniejszy błąd.
  - `samples = 79`: Mniej próbek, ale większa jednorodność.
  - `value = [[0.658], [0.461], [0.002]]`.

Powracając do prawego górnego węzła po pierwszym podziale, kolejne zastosowane kryterium to `RPM` ≤ 2475.0

- Lewy węzeł:
  - `squared_error = 0.032`
  - `samples = 29`
  - `value = [[0.682], [0.584], [0.05]]`.

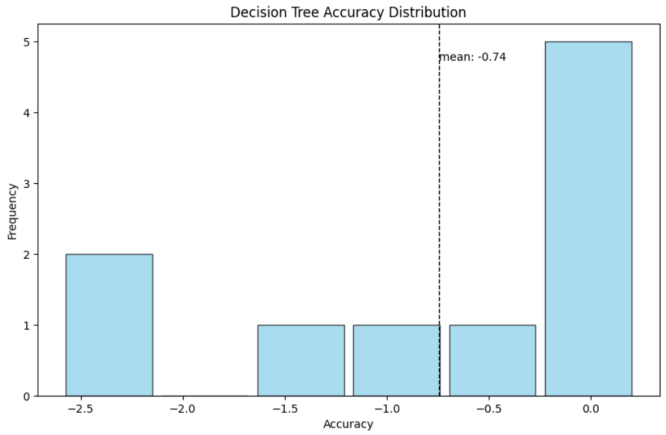
- Prawy węzeł:
  - `squared_error = 0.05`: Największy błąd w tej gałęzi, wskazujący na największą różnorodność próbek.
  - `samples = 8`: Bardzo mała liczba próbek.
  - `value = [[0.599], [0.675], [0.471]]`.

Wartości w `value` reprezentują średnie wartości `kp`, `ki`, `kd` w każdej grupie. Drzewo jest stosunkowo płytkie (`max_depth = 2`), co pozwala uniknąć przeuczenia, ale ogranicza zdolność modelu do uchwycenia złożonych zależności. Na podstawie powyższego schematu możnaby wywnioskować że drzewo jest w stanie poradzić sobie z większą ilością przypadków testowych znajdujących się w danych zakresie, za to nie radzi sobie z mniejszą ilością próbek wliczających się w rzadko uzyskiwane różnorodne wartości (ostatni podział po prawej stronie, 8 próbek i najwyższy wynik MSE).



Rysunek 5.40: Porównanie uzyskanych wartości z prawdziwymi

Niestety wykres uzyskanych wartości dla danych testowych nie pozostawia złudzeń że algorytm nie jest w stanie odpowiednio dobrać nastaw członów PID. Model operuje na bardzo ograniczonym przedziale wartości, jedynie dla wartości `ki` widoczna jest próba poszerzenia zakresu doboru parametrów. Model zaadaptował się jako narzędzie do uzyskiwania średniej wartości członów PID zamiast spróbować przewidywać dokładnie nastawy. Prawdopodnie nie był w stanie odnaleźć wystarczająco zależności dla dostarczonych danych.



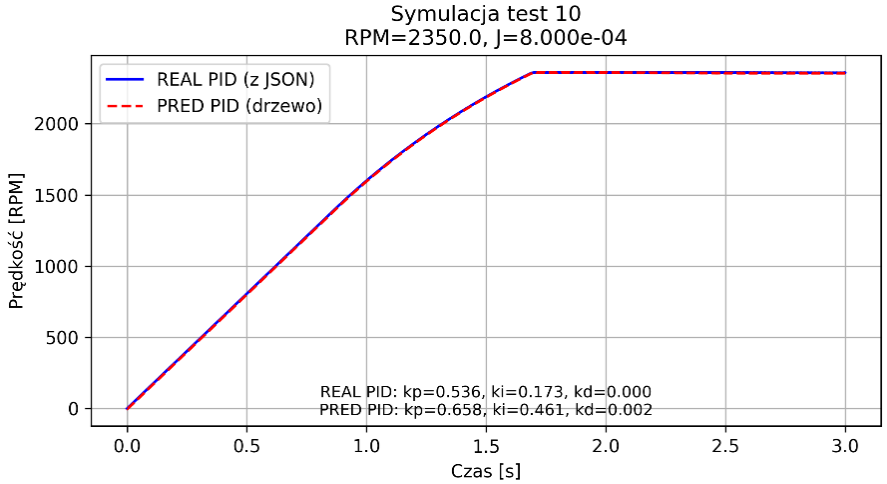
Rysunek 5.41: Uzyskana dokładność drzewa regresyjnego

Oś pozioma (Accuary) przedstawia wartości współczynnika $\mathbf{R^2}$ (współczynnik determinacji), który mierzy jakość dopasowania modelu, natomiast oś pionowa (Frequency) przedstawia liczbę wystąpień danego zakresu dokładności w ramach przeprowadzonych kroswalidacji. Zakres wartości R^2 należy interpretować w następujący sposób:

- $R^2 = 1$ - Idealne dopasowanie – model perfekcyjnie wyjaśnia zmienność danych.
- $R^2 = 0$ - Model nie wyjaśnia żadnej zmienności danych – przewidywane wartości są równe średniej wartości rzeczywistych danych.
- $R^2 < 0$ - Model działa gorzej niż prosta średnia wartości rzeczywistych. Może to oznaczać, że model jest niewłaściwie skonstruowany lub dane są zbyt złożone, by model mógł uchwycić zależności.

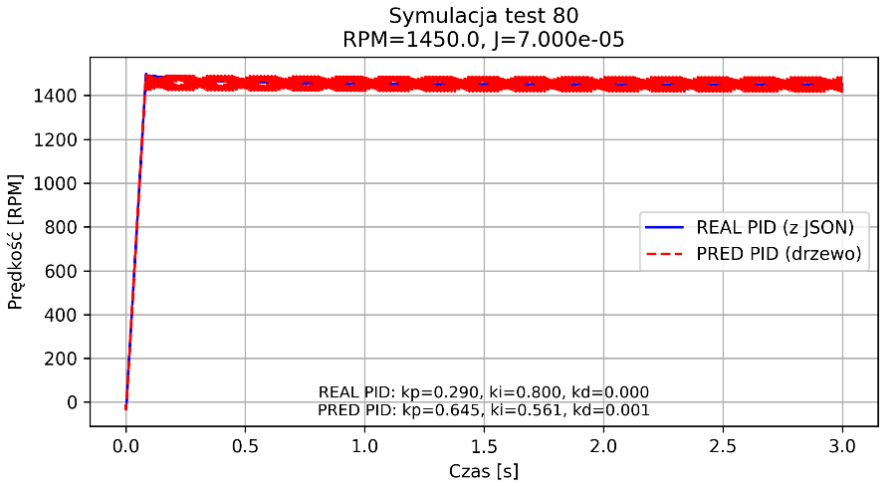
Średnia wartość $\mathbf{R^2}$ dla modelu wynosi około -0.74, jest to bardzo niski wynik, który wskazuje, że model nie radzi sobie dobrze z przewidywaniem wyników.

Otrzymane wyniki porównujące prędkości obrotowe przy danych testowych wejściowych i przedstawionych przez drzewo pokazują że wąskie wartości w których operuje model, mogą dobrze działać w przypadku danych położonych blisko kryteriów podziału drzewa. Widoczne jest to na rysunku 5.42.



Rysunek 5.42: Poprawne wygenerowanie wartości PID dla przykładu bliskiego warunku podziału.

Zdecydowanie gorzej model radzi sobie z przykładami testowymi położonymi parametrami dalej od kryterium podziału, w przebiegu zaczynają być widoczne oscylacje, pokazuje rysunek 5.43.



Rysunek 5.43: Pojawienie się oscylacji dla przykładu oddalonego parametrami od kryterium podziału.

Drzewo regresyjne jest prostym modelem, który dobrze działa w przypadku liniowych i mniej skomplikowanych zależności. Brak różnorodności w danych wejściowych (`RPM` i `Moment Bezwładności`) oraz ich słaba korelacja z wyjściami (`kp`, `ki`, `kd`) ogranicza zdolność modelu do nauki. Niewielka liczba próbek w niektórych podziałach (np. 8 próbek w węźle) dodatkowo pogłębia problem niedostatecznego dopasowania. Dodatkowym utrudnieniem jest binarny podział na węzły spełniające lub niespełniające dany warunek. Model nie jest w stanie wychwycić zależności w danych i przez to nie ma możliwości skutecznego rozbudowania architektury drzewa. Aby poprawić jakość predykcji, konieczne jest zastosowanie bardziej złożonych modeli, zwiększenie różnorodności danych lub optymalizacja hiperparametrów. Obecnie drzewo regresyjne działa głównie jako narzędzie do uśredniania wyników, co nie pozwala na precyzyjne przewidywanie nastaw PID, szczególnie gdyby miało dotyczyć nastaw wykraczających poza dostarczone dane w trakcie treningu.

5.4 SSN do generowania sekwencji napięć

Inną metodą sterowania silnikiem DC będzie sterowanie bezpośrednie przy pomocy generowania listy napięć dla danego przypadku testowego.

5.4.1 Wczytanie serii napięć

Kod posiada zdefiniowany callback do otrzymania wykresów gradientów tak jak zostało to przedstawione w SSN do generowania nastaw PID.


```
{
  "rpm": 500,
  "moment_bezwladnosci": 0.00011,
  "kp": 0.7560915295313526,
  "ki": 0.4304640903995647,
  "kd": 0.0007153215738058608,
  "fitness": 0.9373610344749519,
  "voltage_sequence": [
    20.600209907945725,
    24.0,
    24.0,
    24.0,
```


Rysunek 5.44: Dane zawarte w plikach voltage_data

W tym przypadku do uczenia algorytmu zostaną użyte pliki `voltage_data...` wygenerowane dla każdego z 418 przypadków testowanych w algorytmie genetycznym. Oprócz standardowych danych znajdujących się we wcześiej używanym `results.json` w każdym nowym pliku znajduje się sekwencja 3000 napięć które odpowiadają jednemu kroku czasowemu (0.001 sekundy) w symulacji modelu silnika.

```python
def load_voltage_data(folder="voltage_data"):  1 usage
    import glob, json
    file_list = glob.glob(os.path.join(folder, "*.json"))

    X_list = []
    Y_list = []
    W_list = []

    for fn in file_list:
        with open(fn, "r") as f:
            data = json.load(f)

        rpm = data.get("rpm", None)
        mom = data.get("moment_bezwladnosci", None)
        fit = data.get("fitness", None)
        seq = data.get("voltage_sequence", None)

        if None in [rpm, mom, fit, seq]:
            continue
        if len(seq) != 3000:
            print(f"UWAGA: Plik {fn} ma sekwencję {len(seq)} zamiast 3000. Pomijam.")
            continue

        X_list.append([rpm, mom])
        Y_list.append(seq)
        w_raw = 1.0 / (fit + 1.0)
        w = min(w_raw, 50.0)
        W_list.append(w)

    X_array = np.array(X_list, dtype=np.float32)
    Y_array = np.array(Y_list, dtype=np.float32)
    W_array = np.array(W_list, dtype=np.float32)
    return X_array, Y_array, W_array
```

Rysunek 5.45: Wczytanie danych

Funkcja `load_voltage_data` wczytuje dane z plików JSON znajdujących się w określonym folderze (`voltage_data`). Dane te są następnie przetwarzane i strukturyzowane w formie macierzy `X`, `Y`, i `W`, które reprezentują:

- `X` - dane wejściowe: prędkość obrotowa i moment bezwładności.
- `Y` - sekwencje napięć (`voltage_sequence`) o stałej długości (3000 próbek).
- `W` - wagi próbek, które są zależne od wartości funkcji `fitness` z danego pliku (obliczane są w taki sam sposób, jak w przypadku SSN przeznaczonego do przewidywania PID).

5.4.2 Model MLP

```python
def build_voltage_mlp(input_dim=2, seq_len=3000, hidden_units=64):  1 usage

    from tensorflow.keras import Sequential
    from tensorflow.keras.layers import Dense, Dropout

    model = Sequential()
    model.add(Dense(hidden_units, activation='relu', input_shape=(input_dim,)))
    model.add(Dropout(0.1))

    model.add(Dense(hidden_units, activation='relu'))
    model.add(Dropout(0.1))

    model.add(Dense(hidden_units, activation='relu'))
    model.add(Dropout(0.1))

    # Wyjście: 3000 floatów, bez ograniczenia
    model.add(Dense(seq_len, activation='linear'))


    from tensorflow.keras.optimizers import Adam
    model.compile(optimizer=Adam(learning_rate=0.001),
                  loss='Huber')
    return model
```

Rysunek 5.46: Struktura modelu MLP

Model MLP sieci neuronowej zbudowany jest z trzema w pełni połączonymi (dense) warstwami ukrytymi aktywowanymi funkcją ReLU (Rectified Linear Unit); mechanizmem Dropout, zapobiegającym nadmiernemu dopasowaniu; warstwą wyjściową o liczbie neuronów odpowiadającej długości sekwencji napięć (`seq_len`), aktywowaną funkcją liniową. Kompilacja wykorzystuje optymalizator Adam i funkcją straty Huber (jest kombinacją funkcji średniokwadratowej i średniej wartości bezwzględnej błędu (MAE), przez co jest odporna na wpływ wartości odstających).

5.4.3 Normalizacja, podział danych

```python
    #Normalizacja X
    from sklearn.preprocessing import MinMaxScaler
    scaler = MinMaxScaler()
    X_scaled = scaler.fit_transform(X)

    #Indeksy train/test
    from sklearn.model_selection import train_test_split
    all_indices = np.arange(N)
    train_idx, test_idx = train_test_split( *arrays: all_indices, test_size=0.2, random_state=42)

    X_train = X_scaled[train_idx]
    Y_train = Y[train_idx]
    W_train = W[train_idx]

    X_test = X_scaled[test_idx]
    Y_test = Y[test_idx]
    W_test = W[test_idx]

    # Maska
    is_train_mask = np.zeros(N, dtype=bool)
    is_train_mask[train_idx] = True
```

Rysunek 5.47: Normalizacja, podział danych na treningowe i testowe, stworzenie maski

Funkcja `MinMaxScaler` realizuje skalowanie danych wejściowych `X` na podstawie ich minimalnych i maksymalnych wartości w każdym wymiarze:

$$X_{\text{scaled}} = \frac{X - X_{\min}}{X_{\max} - X_{\min}}$$

gdzie $X_{\min}$ oznacza najmniejszą wartość, $X_{\max}$ największa, a X to bieżąca wartość.`fit_transform` dopasowuje skalę na podstawie danych wejściowych `X` i jednocześnie transformuje je.

Funkcja `train_test_split` rozdziela dane na zbiór treningowy (80% próbek) i testowy (20%) zgodnie z parametrem `test_size=0.2`.

Na końcu następuje utworzenie maski logicznej wskazującej, które próbki należą do zbioru treningowego:

- `np.zeros(N, dtype=bool)` - tworzy tablicę długości odpowiadającej ogólnej liczbie przykładów wypełnioną wartością `False`.
- `is_train_mask[train_idx] = True` - dla próbek należących do zbioru treningowego ustawia wartość `True`.

Maska jest wykorzystywana w wizualizacji wyników, aby rozróżnić próbki treningowe i testowe.

5.4.4 Trening i ewaluacja

```
257  #Trening
258  history = model.fit(
259      X_train, Y_train,
260      sample_weight=W_train,
261      validation_split=0.2,
262      epochs=300,
263      batch_size=8,
264      verbose=1,
265      callbacks=[grad_callback, early_stop, reduce_lr]
266  )
267
268  #Wykres przebiegu uczenia
269  plot_training_history(history, outdir="plots")
270
271  #Ewaluacja
272  test_loss = model.evaluate(X_test, Y_test, sample_weight=W_test)
273  print(f"[INFO] Test Huber = {test_loss:.4f}")
```

Rysunek 5.48: Trening i ewaluacja

Funkcja realizuje dopasowanie modelu do danych treningowych poprzez iteracyjne aktualizowanie wag sieci w taki sposób, aby minimalizować funkcję straty. Jej najważniejsze parametry to:

- `epochs=300` - liczba epok, czyli pełnych przejść przez cały zbiór treningowy.
- `batch_size=8` - liczba próbek w jednej partii (batchu) wykorzystywanej do aktualizacji wag.
- `callbacks=[grad_callback, early_stop, reduce_lr]` - lista callbacków, które wpływają na przebieg treningu:
  - `grad_callback` - wizualizuje gradienty podczas uczenia co 20 epok.
  - `early_stop` - zatrzymuje trening, jeśli wynik na zbiorze walidacyjnym nie poprawia się przez 40 epok.
  - `reduce_lr` - redukuje szybkość uczenia (learning rate), gdy strata walidacyjna przestaje maleć przez 15 epok.

Funkcja `evaluate` oblicza wartość funkcji straty (Huber loss) na zbiorze testowym, uwzględniając wagi próbek. Pozwala to na obiektywną ocenę zdolności modelu do generalizacji, czyli przewidywania wartości napięć na danych, których wcześniej nie widział.

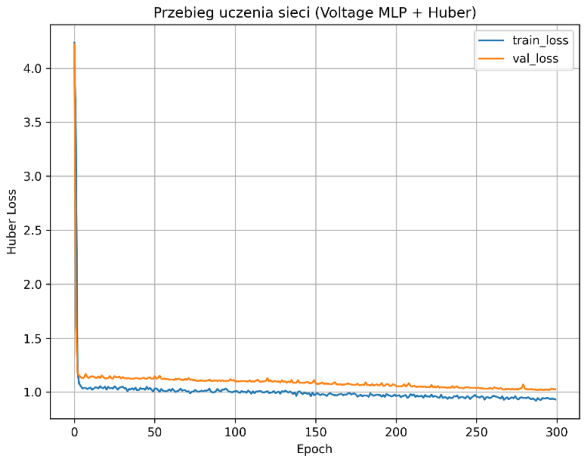
5.4.5 Zakres generowanych napięć

```
284    #OGRANICZENIE NAPIĘĆ DO [0,24]
285    Y_pred_train_clipped = np.clip(Y_pred_train, a_min: 0, a_max: 24)
286    Y_pred_test_clipped = np.clip(Y_pred_test, a_min: 0, a_max: 24)
```

Rysunek 5.49: Ustawienie zakresu generowanych napięć

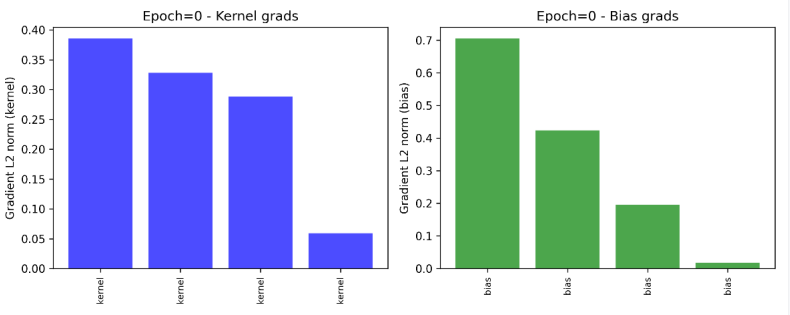
Po pierwszych testach zauważono skłonność modelu do generowania znacznie zawyżonych wartości napięć przy starcie silniki, metoda `clip` ogranicza te wartości w przekazywanym do niej zakresie.

5.4.6 Działanie kodu i wyniki

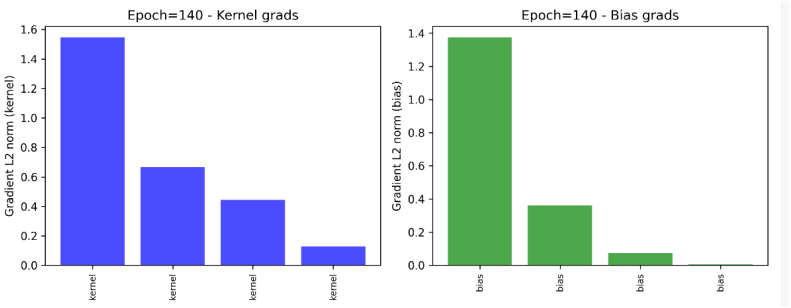


Rysunek 5.50: Wykres pokazujący przebieg uczenia

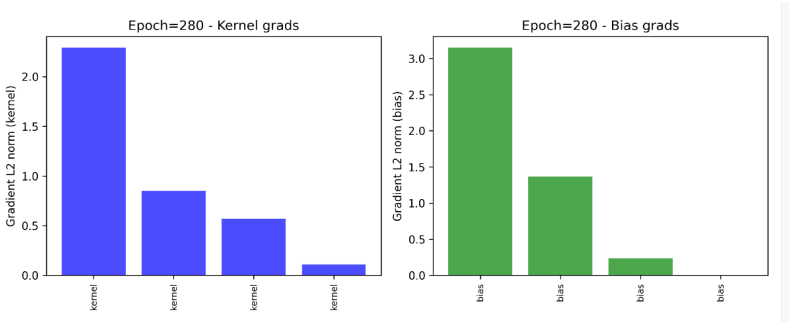
Rysunek 5.50 przedstawia wykres wartości funkcji straty w zależności od liczby epok podczas uczenia sieci MLP. `train_loss` reprezentuje stratę na zbiorze treningowym; `val_loss` oznacza stratę na zbiorze walidacyjnym. Wartość funkcji straty gwałtownie maleje w ciągu pierwszych 10 epok. To oznacza, że model szybko uczy się podstawowych wzorców w danych, co jest typowe dla sieci neuronowych, szczególnie na początku treningu. Niewielkie różnice między `train_loss` a `val_loss` wskazują na brak znaczącego nadmiernego dopasowania. Pomimo stabilności strat na poziomie około 1.0, wartość ta może wskazywać, że model nie osiągnął pełnej precyzji w przewidywaniu sekwencji napięć. Może to wynikać z ograniczeń zastosowanych danych lub architektury sieci.



Rysunek 5.51: Gradient w epoce 0



Rysunek 5.52: Gradient w 140 epoce

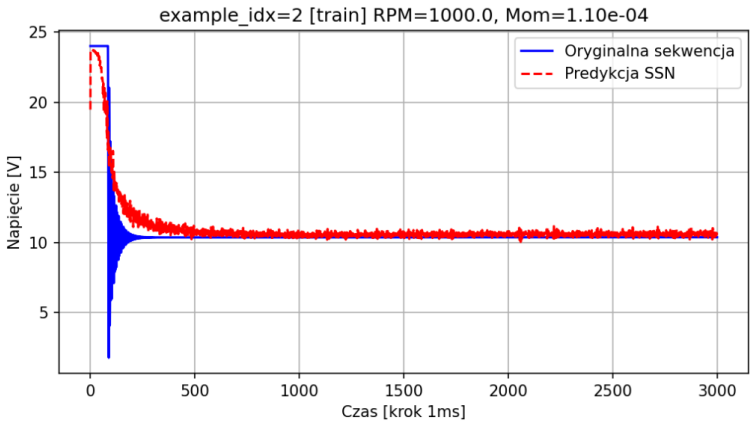


Rysunek 5.53: Gradient w 280 epoce

Wzrost wartości gradientów w późniejszych epokach (140 i 280) jest nietypowy. Zazwyczaj gradienty maleją wraz z konwergencją modelu. Może to świadczyć o:

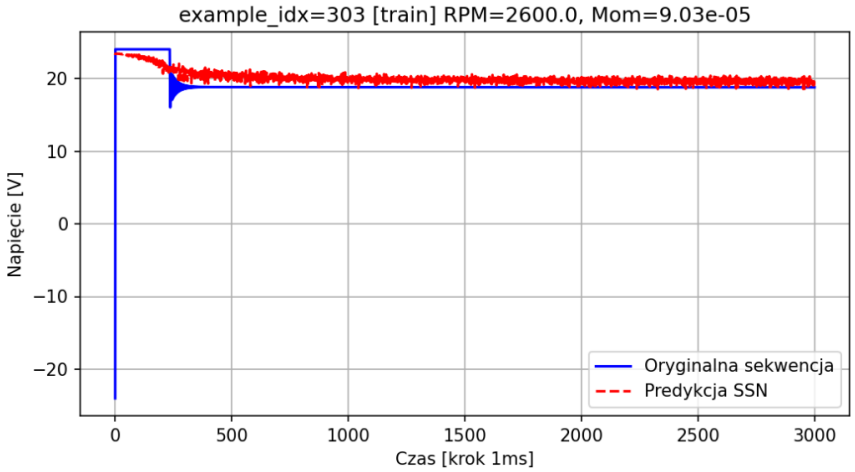
- Problemach ze stabilnością modelu.
- Nadmiernym dopasowaniem w dalszych epokach.
- Użyciu optymalizatora lub parametrów (takich jak np. learning rate), które spowodowały niestabilność.

Najwyższe gradienty są obserwowane w pierwszej warstwie. Świadczy to o tym, że ta warstwa dominuje w procesie nauki, co może być zrozumiałe, jeśli dane mają mocno skorelowane cechy. Głębsze warstwy również wykazują większe gradienty w dalszych epokach, co wskazuje na bardziej złożone dostosowywanie w modelu. Biasy mają większe gradienty niż kernele, co oznacza, że model stara się przede wszystkim korygować globalne przesunięcia w danych. Pomimo wzrostu gradientów udało się uzyskać bardzo zadowalające efekty:



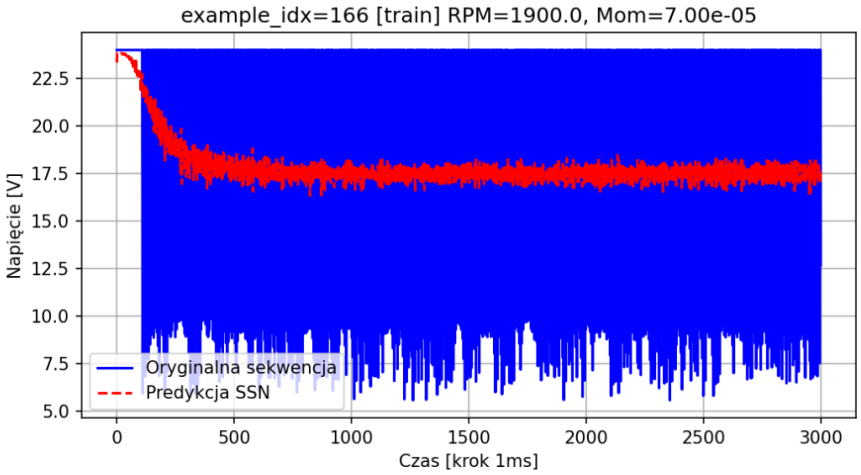
Rysunek 5.54: Uzyskany przebieg napięć dla przykładu z 1000 RPM

W prezentowanym przykładzie, początkowe napięcie dynamicznie spada z wartości około 25V do poziomu stabilizacji w okolicach 10V. Model nie odwzorowuje idealnie zmiany napięcia, generuje łagodniejszy przebieg. Po fazie przejściowej (około 500 kroków) napięcie stabilizuje się, a predykcja modelu jest bardzo zbliżona do sekwencji uzyskanej na modelu matematycznym.



Rysunek 5.55: Uzyskany przebieg napięć dla przykładu z 2600 RPM

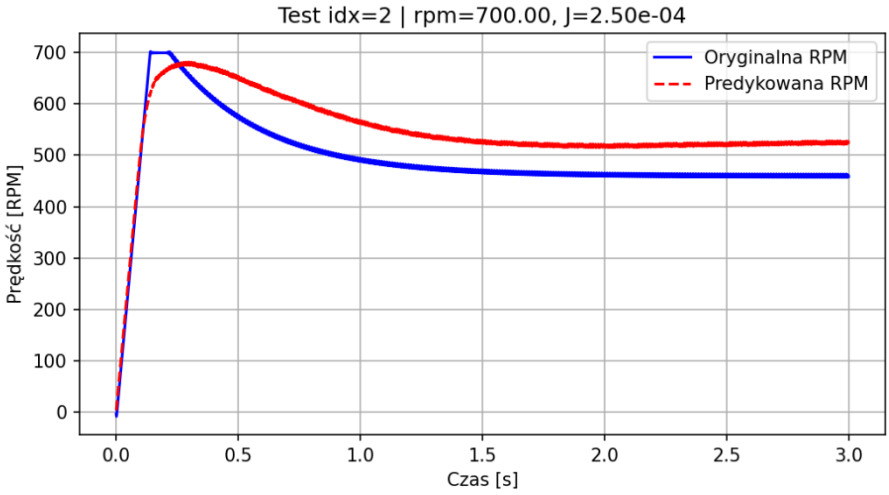
W innym przykładzie predykcja modelu w początkowym zakresie (do około 200 kroków) również odbiega od sekwencji uzyskanej podczas symulacji, co wskazuje na trudność w dokładnym modelowaniu początkowej dynamiki. Po fazie przejściowej napięcie stabilizuje się na poziomie około 20V co zostało dobrze oddane w predykcji SSN.



Rysunek 5.56: Uzyskany przebieg napięć dla przykładu z 1900 RPM

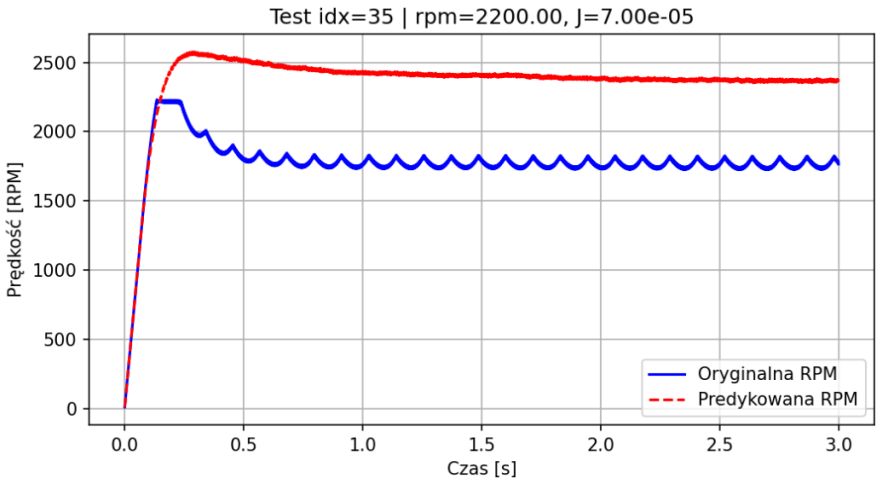
Ten przypadek prezentuje wysoce zaszumioną sekwencje napięcia wygenerowaną w trakcie symulacji. Model SSN generuje sekwencję, która jest znacznie bardziej wygładzona w porównaniu do oryginalnych danych. Predykcja odzwierciedla ogólny trend napięcia (początkowy spadek i późniejszą stabilizację), ale ignoruje szum w danych. Model efektywnie wydobywa istotne informacje z przykładów o niskiej wartości czystego sygnału. Jest to pozytywna cecha jeśli celem jest prognozowanie ogólnego trendu napięcia, a nie dokładne odwzorowanie przebiegu. Jednak w zastosowaniach wymagających precyzyjnej rekonstrukcji szczegółowych danych, jego predykcje mogą okazać się niewystarczające.

Przeprowadzone symulacje na modelu silnika pokazują że sekwencje napięć wygenerowane przez ssn potrafią lepiej wysterować silnikiem niż wartości wejściowe. Przykładem jest przebieg na rysunku 5.57.



Rysunek 5.57: Uzyskany przebieg prędkości obrotowej dla przykładu z oczekiwanym RPM wynoszącym 700

W porównaniu z siecią generującą nastawy PID, ta nauczyła się jak powinien wyglądać prawidłowy przebieg podawanego napięcia i jest w stanie wygenerować lepsze od podanych w przykładach co skutkuje precyzyjniejszym sterowaniem prędkością obrotową.



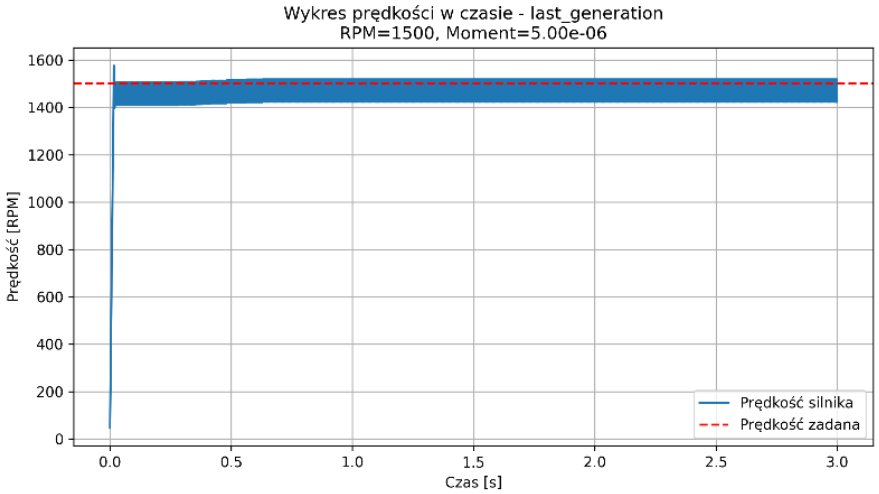
Rysunek 5.58: Uzyskany przebieg prędkości obrotowej dla przykładu z oczekiwanym RPM wynoszącym 2200

Rozdział 6

Symulacja na fizycznym stanowisku

6.1 Symulacja bez obciążenia

Przeprowadzono badanie trafności generowanej listy napięć przez algorytm genetyczny w rzeczywistych warunkach. Model matematyczny wykazywał problem w uzyskaniu czystego sygnału przebiegu prędkości obrotowej dla przypadków testowych bez momentu bezwładności lub z jego znikomą wartością.



Rysunek 6.1: Uzyskany przebieg RPM z nieznaczącym momentem bezwładności

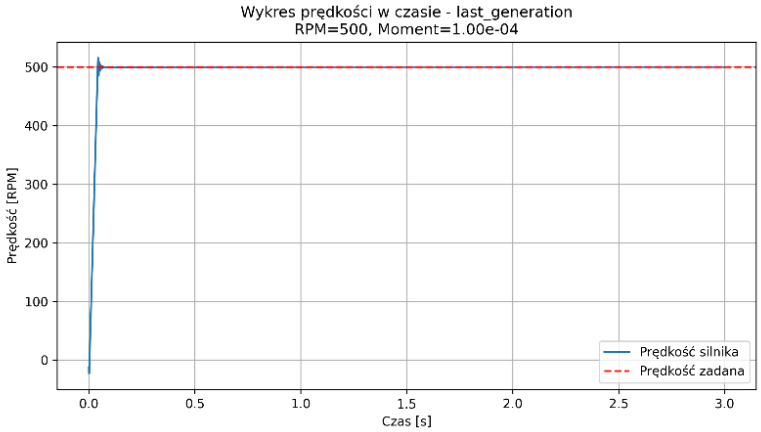
6.2 Przypadki testowe z obciążeniem

W związku z tym zastosowano przypadki testowe odpowiadające założeniu na stanowisku obciążenia silnika wraz z wyliczanym zestawem śrub i podkładek.

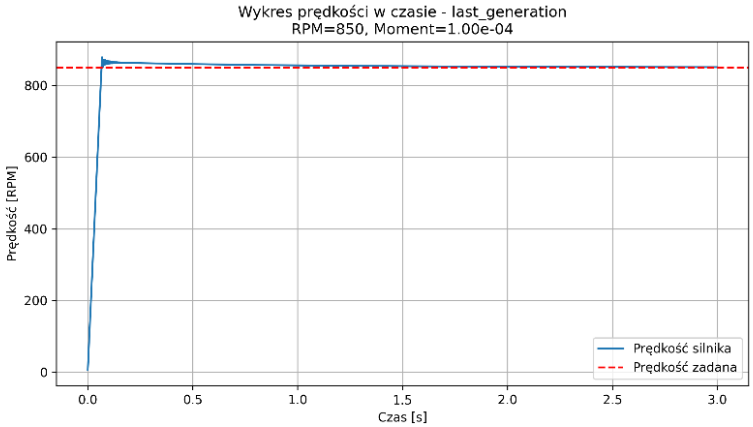
```
114    predkosci = [500, 850]
115    momenty_bezwladnosci = [1e-4]
```

Rysunek 6.2: Wykorzystane przypadki testowe

Dla wyżej wymienionych scenariuszy testu uzyskano w symulacji następujące wyniki:

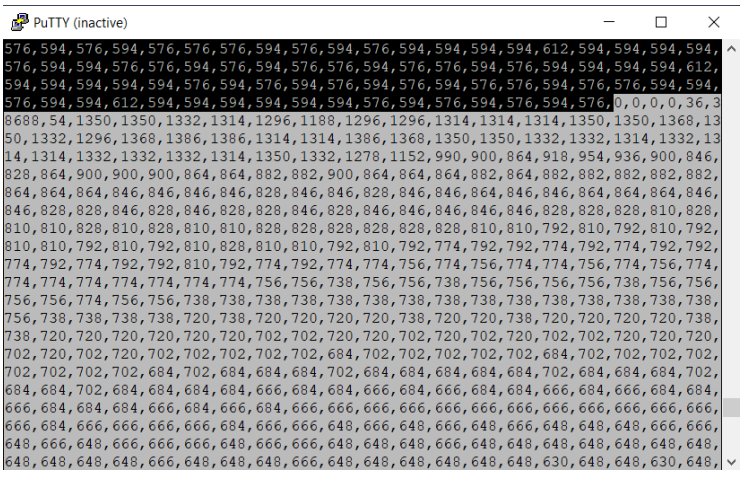


Rysunek 6.3: Symulacja dla 500RPM, moment bezwładności = 1e-4

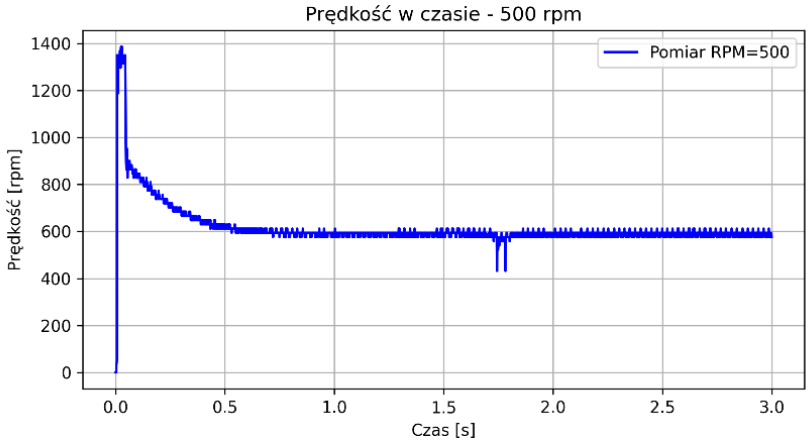


Rysunek 6.4: Symulacja dla 850RPM, moment bezwładności = 1e-4

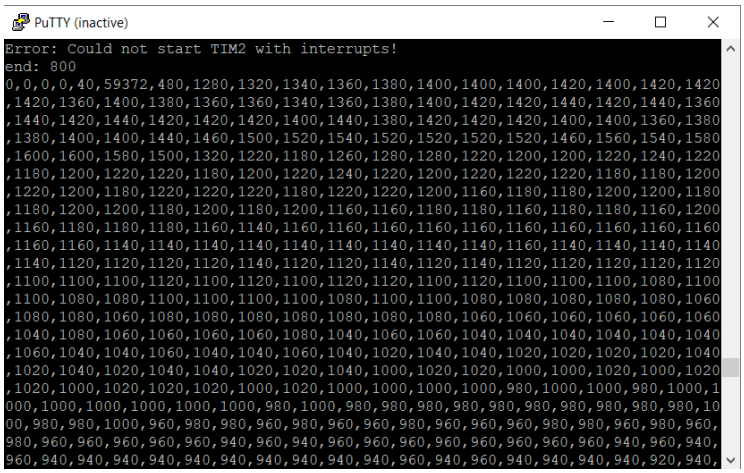
Testy na stanowisku wymagały odczytu przesyłanych wartości z terminala szeregowego Putty i stworzenia wykresu gdzie każda wypisana wartość odpowiada kolejnej 0.001 sekundy.



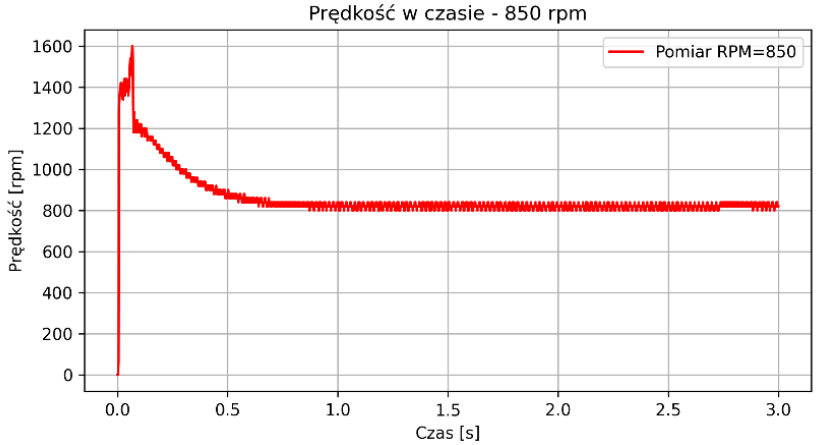
Rysunek 6.5: Dane wyświetlane w terminalu szeregowym dla przykładu 500RPM i momencie bezwładności 1e-4



Rysunek 6.6: Uzyskany rzeczywisty przebieg prędkości obrotowej dla nastawy na 500RPM



Rysunek 6.7: Dane wyświetlane w terminalu szeregowym dla przykładu 850RPM i momencie bezwładności 1e-4



Rysunek 6.8: Uzyskany rzeczywisty przebieg prędkości obrotowej dla nastawy na 850RPM

Rzeczywiste testy potwierdziły działanie stanowiska i poprawność generowanej sekwencji napięć przez algorytm genetyczny. Wykazały jednak jeden problem stanowiska pomiarowego. Zakładane obciążenie nie jest wystarczająco dokładnie zamontowane na wale silnika- przy rozpędzaniu występuje poślizg który pozwala silnikowi osiągnąć chwilowo znacznie wyższe prędkości niż zakładano. Po około 0.5 sekundy wartość prędkości obrotowej stabilizuje się blisko oczekiwanej.

Rozdział 7

Wnioski i obserwacje

7.1 Ocena stanowiska badawczego i sposobu komunikacji

Stanowisko badawcze, które obejmuje silnik prądu stałego, płytkę Nucleo F439ZI oraz enkoder obrotowy, zostało zaprojektowane i wykonane w sposób umożliwiający przeprowadzenie prostego zakresu testów. Opracowane oprogramowanie zapewniło precyzyjne sterowanie silnikiem poprzez generowanie sygnałów PWM, odczyt enkodera oraz sterowanie kierunkiem obrotów. Zawiodła kwestia mechaniczna dotyczące dokładania obciążenia na wał silnika, brak odpowiedniego sposobu montażu zaskutkował poślizgiem przy rozpędzaniu silnika w początkowej fazie pracy. Użycie płytki Nucleo F439ZI umożliwiło efektywne wykorzystanie mocy obliczeniowej mikrokontrolera do implementacji algorytmów SI, jednak w aplikacjach przemysłowych wymagałoby to dodatkowego zabezpieczenia przed zakłóceniami elektromagnetycznymi oraz odpowiedniej certyfikacji sprzętu.

7.2 Podsumowanie zastosowanych algorytmów SI

W ramach pracy przeanalizowano zastosowanie różnych algorytmów sztucznej inteligencji w doborze parametrów regulatora PID oraz generowaniu napięć sterujących dla silnika prądu stałego.

Algorytmy genetyczne

- AG sprawdziły się jako efektywne narzędzie do globalnej optymalizacji parametrów PID. Ich zdolność do przeszukiwania przestrzeni rozwiązań pozwoliła na uzyskanie optymalnych nastaw w zmiennych warunkach pracy. Dzięki odporności na lokalne minima, algorytmy te mogły znaleźć rozwiązania, które byłyby trudne do uzyskania za pomocą klasycznych metod optymalizacji.
- Algorytmy genetyczne wymagają dużej liczby iteracji i są stosunkowo wolne, co ogranicza ich użycie w czasie rzeczywistym. Ich koszt obliczeniowy rośnie wraz ze wzrostem liczby parametrów do optymalizacji.
- Algorytm ten okazał się dobrym wyborem do wstępnego dostrojenia regulatora PID w przypadku dużej liczby zmiennych i dynamicznych warunków pracy.

Sztuczne sieci neuronowe sterujące parametrami PID

- SSN uczyły się na podstawie wcześniej wygenerowanych danych przy pomocy AG, co miało poprawiało ich efektywność w przewidywaniu optymalnych parametrów. Wyniki końcowe pokazały jednak że zbudowany model nie radzi sobie z powierzonym zadaniem.

- SSN wymagały dokładnego i różnorodnego zestawu danych treningowych. Brak odpowiedniego treningu powodował problemy z odnalezieniem zależności pomiędzy przekazywanymi zmiennymi a nastawami PID. Algorytm zatrzymał się na dobieraniu parametrów w bezpiecznym zakresie który posiadał najwięcej przekazanych przykładów.
- Sieci sterujące regulatorem nie wykazały skuteczności w dobieraniu nastaw parametrów członów PID, potrzebowałyby znacznie większego i bardziej zróżnicowanego zbioru danych żeby prawidłowo optymalizować przesyłane nastawy.

Sztuczne sieci neuronowe generujące sekwencję napięć

- Sieci neuronowe sterujące napięciem mogły generować sekwencje napięć w sposób adaptacyjny, zapewniając płynną regulację prędkości obrotowej silnika. W porównaniu do klasycznych metod sterowania napięciem, SSN wykazały większą precyzję i zdolność do kompensacji niestabilności systemu.
- Podobnie jak w przypadku SSN sterujących PID, wymagały dużej ilości danych treningowych, a ich implementacja była bardziej złożona. Ponadto wprowadzenie adaptacyjnej regulacji napięcia wymagało starannej kalibracji (np zastosowania clipowania danych), aby uniknąć nadmiernych przeregulowań.
- SSN sterujące napięciem okazały się przydatne w zastosowaniach wymagających precyzyjnej kontroli prędkości, ale były mniej skuteczne w sytuacjach, gdzie kluczowa była szybka reakcja na zakłócenia. Generowana sekwencja napięć nie pozwalała na natychmiastową zmianę prędkości obrotowej silnika.

Drzewa decyzyjne

- Drzewo decyzyjne okazało się łatwe w implementacji i interpretacji, co czyni je przydatnym w systemach, gdzie kluczowe jest szybkie podejmowanie decyzji na podstawie ograniczonej liczby danych wejściowych.
- Jedgo deterministyczny charakter ograniczał zdolność adaptacji w dynamicznych warunkach pracy. Drzewa decyzyjne były również mniej skuteczne w problemach wymagających ciągłej regulacji, co czyniło je najmniej przydatnym algorytmem w omawianej pracy.
- Drzewo regresyjne było mało skuteczne ze względu na brak zdolności adaptacyjnych.

7.3 Potencjalne zastosowanie

Przedstawione rozwiązania mogą znaleźć zastosowanie w przemyśle, szczególnie w obszarach wymagających adaptacyjnego sterowania, takich jak:

- Sterowanie robotami przemysłowymi- wymagające dynamicznej regulacja napędów i kompensacji zmiennego obciążenia.
- Pojazdy elektryczne- do precyzyjnego sterowanie napędem w warunkach zmiennego obciążenia i napięcia zasilania.
- Automatyka HVAC (systemy grzewczo-wentylacyjne)- w adaptacyjnym sterowaniu ogrzewaniem, wentylacją i klimatyzacją w zmiennych warunkach środowiskowych.

Spis rysunków

Bibliografia

[1] *James Clerk Maxwell*. Wikipedia - biografia Maxwella. Dostęp: 25.01.2025. URL: https://en.wikipedia.org/w/index.php?title=James_Clerk_Maxwell&oldid=1270685840.

[2] *Kryterium stabilności Routha-Hurwitza*. Wikipedia - kryterium stabilności. Dostęp: 25.01.2025. URL: https://pl.wikipedia.org/w/index.php?title=Kryterium_stabilno%C5%9Bci_Routha-Hurwitza&oldid=34258499.

[3] *Regulator PID*. Wikipedia - regulator PID. Dostęp: 23.01.2025. URL: https://pl.wikipedia.org/w/index.php?title=Regulator_PID&oldid=75670534.

[4] *Lampa elektronowa*. Wikipedia - lampa elektronowa. Dostęp: 25.01.2025. URL: https://pl.wikipedia.org/w/index.php?title=Lampa_elektronowa&oldid=75866414.

[5] *Sterowanie predykcyjne*. Wikipedia - sterowanie predykcyjne. Dostęp: 23.01.2025. URL: https://pl.wikipedia.org/w/index.php?title=Sterowanie_predykcyjne&oldid=73171786.

[6] *Intel 4004*. Wikipedia - procesor Intel 4004. Dostęp: 23.01.2025. URL: https://pl.wikipedia.org/w/index.php?title=Intel_4004&oldid=73414448.

[7] *Schemat regulatora PID*. Schemat regulatora PID ze sprzężeniem zwrotnym. Dostęp: 28.01.2025. URL: https://www.astor.com.pl/poradnikautomatyka/regulator-pid-kurs-programowania-plc-od-podstaw-odc-20/.

[8] *Algorytmy genetyczne w Pythonie*. Przykłady algorytmów genetycznych w Pythonie. Dostęp: 23.01.2025. URL: https://mmazurek.dev/algorytmy-genetyczne-w-pythonie/.

[9] Łukasz Knypiński, Krzysztof Kowalski i Lech Nowak. "Adaptacja metody funkcji kary do algorytmu genetycznego w procesie projektowania urządzeń elektromagnetycznych". W: *Poznan University of Technology Academic Journals. Electrical Engineering* 96 (2018). Zastosowanie algorytmu genetycznego w projektowaniu urządzeń EM., s. 9–20. DOI: 10.21008/j.1897-0737.2018.96.0001.

[10] *Artificial Neural Networks and its Applications*. Wprowadzenie do sieci neuronowych. Dostęp: 25.01.2025. URL: https://www.geeksforgeeks.org/artificial-neural-networks-and-its-applications/.

[11] *Schemat sztucznej sieci neuronowej*. Schemat sztucznej sieci neuronowej w architekturze MLP. Dostęp: 28.01.2025. URL: https://forbot.pl/forum/topic/19925-zrozumiec-dzialanie-sieci-neuronowej-opis-dzialania-sztucznych-sieci-neuronowych/.

[12] *What Are Some Popular Activation Functions Used in Deep Learning?* Lista funkcji aktywacji. Dostęp: 25.01.2025. URL: https://www.quora.com/What-are-some-popular-activation-functions-used-in-deep-learning.

[13] *Understanding the Softmax Activation Function: A Comprehensive Guide*. Omówienie funkcji softmax. Dostęp: 23.01.2025. URL: https://www.singlestore.com/blog/a-guide-to-softmax-activation-function/.

[14] *Adam*. optymalizacja Adam. Dostęp: 25.01.2025. URL: https://deepai.org/machine-learning-glossary-and-terms/adam-machine-learning.

[15] *Sieci Neuronowe*. Materiały o sieciach neuronowych. Dostęp: 23.01.2025. URL: http://www.ai.c-labtech.net/sn/pod_prakt.html.

[16] *Czym jest przetwarzanie języka naturalnego (NLP)?* Wprowadzenie do NLP. Dostęp: 25.01.2025. URL: https://www.oracle.com/pl/artificial-intelligence/what-is-natural-language-processing/.

[17] *Multilayer Perceptrons in Machine Learning: A Comprehensive Guide*. Przegląd wielowarstwowych perceptronów. Dostęp: 25.01.2025. URL: https://www.datacamp.com/tutorial/multilayer-perceptrons-in-machine-learning.

[18] *Navigating the Path to Understanding Decision Trees in Machine Learning*. Artykuł o drzewach decyzyjnych. Dostęp: 23.01.2025. URL: https://techntales.medium.com/navigating-the-path-to-understanding-decision-trees-in-machine-learning-63459dc19ef3.

[19] *Konstrukcja Drzewa Decyzyjne*. Artykuł o konstrukcji drzew decyzyjnych. Dostęp: 25.01.2025. URL: https://www.ii.pwr.edu.pl/~kwasnicka/cybula_nedza_www/algorytmy/drzewa.html.

[20] *Czym jest i jak działa drzewo decyzyjne? - Mirosław Mamczur*. Wprowadzenie do drzew decyzyjnych. Dostęp: 25.01.2025. URL: https://miroslawmamczur.pl/czym-jest-i-jak-dziala-drzewo-decyzyjne/.

[21] *Jak działają szczotkowe silniki DC? Budowa i pomiary*. Opis budowy i działania silników szczotkowych DC. Dostęp: 23.01.2025. URL: https://forbot.pl/blog/jak-dzialaja-szczotkowe-silniki-dc-budowa-i-pomiary-id6788.

[22] Ping Zhang, Mingzhe Yuan i Hong Wang. "Self-Tuning PID Based on Adaptive Genetic Algorithms with the Application of Activated Sludge Aeration Process". W: *2006 6th World Congress on Intelligent Control and Automation*. T. 2. Regulacja procesu osadu czynnego przy użyciu PID i algorytmów genetycznych. 2006, s. 9327–9330. DOI: 10.1109/WCICA.2006.1713806.

[23] Hua Ji i Zhiyong Li. "Design of Neural Network PID Controller Based on Brushless DC Motor". W: *2009 Second International Conference on Intelligent Computation Technology and Automation*. T. 3. Kontroler PID z siecią neuronową dla silnika BLDC. 2009, s. 46–49. DOI: 10.1109/ICICTA.2009.479.

[24] Iraj Faraji Davoudkhani i Mohsen Akbari. "Adaptive Speed Control of Brushless DC (BLDC) Motor Based on Interval Type-2 Fuzzy Logic". W: *2016 24th Iranian Conference on Electrical Engineering (ICEE)*. Sterowanie silnikiem BLDC z logiką rozmytą typu 2. 2016, s. 1119–1124. DOI: 10.1109/IranianCEE.2016.7585689. URL: https://ieeexplore.ieee.org/document/7585689.

[25] *Nota katalogowa STM*. STM F439 DataSheet. Dostęp: 28.01.2025. URL: https://www.st.com/en/microcontrollers-microprocessors/stm32f429-439/documentation.html.

Załączniki

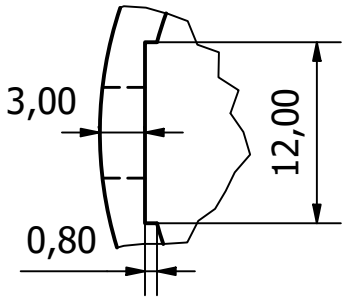
Link do repozytorium github z użytymi kodami:
https://github.com/KubaPawlo/DC-motor-control-using-AI

kod QR do repozytorium:

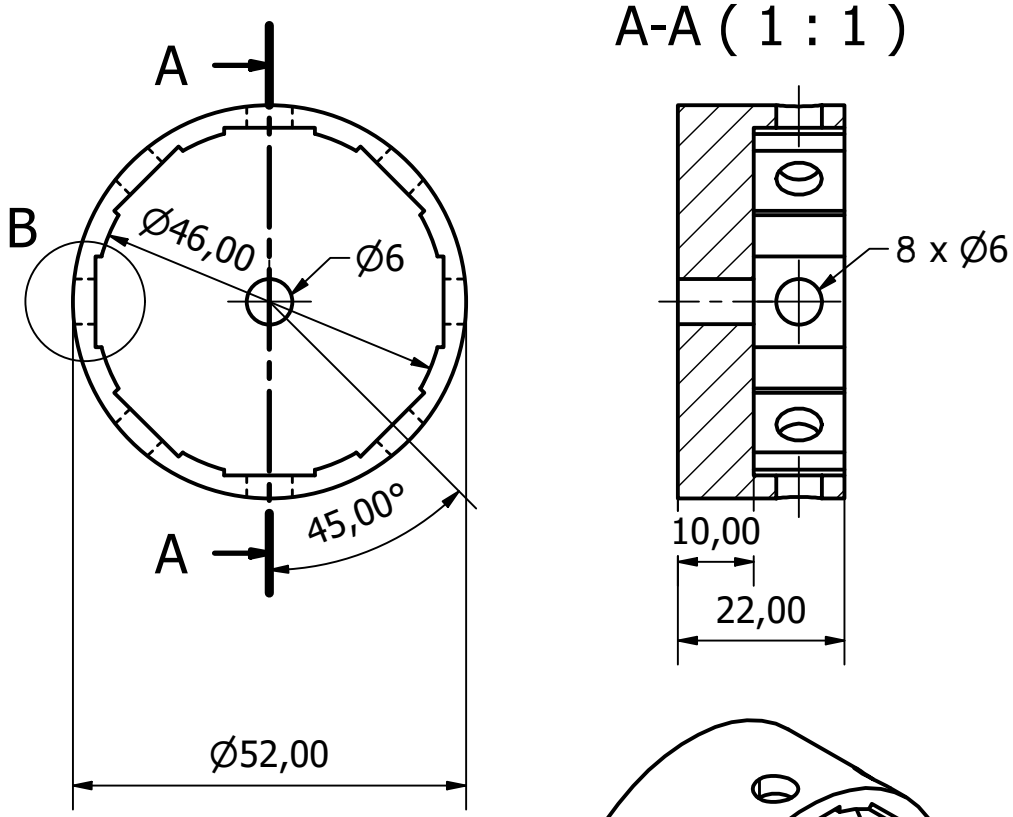
Rysunek techniczny zaprojektowanego obciążenia na silnik

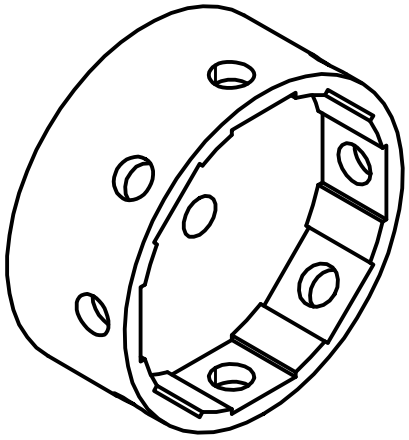
LISTA CZĘŚCI		
POZYCJA	ILOŚĆ	NUMER CZĘŚCI
1	1	Obciążenie silnika

B (2 : 1)



A-A (1 : 1)





Zaprojektowany przez	Sprawdzony przez	Zatwierdzony przez	Data		Data	
Jakub Pawłowicz					29.01.2025	

Obciążenie silnika		Wydanie	Arkusz 1 / 1