Wydział Elektroniki i Technik Informacyjnych
Politechnika Warszawska

Sztuczna Inteligencja w Automatyce

Sprawozdanie z projektu nr 2, zadanie 10

Dominik Bijoch

Warszawa, 2026

Spis treści

1. Cel Projektu

Celem projektu jest przeprowadzenie identyfikacji nieliniowego procesu dynamicznego przy użyciu metod sztucznej inteligencji, a następnie zaprojektowanie i przetestowanie algorytmu regulacji predykcyjnej. Realizacja zadania obejmuje następujące etapy:

1. **Analiza i symulacja procesu:** Wyznaczenie charakterystyki statycznej procesu oraz wygenerowanie zbiorów danych uczących i weryfikujących poprzez symulację losowych zmian sygnału sterującego w zakresie od $u^{min} = -1$ do $u^{max} = 1$, począwszy od punktu pracy $u = y = x = 0$. Zbiory te posłużą do późniejszej identyfikacji obiektów.
2. **Modelowanie neuronowe metodą ELM:** Opracowanie serii modeli neuronowych typu *Extreme Learning Machines* (ELM) z jedną warstwą ukrytą, wykorzystującą funkcję aktywacji tangens hiperboliczny. Celem jest zbadanie wpływu liczby neuronów ukrytych ($K = 5, 10, 15, \dots$) na jakość modelu oraz wybór struktury zapewniającej najmniejszy błąd w trybie rekurencyjnym.
3. **Modelowanie przy użyciu klasycznych sieci neuronowych:** Stworzenie modeli neuronowych uczonych iteracyjnie (LBFGS/Adam w *Deep Learning Toolbox*). Zadanie obejmuje porównanie efektywności różnych algorytmów uczenia oraz ocenę dokładności modeli w trybie predykcji jednokrokowej i rekurencyjnej.
4. **Modelowanie liniowe:** Wyznaczenie referencyjnego modelu liniowego metodą najmniejszych kwadratów w celu porównania jego dokładności z modelami nieliniowymi.
5. **Zaprojektowanie układu regulacji (NPL):** Implementacja algorytmu regulacji predykcyjnej z Nieliniową Predykcją i Linearyzacją (NPL), bazującego na najlepszych wyznaczonych modelach neuronowych. Etap ten wymaga nastrojenia parametrów regulatora (horyzonty, współczynnik kary λ) oraz weryfikacji działania układu dla zadanej trajektorii zmian sygnału wyjściowego.

Wszystkie symulacje, procesy uczenia oraz implementacja algorytmów sterowania przeprowadzono w środowisku MATLAB.

2. Symulacja procesu

W celu poznania właściwości badanego obiektu dynamicznego (`proces10`), przeprowadzono analizę jego właściwości statycznych oraz przygotowano zbiory danych niezbędne do procesu uczenia i weryfikacji modeli neuronowych. Wszystkie symulacje wykonano w oparciu o otrzymany symulator procesu.

2.1. Wyznaczenie charakterystyki statycznej

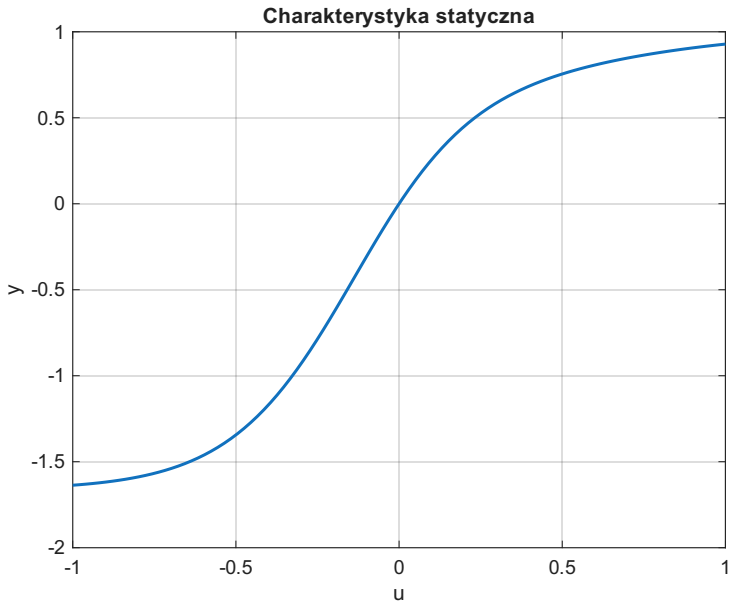
Pierwszym etapem analizy było wyznaczenie charakterystyki statycznej procesu, czyli zależności $y(u)$ w stanie ustalonym. Badanie przeprowadzono dla sygnału sterującego u zmieniającego się w zakresie $u \in [u^{min}, u^{max}]$, gdzie zgodnie z treścią zadania przyjęto $u^{min} = -1$ oraz $u^{max} = 1$.

Dla każdego punktu pomiarowego przeprowadzono symulację trwającą 1000 kroków dyskretnych, co pozwoliło na osiągnięcie stanu ustalonego przez obiekt. Wartość wyjściową y dla danego stałego sterowania u wyznaczono jako średnią arytmetyczną z ostatnich 20 próbek odpowiedzi. Kod realizujący to zadanie przedstawiono na listingu 2.1.

Listing 2.1. Wyznaczanie charakterystyki statycznej

```
u_min = -1;
u_max = 1;
n_points = 100;
kmin = 5;
k_sim = 1000;

u_values = linspace(u_min, u_max, n_points);
y_ss = zeros(size(u_values));

for i = 1:length(u_values)
    u_val = u_values(i);
    u = u_val * ones(1, k_sim);
    y = zeros(1, k_sim);
    x = zeros(1, k_sim);

    % Symulacja dojścia do stanu ustalonego
    for k = kmin:k_sim
        [y(k), x(k)] = proces10_symulator(u(k-3), u(k-4), x(k-1), x(k-2));
    end

    % Uśrednianie ostatnich próbek
    n_avg = min(20, k_sim - kmin + 1);
    y_ss(i) = mean(y(end-n_avg+1:end));
end
```

Uzyskaną charakterystykę statyczną przedstawiono na rys. 2.1. Analiza wykresu wskazuje na silnie nieliniowy charakter obiektu. Kształt krzywej przypomina funkcję sigmoidalną.



Rys. 2.1. Charakterystyka statyczna procesu $y(u)$

2.2. Generowanie danych uczących i weryfikujących

W celu przeprowadzenia identyfikacji neuronowej oraz późniejszej weryfikacji modeli, wygenerowano dwa rozłączne zbiory danych:

— **Zbiór uczący** – wykorzystywany do optymalizacji wag sieci neuronowych.
— **Zbiór weryfikujący** – służący do oceny zdolności generalizacji modeli.

Zgodnie z wymaganiami projektowymi, sygnał sterujący $u(k)$ generowano jako sekwencję losowych zmian skokowych o amplitudzie z rozkładu jednostajnego w przedziale $[-1, 1]$. Przyjęto okres zmian sygnału sterującego wynoszący 40 kroków dyskretnych, co jest wystarczające do osiągnięcia przez obiekt stanu ustalonego. Każdy ze zbiorów zawiera 3996 próbek.

Dane zostały sformatowane do postaci wektorów wejściowych sieci neuronowej, uwzględniając rzędy dynamiki zgodne z symulatorem procesu: wejścia $u(k-3), u(k-4)$ oraz wyjścia $y(k-1), y(k-2)$. Funkcjonalny kod generujący dane oraz formujący macierze uczące przedstawiono na listingu 2.2.

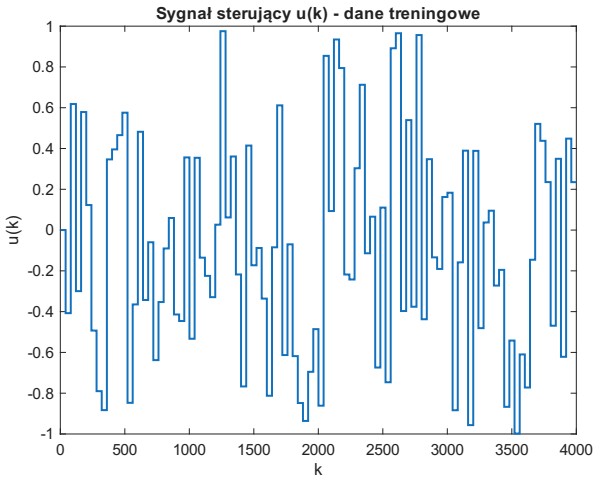
Listing 2.2. Generowanie i formatowanie danych uczących i weryfikujących

```
% Generowanie surowych przebiegów czasowych
[y_train, u_train] = generate_dataset(69, 100);
[y_val, u_val] = generate_dataset(420, 100);

k_min = 5; k_max_train = length(u_train); k_max_val = length(u_val);
N_train = k_max_train - k_min + 1;
N_val = k_max_val - k_min + 1;

% Inicjalizacja macierzy danych
X_train = zeros(4, N_train); X_val = zeros(4, N_val);
Y_train = zeros(1, N_train); Y_val = zeros(1, N_val);

% Tworzenie wektorów uczących [u(k-3); u(k-4); y(k-1); y(k-2)]
for i = k_min:N_train
    X_train(:, i-k_min+1) = [u_train(i-3); u_train(i-4); ...
                             y_train(i-1); y_train(i-2)];
    Y_train(i-k_min+1) = y_train(i);
end
```

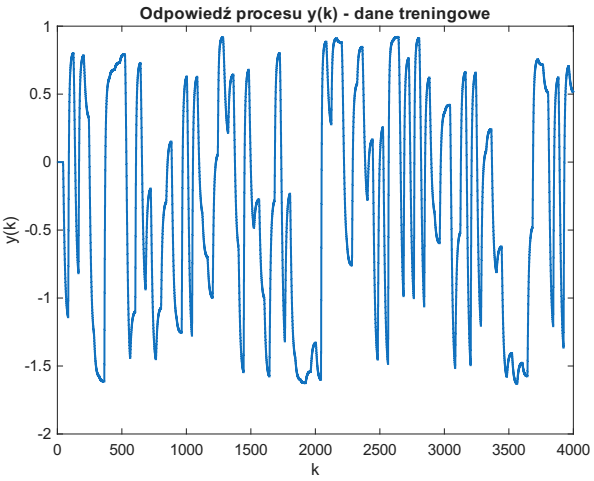
```
% Tworzenie wektorów weryfikujących
for i = k_min:N_val
    X_val(:, i-k_min+1) = [u_val(i-3); u_val(i-4); y_val(i-1); y_val(i-2)];
    Y_val(i-k_min+1) = y_val(i);
end

% Funkcja pomocnicza generująca przebiegi
function [y, u] = generate_dataset(seed, n_points)
    u_min = -1; u_max = 1;
    u_period = 40; % okres zmian u
    N = n_points * u_period;

    rng(seed); % seed dla powtarzalności
    u_values = u_min + (u_max - u_min) * rand(1, n_points);

    u = zeros(1, N);
    for i = 0:(n_points-1)
        u(i*u_period + 1 : (i+1)*u_period) = u_values(i+1);
    end

    y = zeros(1, N); x = zeros(1, N);
    for k = 5:N
        [y(k), x(k)] = proces10_symulator(u(k-3), u(k-4), x(k-1), x(k-2));
    end
end
```

Przebiegi czasowe sygnałów sterujących oraz odpowiedzi obiektu dla zbioru uczącego zaprezentowano na rys. 2.2-2.3, natomiast dla zbioru weryfikującego na rys. 2.4-2.5.

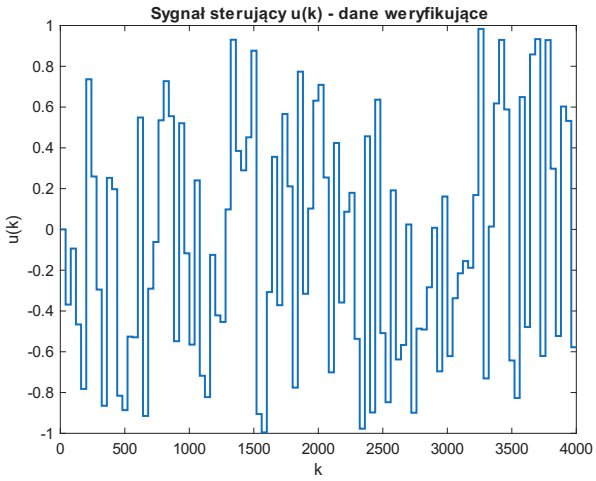
Wygenerowane sygnały pokrywają cały zakres zmienności obiektu zdefiniowany charakterystyką statyczną, co jest kluczowe dla poprawnego nauczenia modeli.



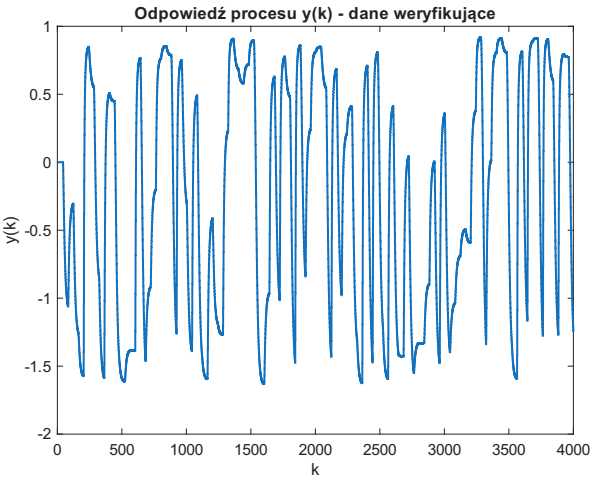
Rys. 2.2. Sygnał sterujący $u(k)$ - zbiór uczący



Rys. 2.3. Odpowiedź procesu $y(k)$ - zbiór uczący



Rys. 2.4. Sygnał sterujący $u(k)$ - zbiór weryfikujący



Rys. 2.5. Odpowiedź procesu $y(k)$ - zbiór weryfikujący

3. Modelowanie neuronowe procesu przy użyciu sieci ELM

Celem tego etapu projektu było opracowanie nieliniowego modelu procesu z wykorzystaniem sieci neuronowych typu *Extreme Learning Machine* (ELM).

3.1. Struktura sieci i algorytm uczenia

Zastosowano sieć neuronową z jedną warstwą ukrytą z losowo przyjętymi wagami pierwszej warstwy oraz warstwą wyjściową rozwiązywaną analitycznie. Zgodnie z rzędem dynamiki symulatora procesu, wektor wejściowy sieci zdefiniowano następująco:

$$\boldsymbol{x}(k) = [u(k-3), u(k-4), y(k-1), y(k-2)]^T \tag{3.1}$$

Jako funkcję aktywacji w warstwie ukrytej przyjęto tangens hiperboliczny (tanh). Kod realizujący ten algorytm przedstawiono na listingu 3.1. Kod wczytuje dane, następnie inicjalizuje losowo wagi pierwszej warstwy - `w10` oraz `w1`. Losowość ta jest kontrolowana zmienną `seed` w celu powtarzalności eksperymentów. Następnie obliczana jest macierz pomiarów oraz obliczane są analitycznie optymalne wagi drugiej warstwy – `w20` oraz `w2`. Ostatecznie przeprowadzane są predykcje oraz obliczane błędy zoptymalizowanego modelu zarówno dla trybu z jak i bez rekurencji. Logika predykcji rekurencyjnej znajduje się w funkcji `recursive_prediction`, która przeprowadza symulację wykorzystującą model.

Listing 3.1. Implementacja algorytmu uczenia i predykcji sieci ELM

```
load('data/data_train.mat'); % X_train, Y_train, N_train
load('data/data_val.mat'); % X_val, Y_val, N_val

% liczba neuronów w warstwie ukrytej
K = 100;
seed = 100; % ziarno generatora liczb losowych
rng(seed);

inputSize = size(X_train, 1);
outputSize = size(Y_train, 1);
numParams=(inputSize+1)*K+(K+1)*outputSize;

Xtrain = X_train;
Ytrain = Y_train(:);
Xval = X_val;
Yval = Y_val(:);

% losowe wagi pierwszej warstwy
w10 = 2*(rand(K,1)-0.5);
w1 = 2*(rand(K,inputSize)-0.5);

% macierz pomiarów generowana blokowo
Vtrain = [ones(N_train,1) tanh(w10 + w1*Xtrain)'];

% obliczenie optymalnych wag drugiej warstwy
weightsW2 = Vtrain\Ytrain;
```

```
w20 = weightsW2(1);
w2 = weightsW2(2:K+1)';

% obliczenie wyjścia modelu dla wszystkich próbek obu zbiorów (z/bez rekurencji)
Ymod_train = model(w10, w1, w20, w2, Xtrain); Ymod_train = Ymod_train(:);
Ymod_val   = model(w10, w1, w20, w2, Xval);   Ymod_val   = Ymod_val(:);
MSEtrain = mean((Ytrain - Ymod_train).^2);
MSEval   = mean((Yval   - Ymod_val).^2);

y_hist_train = [Ytrain(2); Ytrain(1)];
y_hist_val   = [Yval(2);   Yval(1)];
Yrec_train  = recursive_prediction(w10,w1,w20,w2,Xtrain,N_train, y_hist_train);
Yrec_val    = recursive_prediction(w10,w1,w20,w2,Xval,N_val,   y_hist_val);
MSErectrain = mean((Ytrain - Yrec_train).^2);
Mserecval   = mean((Yval   - Yrec_val  ).^2);

%%% funkcja wyjście modelu
function Ymod = model(w10, w1, w20, w2, X)
    Ymod = w20+w2*tanh(w10+w1*X);
end
%%% funkcja predykcji rekurencyjnej
function [Yrec_train] = recursive_prediction(w10,w1,w20,w2,X,n,y_hist)
Yrec_train = zeros(n,1);
for j = 1:n
    x = X(:,j);
    x(3) = y_hist(1);
    x(4) = y_hist(2);
    y_mod = model(w10,w1,w20,w2,x);
    Yrec_train(j) = y_mod;
    y_hist = [y_mod; y_hist(1)];
end
end
```

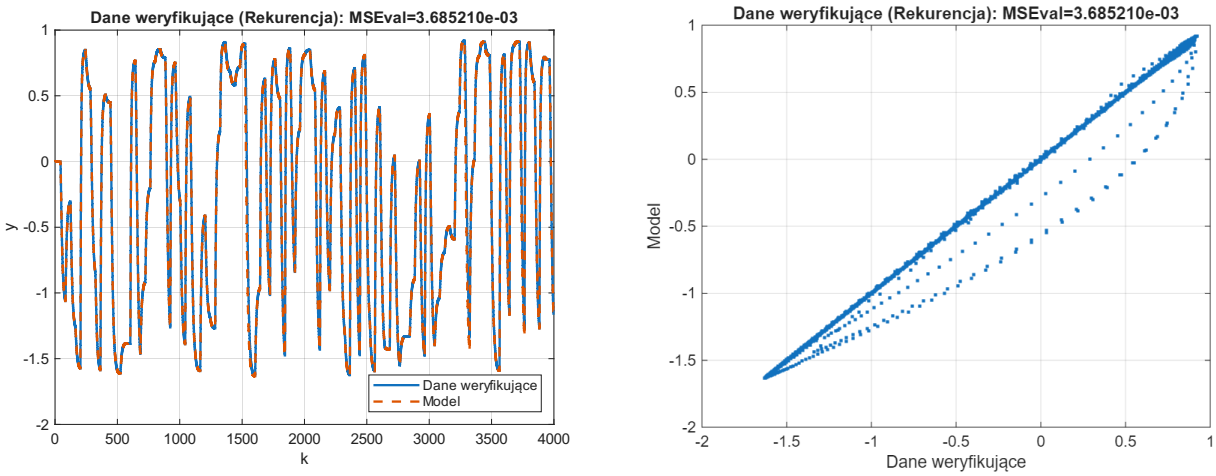
3.2. Dobór liczby neuronów ukrytych

W celu doboru optymalnej struktury sieci przeprowadzono serię eksperymentów dla liczby neuronów ukrytych $K \in \{5, 10, 15, 20, 30, 40, 50, 70, 100, 130, 400\}$. Ze względu na losową inicjalizację wag, dla każdej wartości K proces uczenia powtórzono 10-krotnie dla różnego ziarna generatora liczb losowych. Dla każdej struktury sieci wybrano model o najmniejszym błędzie średniokwadratowym (MSE) na zbiorze weryfikującym w trybie rekurencyjnym.

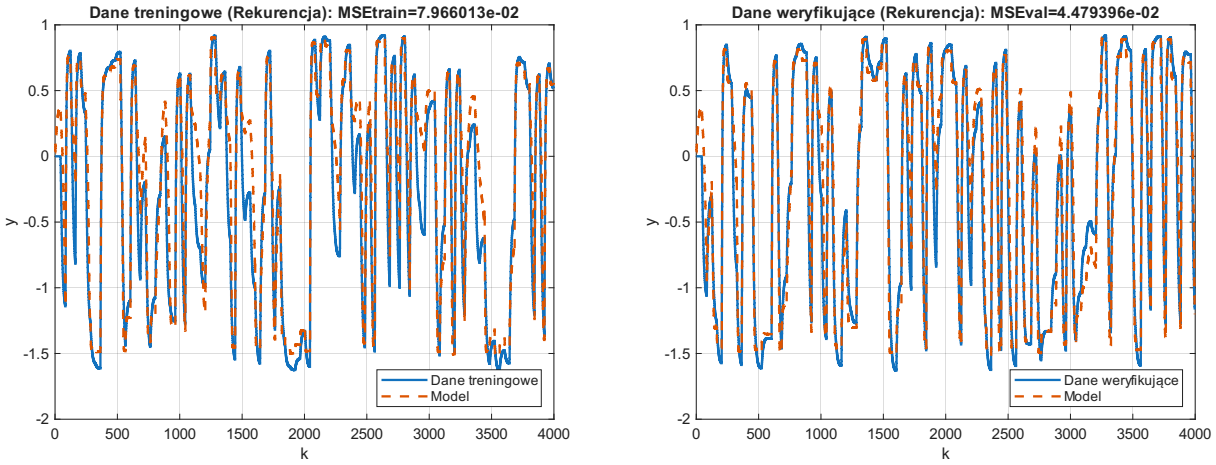
Zależność błędów od liczby neuronów przedstawiono w tab. 3.1.

Tab. 3.1. Wpływ liczby neuronów ukrytych K na błędy treningowe oraz weryfikującę dla trybu bez rekurencji oraz z rekurencją (rek)

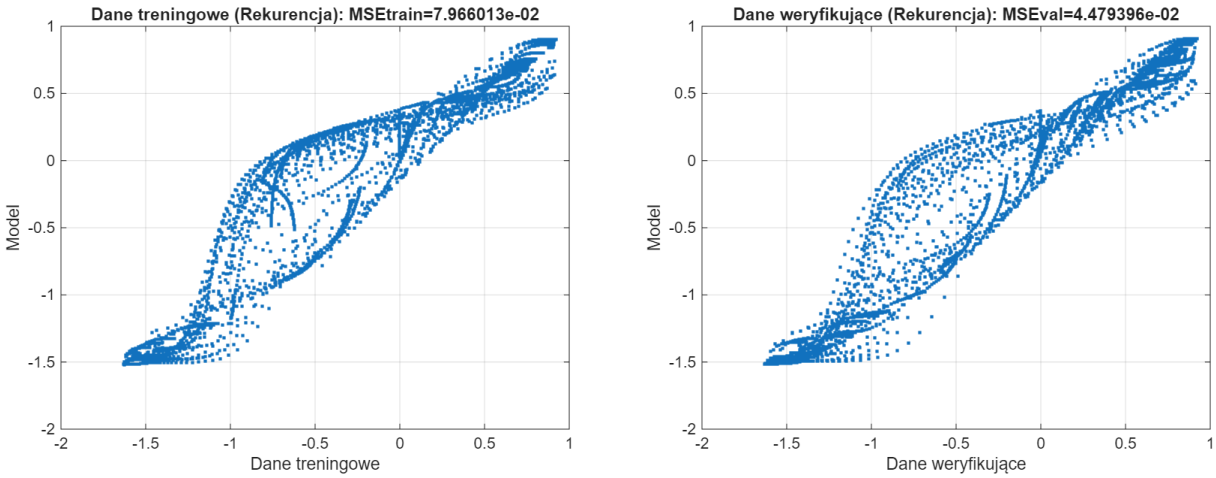
Liczba neuronów K	**MSE train**	**MSE train (rek)**	**MSE val**	**MSE val (rek)**
5	$2{,}077 \cdot 10^{-3}$	$7{,}966 \cdot 10^{-2}$	$2{,}210 \cdot 10^{-3}$	$4{,}479 \cdot 10^{-2}$
10	$6{,}254 \cdot 10^{-4}$	$1{,}177 \cdot 10^{-2}$	$7{,}492 \cdot 10^{-4}$	$1{,}351 \cdot 10^{-2}$
15	$8{,}741 \cdot 10^{-5}$	$7{,}378 \cdot 10^{-3}$	$1{,}121 \cdot 10^{-4}$	$7{,}742 \cdot 10^{-3}$
20	$1{,}801 \cdot 10^{-5}$	$3{,}830 \cdot 10^{-3}$	$1{,}834 \cdot 10^{-5}$	$4{,}130 \cdot 10^{-3}$
30	$6{,}200 \cdot 10^{-6}$	$6{,}364 \cdot 10^{-4}$	$8{,}077 \cdot 10^{-6}$	$9{,}170 \cdot 10^{-4}$
40	$7{,}979 \cdot 10^{-7}$	$5{,}381 \cdot 10^{-5}$	$1{,}483 \cdot 10^{-6}$	$1{,}072 \cdot 10^{-4}$
50	$1{,}281 \cdot 10^{-6}$	$1{,}123 \cdot 10^{-4}$	$1{,}652 \cdot 10^{-6}$	$1{,}168 \cdot 10^{-4}$
70	$2{,}216 \cdot 10^{-7}$	$1{,}941 \cdot 10^{-5}$	$4{,}208 \cdot 10^{-7}$	$3{,}072 \cdot 10^{-5}$
100	$3{,}104 \cdot 10^{-8}$	$3{,}042 \cdot 10^{-6}$	$1{,}056 \cdot 10^{-7}$	$6{,}609 \cdot 10^{-6}$
130	$4{,}173 \cdot 10^{-9}$	$5{,}728 \cdot 10^{-7}$	$1{,}672 \cdot 10^{-7}$	$8{,}152 \cdot 10^{-6}$
400	$2{,}462 \cdot 10^{-12}$	$3{,}214 \cdot 10^{-8}$	$1{,}097 \cdot 10^{-5}$	$3{,}685 \cdot 10^{-3}$



Rys. 3.1. Przebiegi dla zbioru weryfikującego (Model $K = 400$)



Rys. 3.2. Przebiegi dla obydwu zbiorów (Model $K = 5$)



Rys. 3.3. Wykresy punktowe dopasowania w trybie rekurencyjnym (Model $K = 5$)

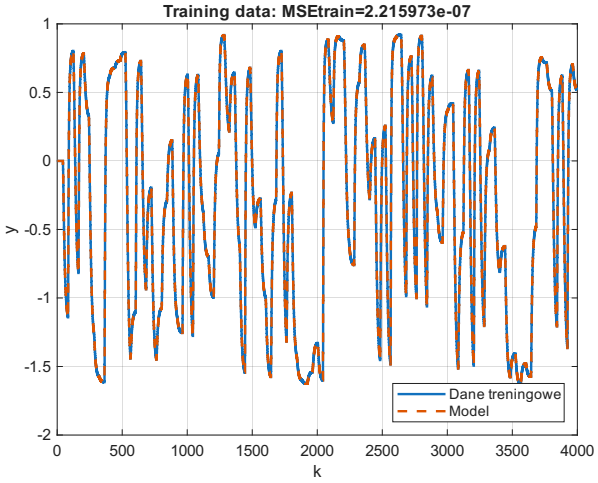
Analiza wyników wykazała zmniejszanie się wartości wszystkich błędów wraz ze wzrostem parametru K, aż do poziomu $K = 100$. Dalsze zwiększanie liczby neuronów ukrytych powoduje spadek błędu na zbiorze treningowym, ale jednocześnie wzrost błędu dla danych weryfikujących

w trybie predykcji rekurencyjnej, co sugeruje przewymiarowanie modelu. Na rys. 3.2-3.3 przedstawiono przebiegi ilustrujące model dla $K = 5$, natomiast na rys. 3.1 przedstawiono model dla $K = 400$. W przypadku 5 neuronów widoczne jest, że model posiada zbyt małą liczbę parametrów, gdyż w trybie rekurencyjnym nie jest w stanie całkowicie poprawnie odwzorować danych, nawet dla zbioru treningowego. Natomiast w przypadku 400 neuronów w warstwie ukrytej zauważalne jest przewymiarowanie modelu, co dobrze widać na wykresie punktowym.

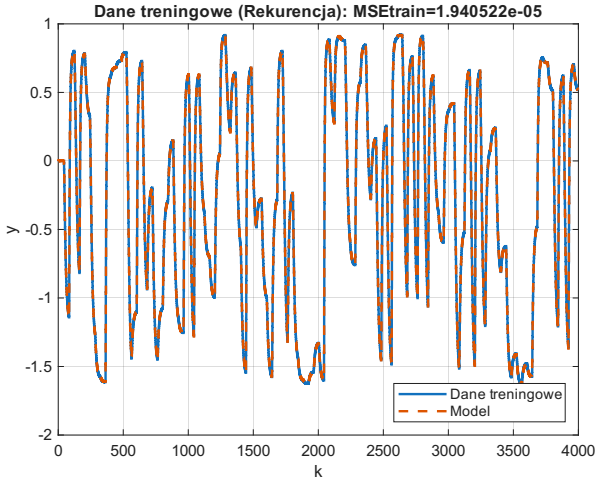
Ostatecznie wybrano strukturę sieci zawierającą $\mathbf{K = 70}$ **neuronów ukrytych**. Decyzję tę podjęto, ponieważ po przekroczeniu $K \approx 100$ neuronów model ulega przewymiarowaniu. Ponadto różnica błędu średniokwadratowego na zbiorze weryfikującym z rekurencją między $K = 70$ a $K = 100$ nie jest na tyle duża, aby uzasadnić wybór bardziej złożonego modelu.

3.3. Porównanie najlepszego modelu w trybie bez rekurencji i z rekurencją

Na rys. 3.6 zestawiono przebiegi wyjścia modelu w porównaniu z danymi treningowymi. Natomiast rys. 3.9 przedstawia wykresy punktowe dla zbioru uczącego. Analogiczne zestawienie dla danych weryfikujących przedstawiono na rys. 3.12-3.15. Zarówno na przebiegach czasowych, jak i na wykresach punktowych widoczne jest poprawne działanie modelu. Model skutecznie odwzorowuje dane treningowe oraz weryfikujące zarówno w trybie jednokrokowym, jak i rekurencyjnym.

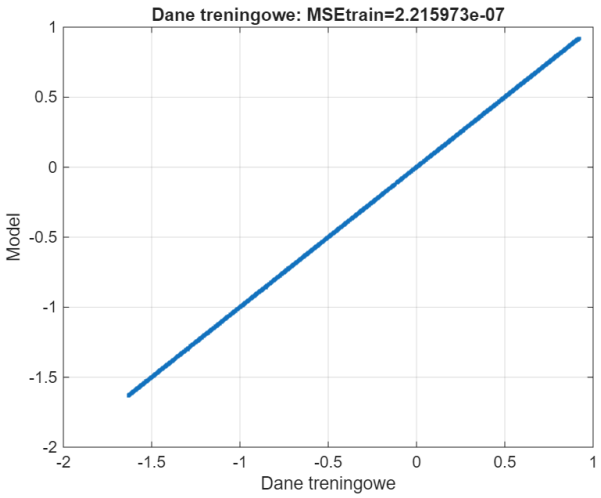


Rys. 3.4. Tryb bez rekurencji

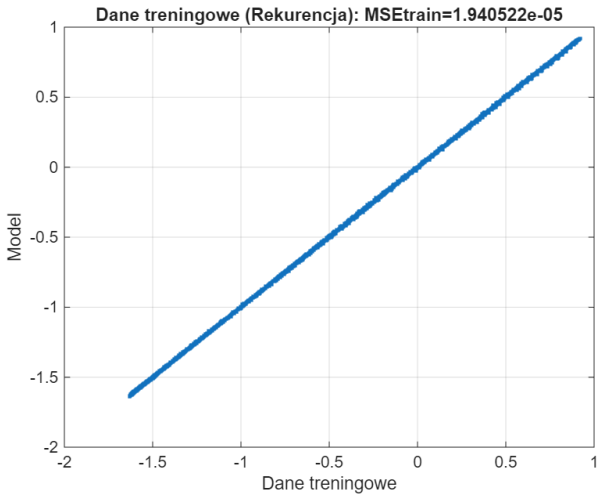


Rys. 3.5. Tryb rekurencyjny

Rys. 3.6. Przebiegi czasowe dla zbioru uczącego (Model $K = 70$)

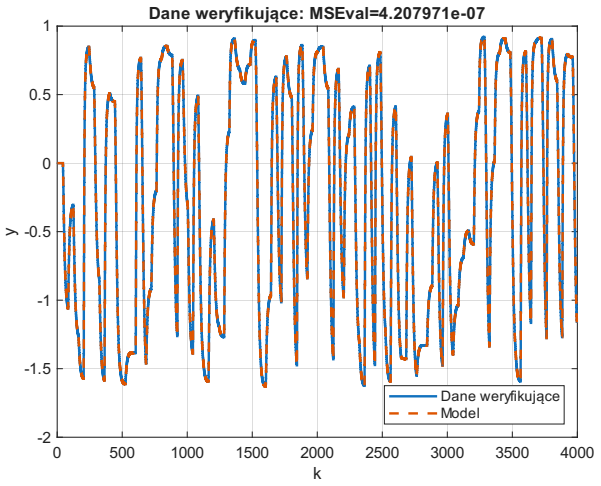


Rys. 3.7. Tryb bez rekurencji

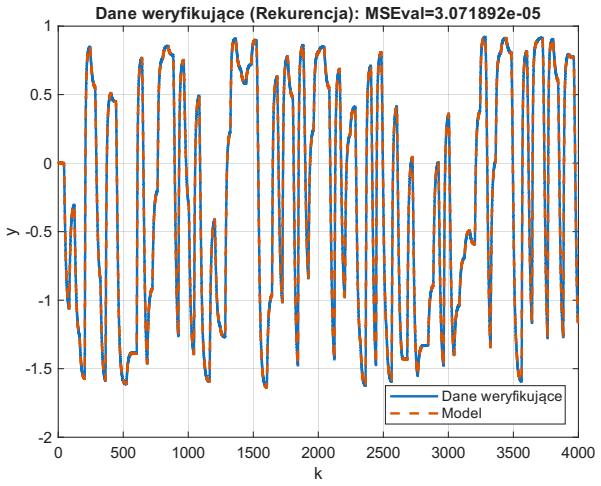


Rys. 3.8. Tryb rekurencyjny

Rys. 3.9. Wykresy punktowe dopasowania dla zbioru uczącego (Model K=70)

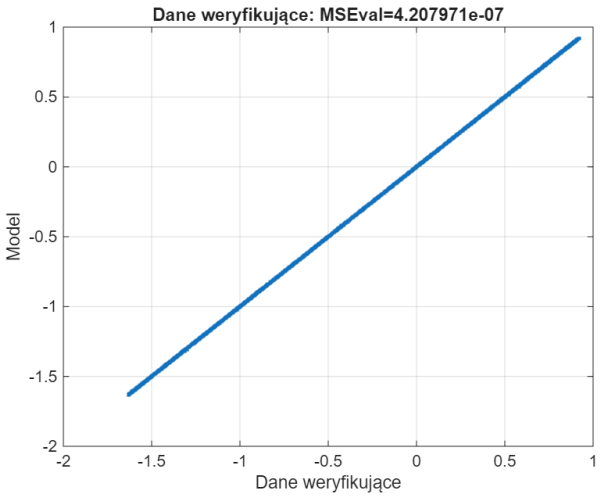


Rys. 3.10. Tryb bez rekurencji

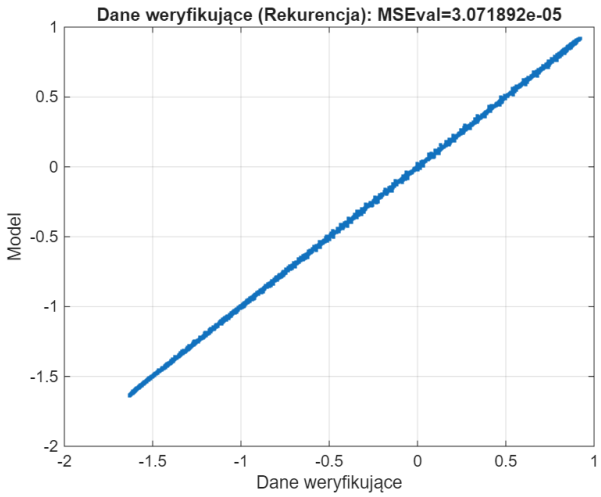


Rys. 3.11. Tryb rekurencyjny

Rys. 3.12. Przebiegi czasowe dla zbioru weryfikującego (Model $K = 70$)



Rys. 3.13. Tryb bez rekurencji



Rys. 3.14. Tryb rekurencyjny

Rys. 3.15. Wykresy punktowe dopasowania dla zbioru weryfikującego (Model K=70)

4. Modelowanie przy użyciu Deep Learning Toolbox

W ramach realizacji tego zadania przeprowadzono identyfikację procesu z wykorzystaniem *Deep Learning Toolbox*.

4.1. Definicja i implementacja modelu

Zdefiniowano wejścia sieci jako $x_1 = u(k-3)$, $x_2 = u(k-4)$, $x_3 = y(k-1)$, $x_4 = y(k-2)$. Zaprojektowano sieć, w której sygnał wyjściowy $y(k)$ zależy zarówno od warstwy ukrytej, jak i bezpośrednich przekształceń sygnałów wejściowych.

Równanie wyjścia sieci ma postać:

$$y(k) = w_1^{22}\cos(x_1(k)) + w_2^{22}x_2(k) + w_0^2 + \sum_{i=1}^{K} w_i^2 v_i \tag{4.1}$$

gdzie $v_i(k) = \tanh(z_i(k))$ to wyjście i-tego neuronu ukrytego.

Suma sygnałów wejściowych poszczególnych neuronów ukrytych ma postać:

$$z_i(k) = \begin{cases} w_{i,0}^1 + \sum_{j=1}^{4} w_{i,j}^1 x_j(k) + w_1^{11}\sin(x_2(k)) & \text{gdy } i = 1 \\ w_{i,0}^1 + \sum_{j=1}^{4} w_{i,j}^1 x_j(k) & i > 1 \end{cases} \tag{4.2}$$

Kod realizujący model neuronowy z wykorzystaniem pakietu *Deep Learning Toolbox* przedstawiono na listingu 4.1. Kod realizuje inicjalizację wag jako obiektów `dlarray` oraz definicję funkcji `model`, która odwzorowuje równania 4.1-4.2, przy zastosowaniu funkcji aktywacji tangens hiperboliczny w warstwie ukrytej o K neuronach.

Proces uczenia odbywa się w pętli optymalizacyjnej z wykorzystaniem mechanizmu automatycznego różniczkowania (`dlgradient`) do minimalizacji błędu średniokwadratowego (MSE), z możliwością wyboru algorytmu LBFGS lub Adam. Po zakończeniu treningu skrypt przeprowadza walidację modelu, wyznaczając błędy zarówno dla predykcji jednokrokowej, jak i predykcji rekurencyjnej.

Listing 4.1. Kod realizujący model neuronowy

```
%% dane
load('data/data_train.mat');
load('data/data_val.mat');

K=4;%liczba neuronów w warstwach ukrytych

% konfiguracja algorytmu uczącego
algorithm = 'lbfgs';%adam, 'lbfgs'
numEpochs = 200;

% ustalenie ziarna generatora liczb losowych
seed = 30;
rng(seed);

% tylko dla 'adam'
learningRate = 0.01;
```

```
inputSize = size(X_train, 1);
outputSize = size(Y_train, 1);

Xtrain = dlarray(X_train);
Ytrain = dlarray(Y_train);
Xval = dlarray(X_val);
Yval = dlarray(Y_val);

% Inicjalizacja wag
net.w1 = dlarray(randn(K, inputSize) * 0.1);
net.w10 = dlarray(zeros(K, 1));
net.w2 = dlarray(randn(outputSize, K) * 0.1);
net.w20 = dlarray(zeros(outputSize, 1));
net.w11 = dlarray(randn(inputSize, 1) * 0.1);
net.w22 = dlarray(randn(inputSize, 1) * 0.1);

numParams = (inputSize+1)*K+(K+1)*outputSize+2*inputSize;

% błędy dla całych zbiorów
lossTrainGlobal = zeros(numEpochs, 1);
lossValGlobal = zeros(numEpochs, 1);

fprintf("algorithm epoch/epochmax MSEtrainGlobal MSEvalGlobal\n");
fprintf('%s\n', repmat(char(9472), 1, 45));

% inicjalizacja stanu dla optymalizatorów
trailingAvg = [];
trailingAvgSq = [];
stateLBFGS = [];
iteration = 0;

tic;
switch algorithm
    case 'lbfgs'
        lossFcn =  @(net) dlfeval(@modelLoss, net, Xtrain, Ytrain);
        for epoch = 1:numEpochs
            [net, stateLBFGS] = lbfgsupdate(net, lossFcn, stateLBFGS);

            % Globalny błąd na całym zbiorze uczących i weryfikującym
            lossTrain = modelLoss(net, Xtrain, Ytrain);
            lossVal = modelLoss(net, Xval, Yval);
            lossTrainGlobal(epoch) = 2*extractdata(lossTrain);
            lossValGlobal(epoch) = 2*extractdata(lossVal);
            fprintf("%s: epoch %d/%d, MSEtrain=%e, MSEval=%e\n", ...
                algorithm, epoch, numEpochs, ...
                lossTrainGlobal(epoch), lossValGlobal(epoch));
        end
    case 'adam'
        for epoch = 1:numEpochs
            iteration = iteration + 1;
            [loss, gradients] = dlfeval(@modelLoss, net, Xtrain, Ytrain);
            [net, trailingAvg, trailingAvgSq] = adamupdate(net, gradients, ...
                trailingAvg, trailingAvgSq, iteration, learningRate);
            % Globalny błąd na całym zbiorze uczących i weryfikującym
            lossTrain = modelLoss(net, Xtrain, Ytrain);
            lossVal = modelLoss(net, Xval, Yval);
            lossTrainGlobal(epoch) = 2*extractdata(lossTrain);
            lossValGlobal(epoch) = 2*extractdata(lossVal);
            fprintf("%s: epoch %d/%d, MSEtrain=%e, MSEval=%e\n", ...
                algorithm, epoch, numEpochs, ...
                lossTrainGlobal(epoch), lossValGlobal(epoch));
        end
            fprintf("%s: epoch %d/%d, iteration %d, MSEtrain=%e, MSEval=%e\n",...
```

```
                algorithm, epoch, numEpochs, iteration, ...
                lossTrainGlobal(epoch), lossValGlobal(epoch));
end
time = toc;

Ymod_train = model(net, Xtrain);
Ymod_val = model(net, Xval);
y_hist_train = [Y_train(2); Y_train(1)];
y_hist_val   = [Y_val(2);   Y_val(1)];
Yrec_train = recursive_prediction(net, Xtrain, N_train, y_hist_train);
Yrec_val = recursive_prediction(net, Xval, N_val, y_hist_val);

% extractdata: dlarray -> zwykła macierz
Xtrain = extractdata(Xtrain);
Ytrain = extractdata(Ytrain);
Xval = extractdata(Xval);
Yval = extractdata(Yval);
Ymod_train = extractdata(Ymod_train);
Ymod_val = extractdata(Ymod_val);

MSEtrain = mse(Ytrain, Ymod_train);
MSEval = mse(Yval, Ymod_val);
MSErectrain = mse(Ytrain, Yrec_train);
Mserecval = mse(Yval, Yrec_val);

%%% forward model: wyjście modelu
function Ymod = model(net, X)
z = net.w1*X+net.w10;
z(1,:) = z(1,:)+net.w11(1)*sin(X(2,:));
v = tanh(z);
Ymod = net.w22(1)*cos(X(1,:)) + net.w22(2)*X(2,:) + net.w20 + net.w2*v;
end

%%5 funkcja predykcji rekurencyjnej
function [Yrec_train] = recursive_prediction(net, X, n, y_hist)
Yrec_train = zeros(n,1);
for j = 1:n
    x = X(:,j);
    x(3) = y_hist(1); % y(k-1)
    x(4) = y_hist(2); % y(k-2)
    y_mod = model(net, x);
    Yrec_train(j) = y_mod;
    y_hist = [y_mod; y_hist(1)];
end
end

%%% błąd modelu
function [loss, gradients] = modelLoss(net, X, Y)
Ypred = model(net, X);
loss = mse(Ypred, Y, 'DataFormat', 'CB');

if nargout > 1
    gradients = dlgradient(loss, net);
end
end
```

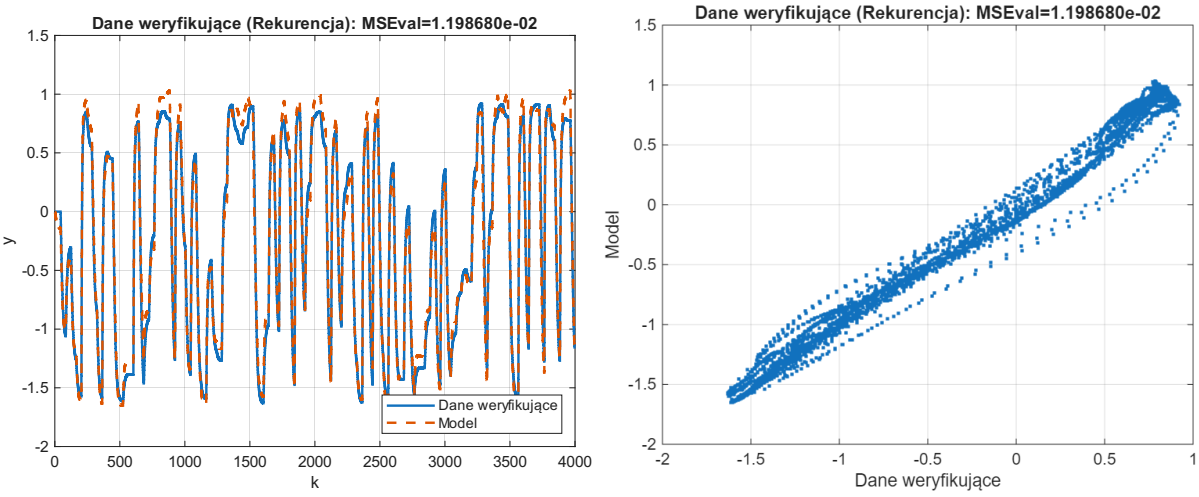
4.2. Dobór struktury i wyniki (LBFGS)

Przeprowadzono badania dla liczby neuronów ukrytych $K \in \{1, 2, 3, 4, 5, 6, 7, 10, 20, 60\}$ oraz liczby epok równej 200, gdyż eksperymentalnie sprawdzono, że ta liczba epok jest wystarczająca dla algorytmu LBFGS. Dla każdej struktury wykonano 5 prób uczenia, wybierając model o

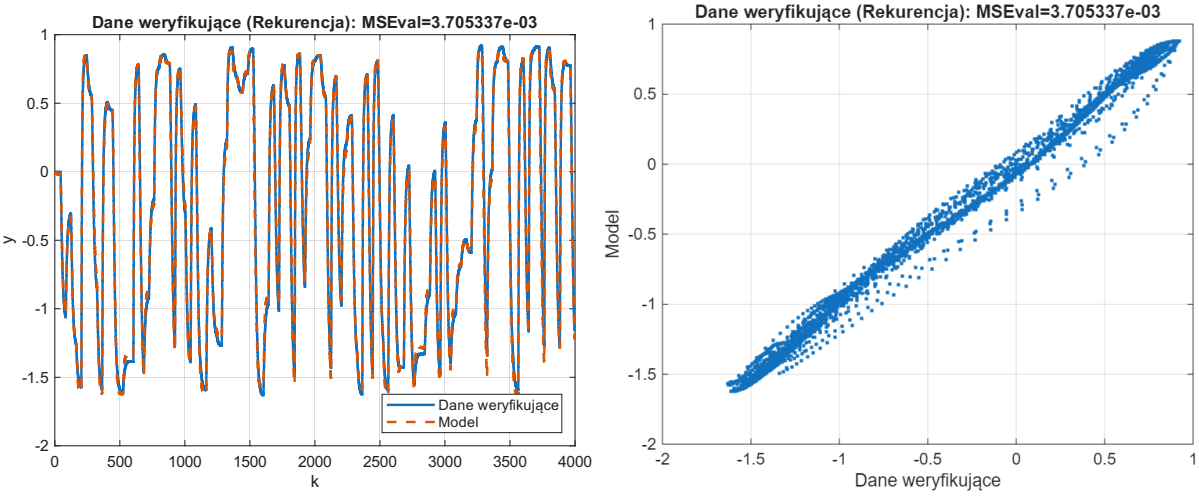
najmniejszym błędzie na zbiorze weryfikującym w trybie rekurencyjnym. Wyniki zestawiono w tab. 4.1. Natomiast na rys. 4.1-4.2 przedstawiono przebiegi dla zbioru weryfikującego w trybie rekurencyjnym w przypadku zbyt małej oraz bardzo dużej liczby neuronów ukrytych.

Tab. 4.1. Wpływ liczby neuronów ukrytych K na błędy treningowe oraz weryfikujące dla trybu bez rekurencji oraz z rekurencją (rek)

Liczba neuronów K	**MSE train**	**MSE train (rek)**	**MSE val**	**MSE val (rek)**
1	$4{,}593 \cdot 10^{-5}$	$1{,}003 \cdot 10^{-2}$	$6{,}026 \cdot 10^{-5}$	$1{,}199 \cdot 10^{-2}$
2	$2{,}819 \cdot 10^{-5}$	$1{,}018 \cdot 10^{-2}$	$3{,}245 \cdot 10^{-5}$	$1{,}070 \cdot 10^{-2}$
3	$2{,}105 \cdot 10^{-5}$	$4{,}273 \cdot 10^{-3}$	$2{,}409 \cdot 10^{-5}$	$4{,}909 \cdot 10^{-3}$
4	$1{,}217 \cdot 10^{-5}$	$2{,}826 \cdot 10^{-3}$	$1{,}513 \cdot 10^{-5}$	$3{,}041 \cdot 10^{-3}$
5	$1{,}807 \cdot 10^{-5}$	$3{,}100 \cdot 10^{-3}$	$2{,}187 \cdot 10^{-5}$	$3{,}797 \cdot 10^{-3}$
6	$2{,}214 \cdot 10^{-5}$	$4{,}186 \cdot 10^{-3}$	$2{,}531 \cdot 10^{-5}$	$4{,}450 \cdot 10^{-3}$
7	$1{,}046 \cdot 10^{-5}$	$2{,}993 \cdot 10^{-3}$	$1{,}200 \cdot 10^{-5}$	$3{,}700 \cdot 10^{-3}$
10	$1{,}526 \cdot 10^{-5}$	$3{,}343 \cdot 10^{-3}$	$1{,}847 \cdot 10^{-5}$	$3{,}643 \cdot 10^{-3}$
20	$1{,}264 \cdot 10^{-5}$	$2{,}616 \cdot 10^{-3}$	$1{,}450 \cdot 10^{-5}$	$3{,}282 \cdot 10^{-3}$
60	$1{,}288 \cdot 10^{-5}$	$2{,}423 \cdot 10^{-3}$	$1{,}736 \cdot 10^{-5}$	$3{,}705 \cdot 10^{-3}$



Rys. 4.1. Przebiegi dla zbioru weryfikującego w trybie rekurencji (Model $K = 1$)



Rys. 4.2. Przebiegi dla zbioru weryfikującego w trybie rekurencji (Model $K = 60$)

Na podstawie przeprowadzonych badań symulacyjnych oraz analizy błędów zestawionych w tabeli 4.1, sformułowano następujące wnioski dotyczące doboru struktury sieci neuronowej:

— **Optymalna liczba neuronów:** Najlepsze właściwości generalizacyjne w trybie rekurencyjnym uzyskano dla sieci z liczbą neuronów ukrytych $K = 4$. Model ten osiągnął najniższy błąd weryfikacji w trybie rekurencyjnym (MSE val rek) wynoszący $3{,}041 \cdot 10^{-3}$.
— **Wpływ złożoności na jakość modelu:** Dla małych wartości K $(1 - 2)$ widoczna jest tendencja spadkowa wartości błędów – sieć posiada zbyt małą liczbę parametrów, aby poprawnie odwzorować dynamikę procesu. Zwiększenie liczby neuronów powyżej $K = 3$ pozwala znacząco zredukować błąd.
— **Stabilizacja wyników i ryzyko przeuczenia:** Dalsze zwiększanie liczby neuronów ($K > 4$) nie prowadzi do istotnej poprawy wyników weryfikacji rekurencyjnej (błędy oscylują wokół wartości $3{,}6 \cdot 10^{-3} - 4{,}5 \cdot 10^{-3}$). W przypadku większych struktur (np. $K = 60$) nie obserwuje się zysku w dokładności, co przy zwiększonym koszcie obliczeniowym uzasadnia wybór mniejszej struktury.
— **Różnice między trybami predykcji:** Błędy w trybie rekurencyjnym są o dwa rzędy wielkości wyższe niż błędy w trybie bez rekurencji (jednokrokowym). Jest to zjawisko naturalne, wynikające z kumulacji błędów predykcji w kolejnych krokach czasowych, co czyni wskaźnik *MSE val (rek)* kluczowym kryterium przy wyborze modelu.

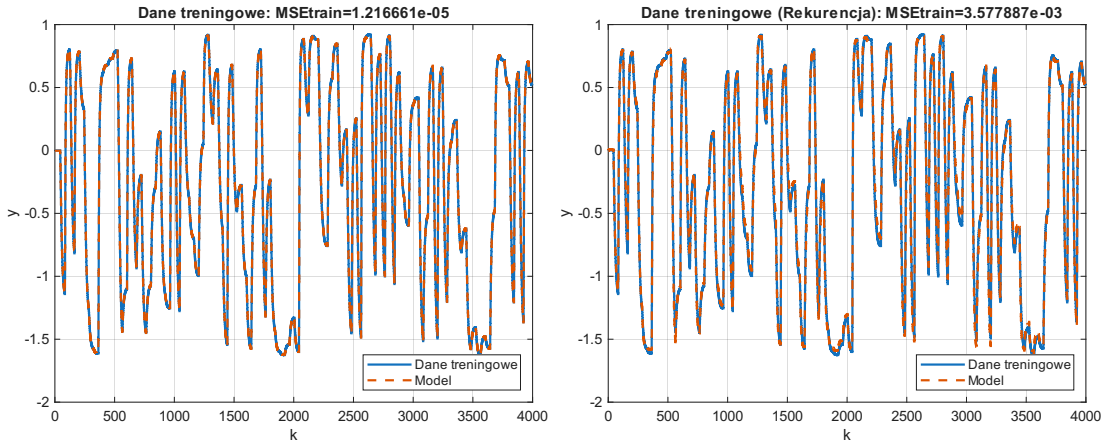
Biorąc pod uwagę powyższe przesłanki, do dalszych badań i zastosowań wybrano strukturę o $K = 4$ neuronach ukrytych, jako kompromis pomiędzy dokładnością odwzorowania dynamiki a złożonością modelu.

4.3. Porównanie najlepszego modelu w trybie bez rekurencji i z rekurencją

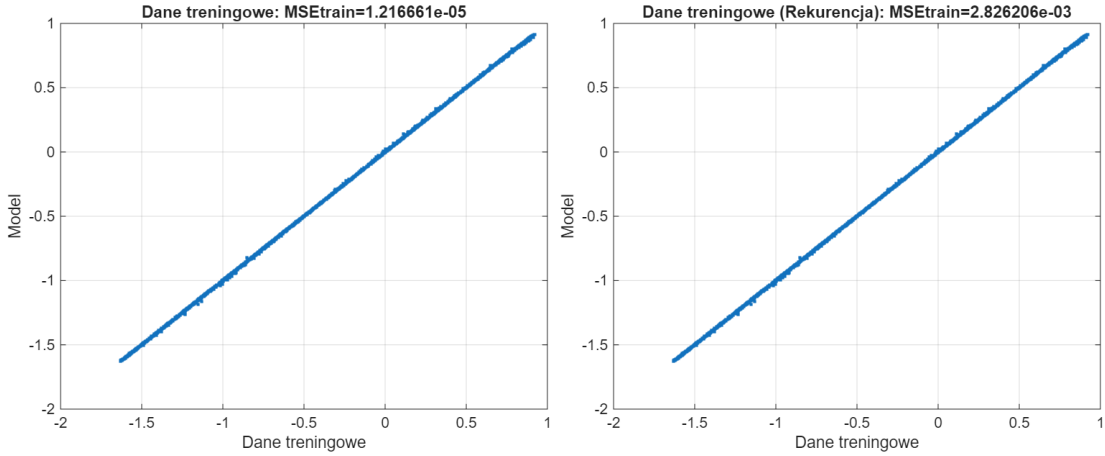
Na rys. 4.3 oraz rys. 4.5 zaprezentowano przebiegi czasowe wyjścia wybranego modelu ($K = 4$) odpowiednio dla zbioru treningowego i walidacyjnego. Wykresy te zestawiają działanie modelu w trybie bez rekurencji oraz z rekurencją. Analogiczne porównanie w formie wykresów punktowych przedstawiono na rys. 4.4–4.6.

Analiza otrzymanych wyników pozwala stwierdzić, że w trybie bez rekurencji model poprawnie odwzorowuje dane rzeczywiste. Potwierdza to pokrywanie się przebiegów czasowych oraz koncentracja punktów wokół prostej $y = x$ na wykresach punktowych.

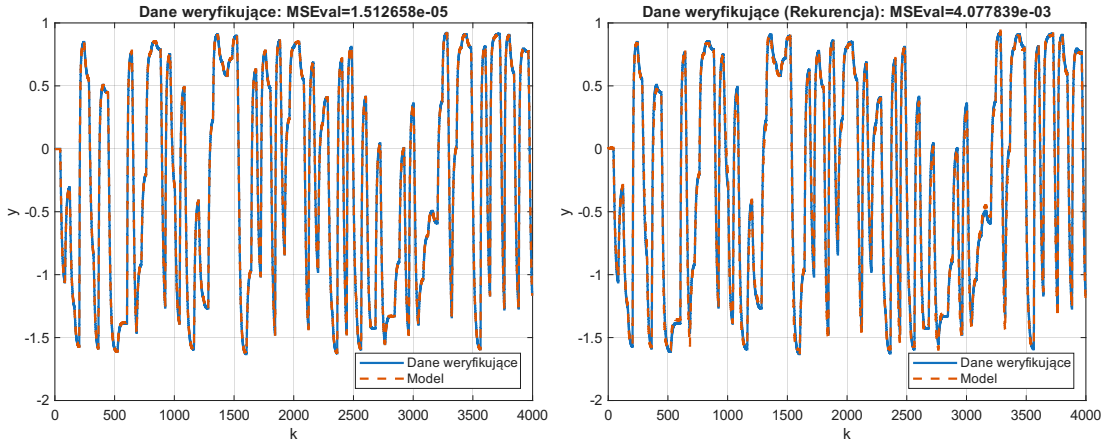
W trybie rekurencyjnym wysoka dokładność odwzorowania zachowana jest głównie dla zbioru treningowego. W przypadku danych weryfikujących, na wykresie punktowym widoczne są pewne rozbieżności względem prostej $y = x$. Mimo to, model zachowuje zadowalającą skuteczność, co potwierdza analiza przebiegów czasowych, chociaż precyzja odwzorowania jest niższa w porównaniu do trybu bez rekurencji. Błąd rekurencyjny jest większy niż w przypadku sieci ELM. Zjawisko to może wynikać z tego, że dodatkowe człony $cos(x) oraz sin(x)$, wykorzystywane w hybrydowej sieci neuronowej, powodują pogorszenie działania sieci neuronową, zamiast je polepszać.



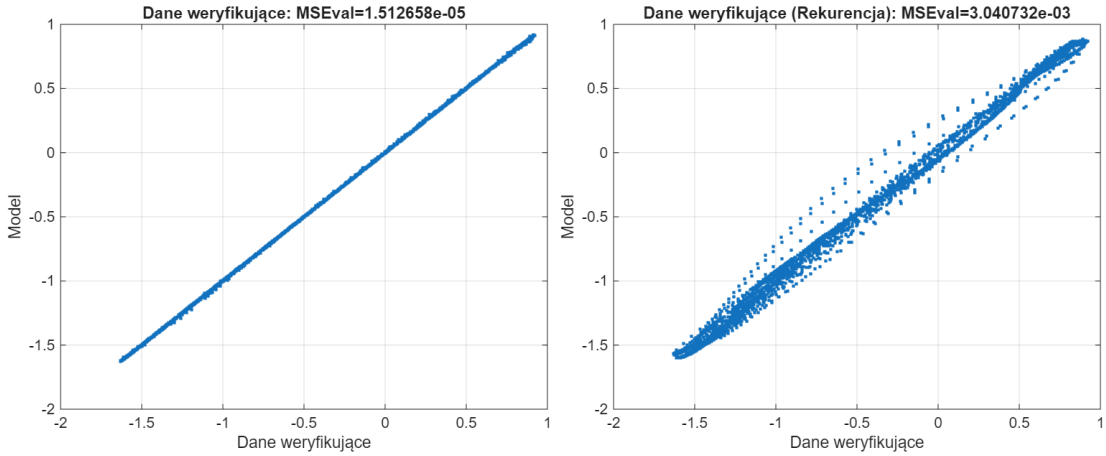
Rys. 4.3. Przebiegi czasowe dla zbioru treningowego w trybie bez rekurencji (lewa), z rekurencją (prawa)



Rys. 4.4. Wykresy punktowe dla zbioru treningowego w trybie bez rekurencji (lewa), z rekurencją (prawa)



Rys. 4.5. Przebiegi czasowe dla zbioru weryfikującego w trybie bez rekurencji (lewa), z rekurencją (prawa)



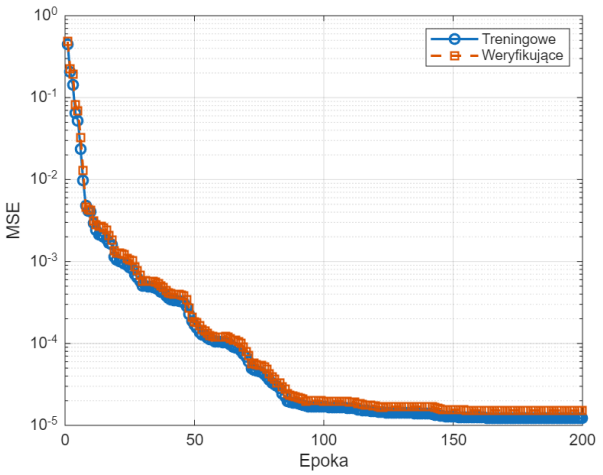
Rys. 4.6. Wykresy punktowe dla zbioru weryfikującego w trybie bez rekurencji (lewa), z rekurencją (prawa)

4.4. Porównanie z algorytmem Adam

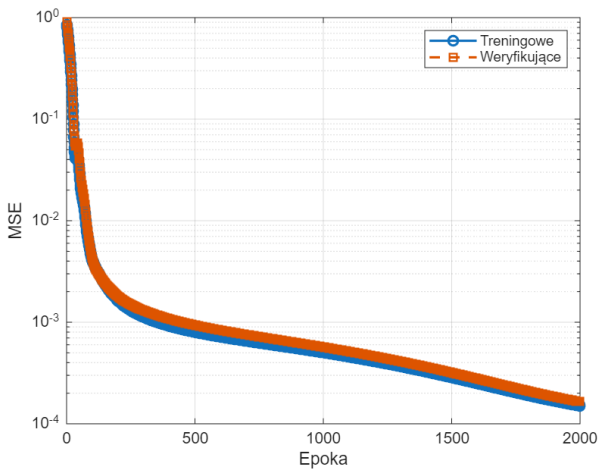
Dla wybranej struktury ($K = 4$) porównano efektywność algorytmu LBFGS oraz Adam. Oba algorytmy optymalizowały tę samą funkcję kosztu na pełnym zbiorze danych. Dla algorytmu Adam przetestowano różne wartości parametru learning rate i do porównania wybrano wartość równą 0,01. W tab. 4.2 porównano oba algorytmy. Natomiast na rys. 4.7- 4.8 przedstawiono porównanie wykresów błędu na zbiorze treningowym oraz weryfikującym w trakcie uczenia. Z uwagi na zbliżony błąd na zbiorze rekurencyjnym, zdecydowano się nie zamieszczać wykresów przebiegów czasowych ani wykresów punktowych dla algorytmu ADAM, gdyż są one zbliżone do wcześniej zamieszczanych wykresów algorytmu LBFGS.

Tab. 4.2. Porównanie wyników dla algorytmów LBFGS oraz Adam przy architekturze sieci $K = 4$

Algorytm	**Iteracje**	**Czas [s]**	**MSE train**	**MSE train (rek)**	**MSE val**	**MSE val (rek)**
LBFGS	200	7,38	$1{,}217 \cdot 10^{-5}$	$2{,}826 \cdot 10^{-3}$	$1{,}513 \cdot 10^{-5}$	$3{,}041 \cdot 10^{-3}$
Adam	2000	10,73	$1{,}509 \cdot 10^{-4}$	$3{,}578 \cdot 10^{-3}$	$1{,}647 \cdot 10^{-4}$	$4{,}078 \cdot 10^{-3}$



Rys. 4.7. MSE podczas uczenia (algorytm LBFGS)



Rys. 4.8. MSE podczas uczenia (algorytm Adam)

Analiza przedstawionych wyników jednoznacznie wskazuje na przewagę algorytmu LBFGS nad optymalizatorem Adam w badanej architekturze sieci. LBFGS pozwala uzyskać wyższą

precyzję Oznacza to, że model trenowany tą metodą lepiej odwzorowuje dane zarówno w procesie uczenia, jak i weryfikacji.

Równie istotna jest przewaga w wydajności obliczeniowej. Algorytm LBFGS osiągnął zadowalające wyniki znacznie szybciej, potrzebując do tego dziesięciokrotnie mniej iteracji oraz krótszego czasu pracy. Wykresy potwierdzają, że zbieżność tej metody jest o wiele szybsza, podczas gdy Adam, mimo dłuższego działania, nie zdołał zredukować błędu do równie niskiego poziomu.

5. Modelowanie liniowe procesu

W celu stworzenia punktu odniesienia dla modeli neuronowych przeprowadzono identyfikację obiektu przy użyciu modelu liniowego. Do wyznaczenia parametrów modelu wykorzystano metodę najmniejszych kwadratów.

5.1. Struktura modelu i algorytm uczenia

Przyjęto strukturę modelu liniowego typu ARX o rzędzie dynamiki zgodnym z rzędem symulatora procesu oraz modeli neuronowych. Model opisuje zależność wyjścia $y(k)$ od opóźnionych wartości sterowania u oraz wyjścia y:

$$y(k) = w_1 u(k-3) + w_2 u(k-4) + w_3 y(k-1) + w_4 y(k-2) \tag{5.1}$$

gdzie $w_1, \ldots, w_4$ to wagi modelu wyznaczane w procesie optymalizacji.

Do znalezienia optymalnych wartości wag posłużono się metodą najmniejszych kwadratów, rozwiązując równanie za pomocą operatora „\". Następnie przy użyciu otrzymanych wag wyznaczono predykcje jednokrokową, oraz predykcje rekurencyjną dla zbioru treningowego oraz weryfikującego.

Kod realizujący wyznaczanie modelu oraz obliczanie błędów predykcji przedstawiono na listingu 5.1.

Listing 5.1. Identyfikacja i weryfikacja modelu liniowego

```
load('data/data_train.mat');
load('data/data_val.mat');

Y_train = Y_train(:);
Y_val = Y_val(:);

% Macierz X_train zawiera kolumny [u(k-3); u(k-4); y(k-1); y(k-2)]
M = X_train';
% Wyznaczenie wag metoda najmniejszych kwadratow
W = M\Y_train;

% Predykcja jednokrokowa
Ymod_train = model(W, X_train');
Ymod_val = model(W, X_val');
MSEtrain = mean((Y_train - Ymod_train).^2);
MSEval = mean((Y_val - Ymod_val).^2);

% Predykcja rekurencyjna
y_hist_train = [Y_train(2); Y_train(1)]; % inicjalizacja y(k-1), y(k-2)
Yrec_train = recursive_prediction(W, X_train', N_train, y_hist_train);
MSErectrain = mean((Y_train - Yrec_train).^2);

y_hist_val = [Y_val(2); Y_val(1)];
Yrec_val = recursive_prediction(W, X_val', N_val, y_hist_val);
MSErecval = mean((Y_val - Yrec_val).^2);

% Funkcja obliczajaca wyjscie mod. liniowego
function [Y] = model(w, X)
```

```
    Y = w(1) * X(:,1) + w(2) * X(:,2) + w(3) * X(:,3) + w(4) * X(:,4);
end

% Funkcja do predykcji rekurencyjnej
function [Yrec] = recursive_prediction(w, X, n, y_hist)
    Yrec = zeros(n,1);
    for j = 1:n
        x = X(j, :);
        x(3) = y_hist(1); % y(k-1)
        x(4) = y_hist(2); % y(k-2)
        y_mod = model(w, x);
        Yrec(j) = y_mod;
        y_hist(2) = y_hist(1);
        y_hist(1) = y_mod;
    end
end
```

5.2. Wyniki i weryfikacja modelu

Jakość modelu oceniono na podstawie błędu średniokwadratowego (MSE) dla zbioru uczącego i weryfikującego w dwóch trybach pracy:

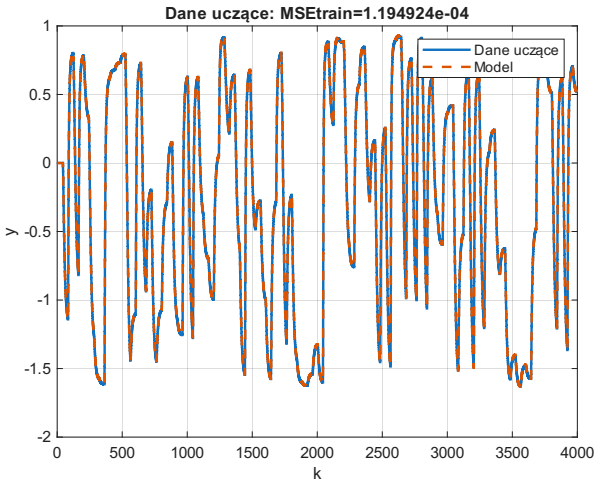
— **Bez rekurencji :** model korzysta z rzeczywistych, zmierzonych wartości wyjść procesu $y(k-1), y(k-2)$.
— **Z rekurencją :** model korzysta z własnych, wcześniej wyznaczonych wyjść.

Uzyskane wartości błędów zestawiono w tab. 5.1.

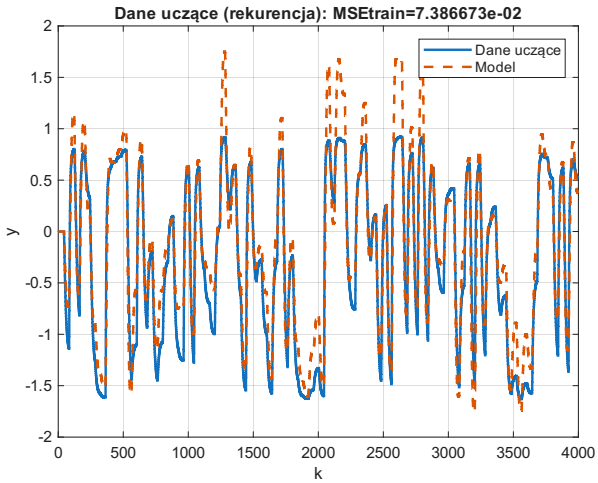
Tab. 5.1. Błędy MSE dla modelu liniowego

Zbiór danych	Tryb bez rekurencji	Tryb rekurencyjny
Uczący	$1{,}1949 \cdot 10^{-4}$	$7{,}3867 \cdot 10^{-2}$
Weryfikujący	$1{,}4561 \cdot 10^{-4}$	$9{,}2424 \cdot 10^{-2}$

Na rysunkach 5.3 oraz 5.6 przedstawiono porównanie przebiegów wyjścia modelu neuronowego z danymi rzeczywistymi zarówno w trybie jednokrokowym jak i rekurencyjnym. Jako dodatkową weryfikację jakości modelu umieszczono wykresy punktowe przedstawione na rys. 5.7 oraz 5.8.

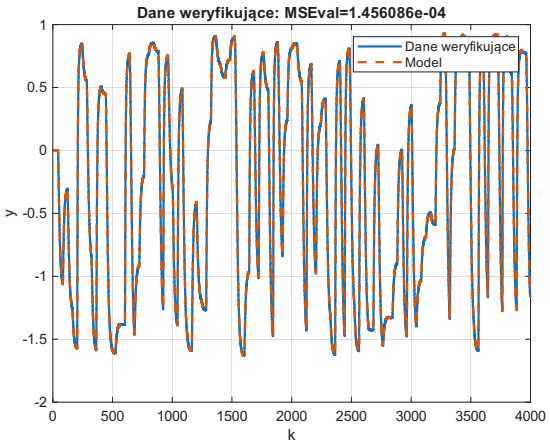


Rys. 5.1. Tryb bez rekurencji

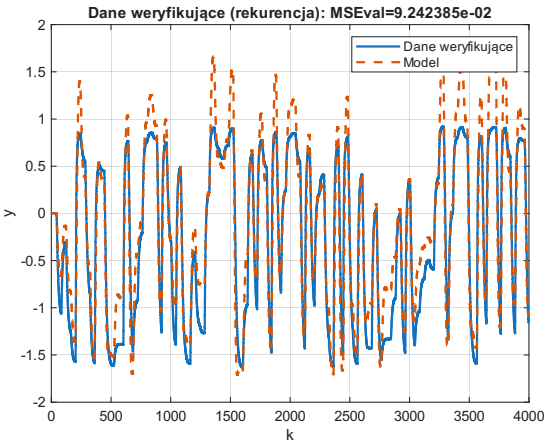


Rys. 5.2. Tryb rekurencyjny

Rys. 5.3. Porównanie przebiegów czasowych dla danych uczących

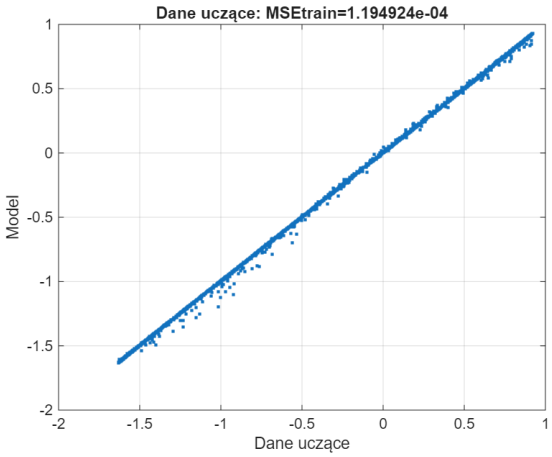


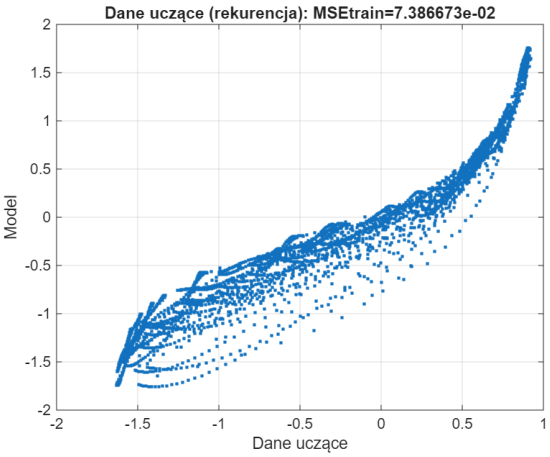
Rys. 5.4. Tryb bez rekurencji



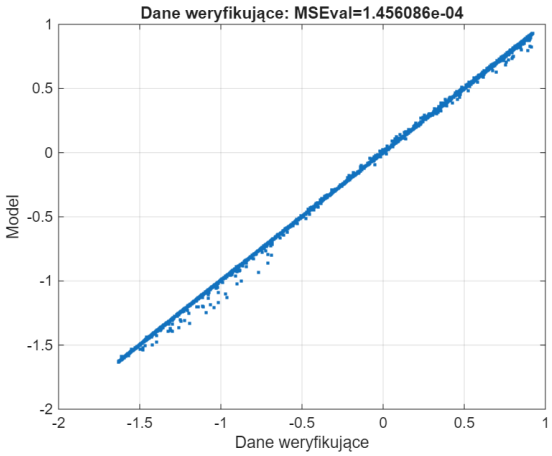
Rys. 5.5. Tryb rekurencyjny

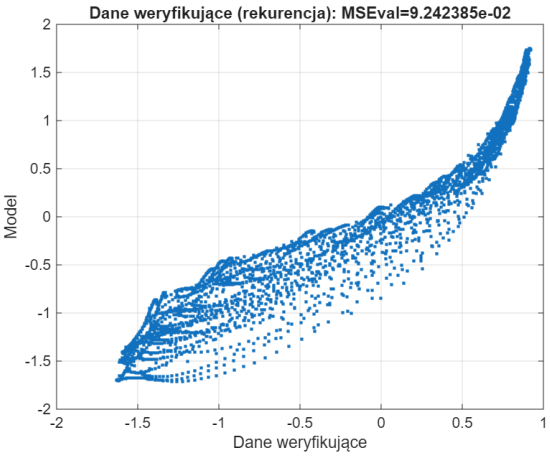
Rys. 5.6. Porównanie przebiegów czasowych dla danych weryfikujących





Rys. 5.7. Wykresy punktowe dla danych uczących





Rys. 5.8. Wykresy punktowe dla danych weryfikujących

Analiza przeprowadzonych symulacji oraz uzyskanych wartości błędów pozwala na sformułowanie następujących wniosków:

1. **Wysoka jakość predykcji jednokrokowej:** W trybie bez rekurencji model liniowy osiąga bardzo małe błędy ($MSE \approx 10^{-4}$), a punkty na wykresach punktowych układają się niemal idealnie na prostej $y = x$. Oznacza to, że metoda najmniejszych kwadratów poprawnie zidentyfikowała zależności między próbkami.
2. **Pogorszenie w trybie rekurencyjnym:** Przejście w tryb z rekurencją powoduje wzrost błędu średniokwadratowego o dwa rzędy wielkości ($MSE \approx 10^{-2}$). Na wykresach czasowych widoczne są odchylenia trajektorii modelu od obiektu, a chmura punktów na wykresach punktowych znacznie odbiega od prostej $y = x$.
3. **Przyczyna błędów:** Głównym powodem pogorszenia jakości w trybie rekurencyjnym jest nieliniowy charakter badanego procesu. Model liniowy ARX jest jedynie aproksymacją, która nie potrafi odwzorować silnej nieliniowości obiektu. W trybie rekurencyjnym błędy te kumulują się w każdym kroku, ponieważ model korzysta z własnych, niedokładnych predykcji z chwil poprzednich.

6. Regulacja procesu

Ostatnim etapem projektu było zaimplementowanie oraz przetestowanie algorytmu regulacji predykcyjnej dla badanego procesu nieliniowego, zarówno dla sieci neuronowej ELM jak i hybrydowej. Zgodnie z wymaganiami projektowymi zastosowano algorytm z Nieliniową Predykcją i Linearyzacją (NPL).

6.1. Implementacja algorytmu NPL

Algorytm regulacji został zaimplementowany w następujących krokach:

6.1.1. Pomiar wyjścia procesu

W każdej iteracji algorytmu k, pierwszym krokiem jest pobranie aktualnej wartości wyjścia procesu $y(k)$. Symulacja obiektu rzeczywistego realizowana jest za pomocą funkcji `proces10_symulator`, która zwraca aktualny pomiar oraz zaktualizowany wektor stanu:

```
% Pomiar aktualnego wyjscia procesu y(k) oraz stanu x(k)
[Y(k), X(k)] = proces10_symulator(U(k-3), U(k-4), X(k-1), X(k-2));
```

6.1.2. Obliczenie wyjścia modelu i estymacja zakłócenia

Kolejnym krokiem jest estymacja niemierzalnego zakłócenia $d(k)$. W pierwszej kolejności obliczane jest wyjście modelu $\hat{y}(k)$ (oznaczone w kodzie jako `Yelm` lub `Ydl`) na podstawie znanych przeszłych wartości wejść i wyjść. Następnie obliczana jest wartość zmiennej `dmod` ($d(k) = y(k) - \hat{y}(k)$).

W zależności od wybranej struktury modelu (sieć ELM lub model hybrydowy), algorytm wykonuje odpowiednie obliczenia:

```
switch model
    case 'elm'
        % Obliczenie wyjscia modelu ELM
        % Wektor wejsciowy: [u(k-3), u(k-4), y(k-1), y(k-2)]
        Yelm(k) = model_elm(elm.w10, elm.w1, elm.w20, elm.w2, ...
                            [U(k-3); U(k-4); Y(k-1); Y(k-2)]);

        % Obliczenie bledu modelowania (zaklocenia)
        dmod = Y(k) - Yelm(k);

    case 'hybrydowy'
        % Obliczenie wyjscia modelu hybrydowego (Deep Learning)
        Ydl(k) = model_dl(dl_tbx.net, ...
                          [U(k-3); U(k-4); Y(k-1); Y(k-2)]);

        % Obliczenie bledu modelowania (zaklocenia)
        dmod = Y(k) - Ydl(k);
end
```

6.1.3. Wyznaczenie odpowiedzi swobodnej

Kolejnym etapem jest obliczenie odpowiedźi swobodnej $\boldsymbol{Y}^0$. Procedura ta odbywa się rekurencyjnie na horyzoncie predykcji N. Do predykcji wykorzystywany jest wybrany model (ELM lub hybrydowy):

```
% Inicjalizacja wektora odpowiedzi swobodnej
Y0 = zeros(N, 1);

switch model
    case 'elm'
        % Pierwsze dwa kroki predykcji
        Y0(1) = model_elm(elm.w10, elm.w1, elm.w20, elm.w2, ...
                [U(k-2); U(k-3); Yelm(k); Yelm(k-1)]) + dmod;
        Y0(2) = model_elm(elm.w10, elm.w1, elm.w20, elm.w2, ...
                [U(k-1); U(k-2); Y0(1); Yelm(k)]) + dmod;

        % Kolejne kroki p od 3 do N
        for p = 3:N
            Y0(p) = model_elm(elm.w10, elm.w1, elm.w20, elm.w2, ...
                    [U(k-1); U(k-1); Y0(p-1); Y0(p-2)]) + dmod;
        end

    case 'hybrydowy'
        % Analogiczne obliczenia dla modelu hybrydowego
        Y0(1) = model_dl(dl_tbx.net, ...
                [U(k-2); U(k-3); Ydl(k); Ydl(k-1)]) + dmod;
        Y0(2) = model_dl(dl_tbx.net, ...
                [U(k-1); U(k-2); Y0(1); Ydl(k)]) + dmod;

        for p = 3:N
            Y0(p) = model_dl(dl_tbx.net, ...
                    [U(k-1); U(k-1); Y0(p-1); Y0(p-2)]) + dmod;
        end
end
```

6.1.4. Linearyzacja modelu w punkcie pracy

W celu wyznaczenia macierzy dynamicznej algorytmu NPL, nieliniowy model neuronowy poddawany jest linearyzacji w bieżącym punkcie pracy. Pozwala to na uzyskanie parametrów lokalnego modelu liniowego A i B:

```
if strcmp(model, 'elm')
    [a1, a2, b3, b4] = linearize_model_elm(elm, U(k-3), U(k-4), ...
                                           Yelm(k-1), Yelm(k-2));
else
    [a1, a2, b3, b4] = linearize_model_dl(dl_tbx, U(k-3), U(k-4), ...
                                          Ydl(k-1), Ydl(k-2));
end
```

Obie funkcje linearyzujące działają na tej samej zasadzie numerycznej. Zmieniają wyjście modelu po wprowadzeniu bardzo małej zmiany na wejściu (parametr $\delta = 10^{-5}$).

Implementacja dla modelu ELM:

```
function [a1, a2, b3, b4] = linearize_model_elm(elm, ukm3, ukm4, ...
                                                ykm1, ykm2)
    delta = 1e-5;
    % Obliczenie punktu bazowego
    f0 = model_elm(elm.w10, elm.w1, elm.w20, elm.w2, [ukm3; ukm4; ...
                                                      ykm1; ykm2]);

    f_ukm3 = model_elm(elm.w10, elm.w1, elm.w20, elm.w2, [ukm3 + delta; ukm4; ...
```

```
                                                    ykm1; ykm2]);
    b3 = (f_ukm3 - f0) / delta;

    f_ukm4 = model_elm(elm.w10, elm.w1, elm.w20, elm.w2, [ukm3; ukm4 + delta; ...
                                                    ykm1; ykm2]);
    b4 = (f_ukm4 - f0) / delta;

    f_ykm1 = model_elm(elm.w10, elm.w1, elm.w20, elm.w2, [ukm3; ukm4; ...
                                                    ykm1 + delta; ykm2]);
    a1 = - (f_ykm1 - f0) / delta;

    f_ykm2 = model_elm(elm.w10, elm.w1, elm.w20, elm.w2, [ukm3; ukm4; ...
                                                    ykm1; ykm2 + delta]);
    a2 = - (f_ykm2 - f0) / delta;
end
```

Implementacja dla modelu hybrydowego (Deep Learning) jest analogiczna, różni się jedynie sposobem wywołania predykcji z sieci:

```
function [a1, a2, b3, b4] = linearize_model_dl(dl_tbx, ukm3, ukm4, ykm1, ykm2)
    delta = 1e-5;
    f0 = model_dl(dl_tbx.net, [ukm3; ukm4; ykm1; ykm2]);

    f_ukm3 = model_dl(dl_tbx.net, [ukm3 + delta; ukm4; ...
                        ykm1; ykm2]);
    b3 = (f_ukm3 - f0) / delta;

    f_ukm4 = model_dl(dl_tbx.net, [ukm3; ukm4 + delta; ...
                        ykm1; ykm2]);
    b4 = (f_ukm4 - f0) / delta;

    f_ykm1 = model_dl(dl_tbx.net, [ukm3; ukm4; ...
                        ykm1 + delta; ykm2]);
    a1 = - (f_ykm1 - f0) / delta;

    f_ykm2 = model_dl(dl_tbx.net, [ukm3; ukm4; ...
                        ykm1; ykm2 + delta]);
    a2 = - (f_ykm2 - f0) / delta;
end
```

6.1.5. Konstrukcja macierzy dynamicznej

Na podstawie parametrów zlinearyzowanych wyznaczane są współczynniki odpowiedzi skokowej s_i, które służą do budowy macierzy dynamicznej $\boldsymbol{M}$. Macierz ta opisuje wpływ przyszłych przyrostów sterowania na wyjście procesu:

```
% Wyznaczenie odpowiedzi skokowej s(k)
S = zeros(1, N);
A = [a1, a2];
B = [0, 0, b3, b4]; % Uwzglednienie opoznienia

for q = 1:N
    for i = 1:min(q, nB)
        S(q) = S(q) + B(i);
    end
    for i = 1:min(q - 1, nA)
        S(q) = S(q) - A(i) * S(q - i);
    end
end

% Budowa macierzy M
M = zeros(N, Nu);
```

```
for row = 1:N
    for col = 1:Nu
        if row >= col
            M(row, col) = S(row - col + 1);
        end
    end
end
```

6.1.6. Wyznaczenie prawa sterowania

Ostatnim krokiem jest obliczenie optymalnego przyrostu sterowania $\Delta u(k)$. Wykorzystuje się do tego analityczne rozwiązanie minimalizowanego wskaźnika. Obliczona wartość sterowania $u(k)$ jest następnie ograniczana do $(u_{min} = -1, u_{max} = 1)$:

```
% Obliczenie macierzy wzmocnienia K
K = (M' * M + lambda * eye(Nu)) \ (M');

% Obliczenie wektora przyrostow sterowania dU
% yzad(k) to wartosc zadana w chwili k
dU = K * (yzad(k) * ones(N, 1) - Y0);

% Wyznaczenie i nasycenie sygnalu sterujacego u(k)
U(k) = max(min(U(k - 1) + dU(1), u_max), u_min);
```

6.2. Dobór parametrów regulatora

Dobór parametrów regulatora NPL przeprowadzono początkowo dla nieliniowego modelu ELM, ponieważ eksperymenty wykazały, że osiąga on lepsze wyniki niż sieć hybrydowa. W trakcie kolejnych symulacji modyfikowano horyzonty predykcji N i sterowania N_{u}, a także wartość współczynnika kary λ. Jakość regulacji oceniano na podstawie stabilności sygnału wyjściowego, przeregulowania oraz szybkości reakcji układu na zmiany wartości zadanej. Dodatkowo uwzględniono dynamikę zmian sygnału sterującego.

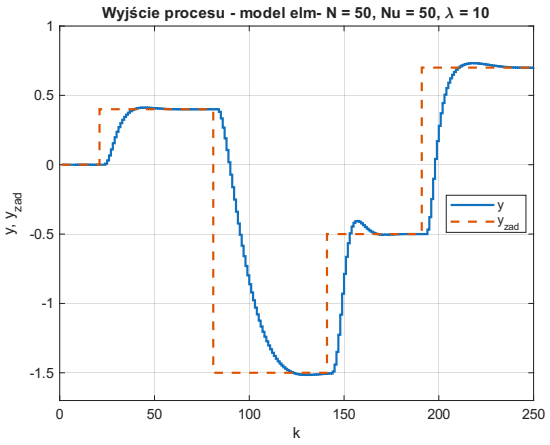
Dostrajanie parametrów regulatora podzielono na cztery etapy. Badania zrealizowano dla trajektorii obejmującej szeroki zakres zmian sygnału wyjściowego. Sekwencja zmian wartości zadanej y_{zad} była następująca:

$$y^{\mathrm{zad}}(k) = \begin{cases} 0 & \text{dla } k \in \langle 1, 20\rangle \\ 0.4 & \text{dla } k \in \langle 21, 80\rangle \\ -1.5 & \text{dla } k \in \langle 81, 140\rangle \\ -0.5 & \text{dla } k \in \langle 141, 190\rangle \\ 0.7 & \text{dla } k \in \langle 191, 250\rangle \end{cases} \tag{6.1}$$

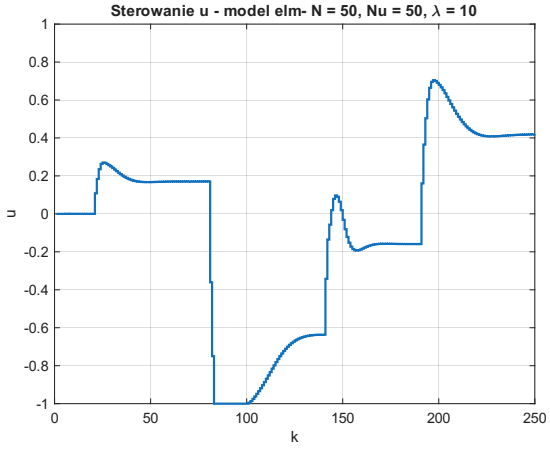
6.2.1 Etap 1: Wartości początkowe – $N = N_{\mathrm{u}} = 50$, $\lambda = 10$

W pierwszym kroku przyjęto duże wartości horyzontów predykcji i sterowania, co skutkuje większą macierzą dynamiczną $\boldsymbol{M}$. Takie ustawienie zapewnia najwyższą dokładność predykcji, ale może prowadzić do nadmiernej złożoności obliczeniowej z powodu zwiększonych rozmiarów macierzy.

Przebiegi wyjścia oraz sterowania dla początkowych parametrów algorytmu przedstawiono na rys. 6.1 oraz 6.2.



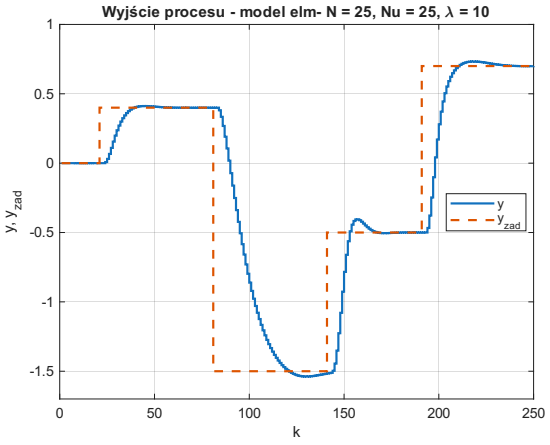
Rys. 6.1. Przebieg wyjścia i wartości zadanej dla parametrów początkowych ($N = 50$)



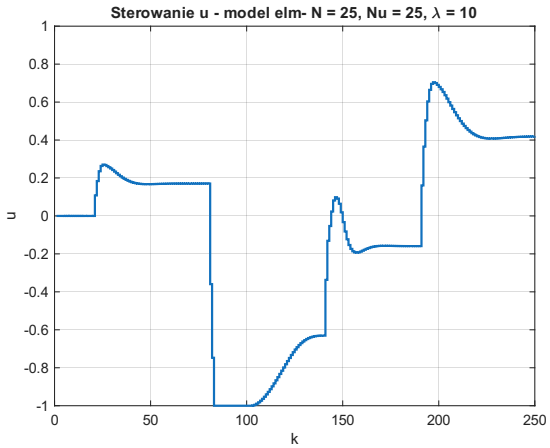
Rys. 6.2. Przebieg sygnału sterowania dla parametrów początkowych

6.2.2 Etap 2: Redukcja horyzontów – $N = N_{\mathrm{u}} = 25$, $\lambda = 1$

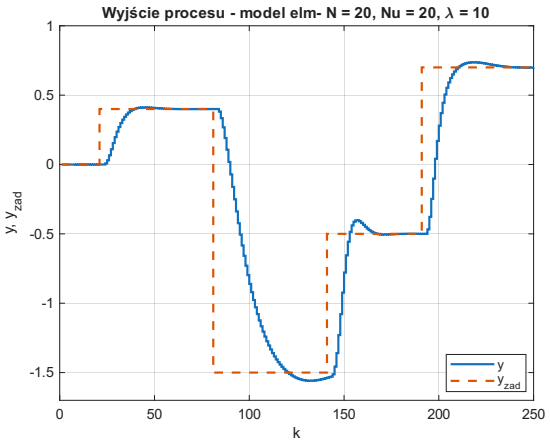
Redukcja horyzontów miała na celu możliwie największe uproszczenie struktury regulatora, bez utraty jakości, poprzez dobranie możliwie małego horyzontu predykcji. Prowadzi to do znacznie mniej wymagających obliczeń. Zdecydowano się na wartość 25, gdyż poniżej tej wartości jakość regulacji ulegała pogorszeniu, co jest widoczne na rys. 6.3–6.6 (szczególnie w przypadku skoku wartości zadanej do wartości -1.5).



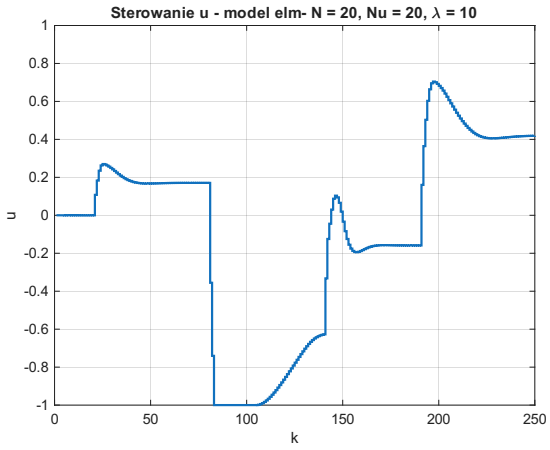
Rys. 6.3. Wyjście dla horyzontu predykcji $N = 25$



Rys. 6.4. Sygnał sterujący dla $N = 25$



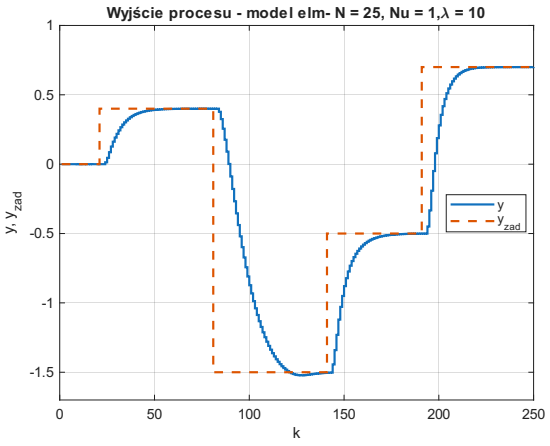
Rys. 6.5. Wyjście dla horyzontu predykcji $N = 20$



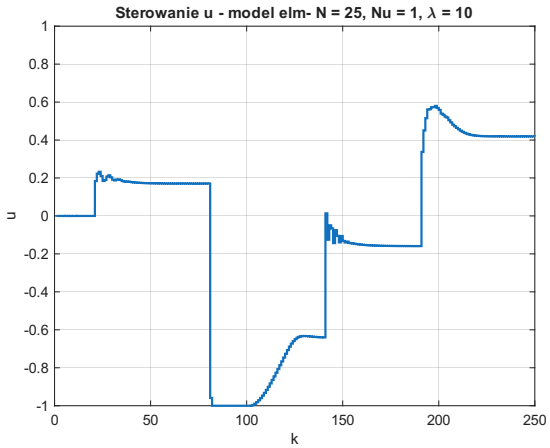
Rys. 6.6. Sygnał sterujący dla $N = 20$

6.2.3 Etap 3: Zmniejszanie horyzontu sterowania - $N = 25$, $N_u = 4$, $\lambda = 10$

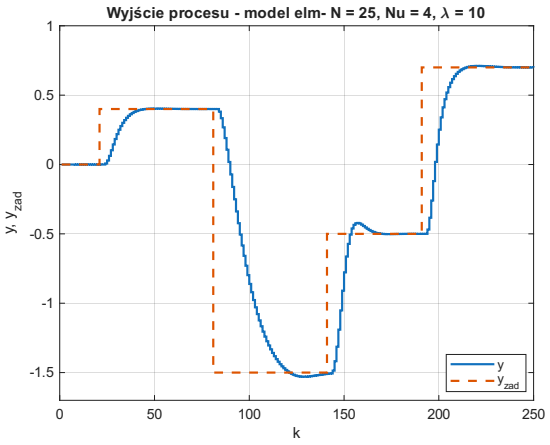
W tym etapie zmniejszono horyzont sterowania, aby poprawić jakość regulacji regulatora. Aby znaleźć najmniejszy możliwy horyzont sterowania, strojenie rozpoczęto od wartości $N_u = 1$, następnie ją zwiększano aż do osiągnięcia satysfakcjonującej jakości regulacji.
Przebiegi wyjścia oraz sterowania poszczególnych etapów zostały zaprezentowane na rys. 6.7–6.12.



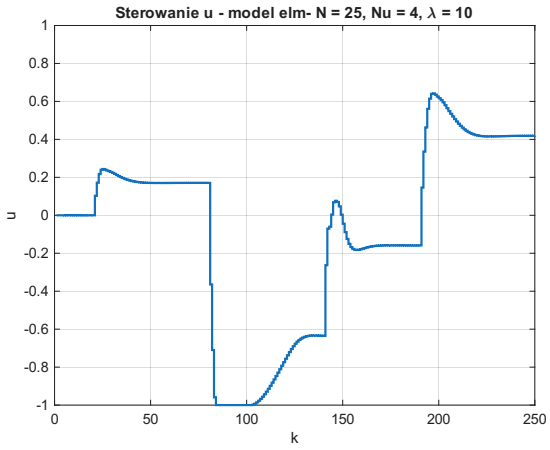
Rys. 6.7. Wyjście dla horyzontu sterowania $N_u = 1$



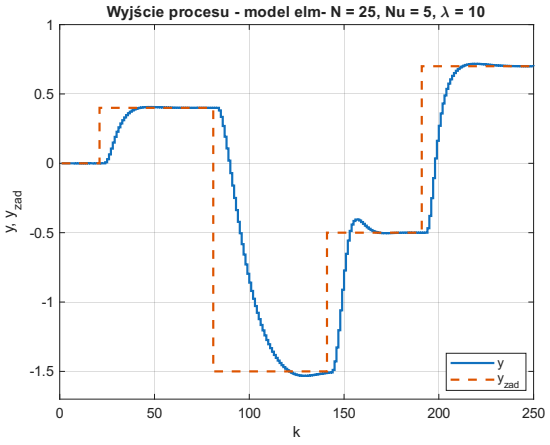
Rys. 6.8. Sygnał sterujący dla $N_u = 1$



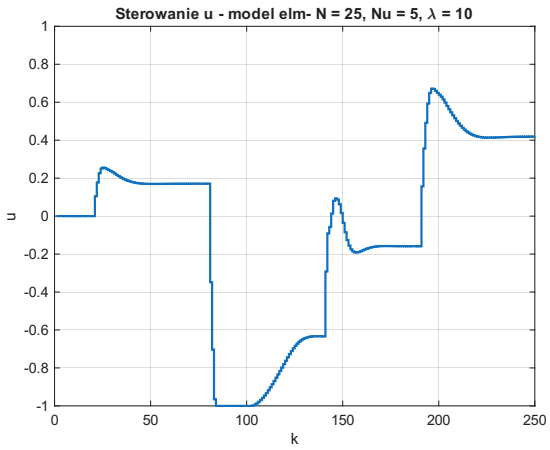
Rys. 6.9. Wyjście dla horyzontu sterowania $N_{\mathrm{u}} = 4$



Rys. 6.10. Sygnał sterujący dla $N_{\mathrm{u}} = 4$



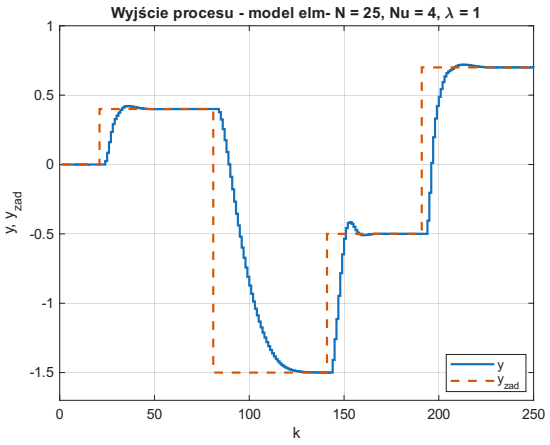
Rys. 6.11. Wyjście dla horyzontu sterowania $N_{\mathrm{u}} = 5$



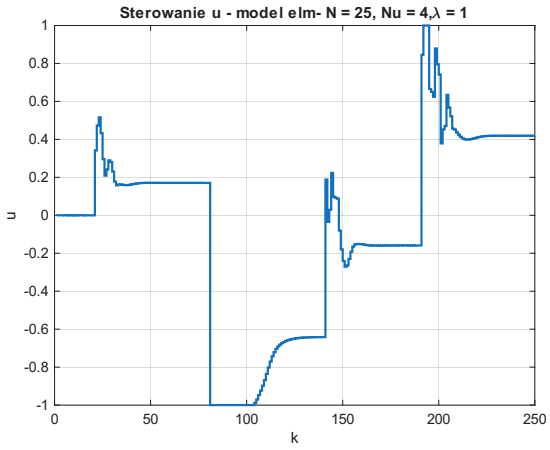
Rys. 6.12. Sygnał sterujący dla $N_{\mathrm{u}} = 5$

6.2.4 Etap 4: Zmiana parametru kary λ – $N = 25$, $N_{\mathrm{u}} = 4$, $\lambda = 5$

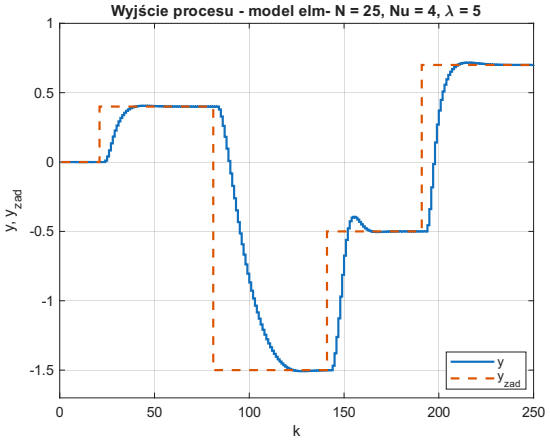
Ostatni etap obejmuje dostrojenie parametru λ, który wpływa na kompromis pomiędzy szybkością regulacji a dynamiką zmian sygnału sterującego.
Przebiegi wyjścia oraz sterowania dla różnych wartości parametru λ przedstawiono na wykresach 6.13–6.18.



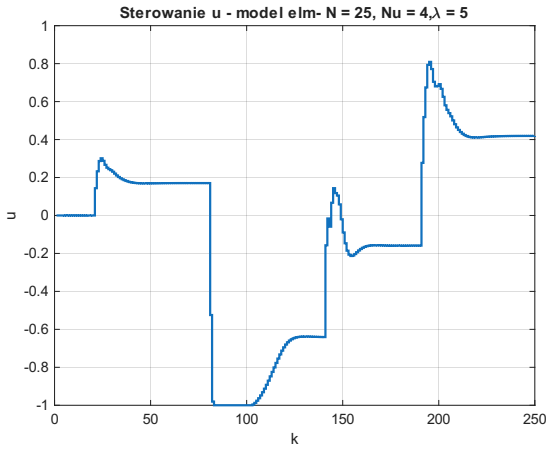
Rys. 6.13. Wyjście dla małej wartości kary $\lambda = 1$



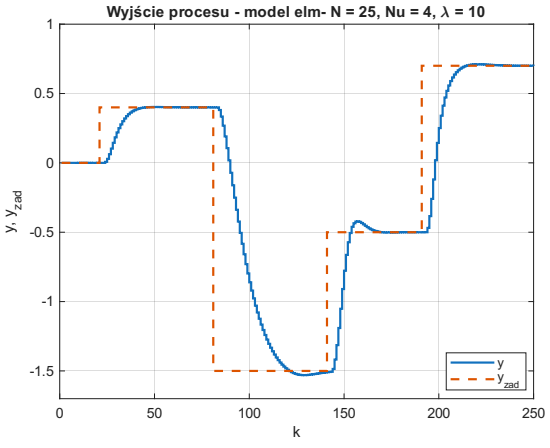
Rys. 6.14. Sygnał sterujący dla $\lambda = 1$



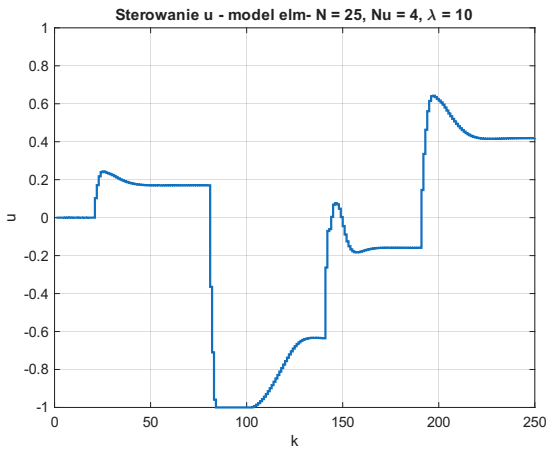
Rys. 6.15. Wyjście dla wybranej wartości $\lambda = 5$



Rys. 6.16. Sygnał sterujący dla $\lambda = 5$



Rys. 6.17. Wyjście dla dużej wartości kary $\lambda = 10$



Rys. 6.18. Sygnał sterujący dla $\lambda = 10$

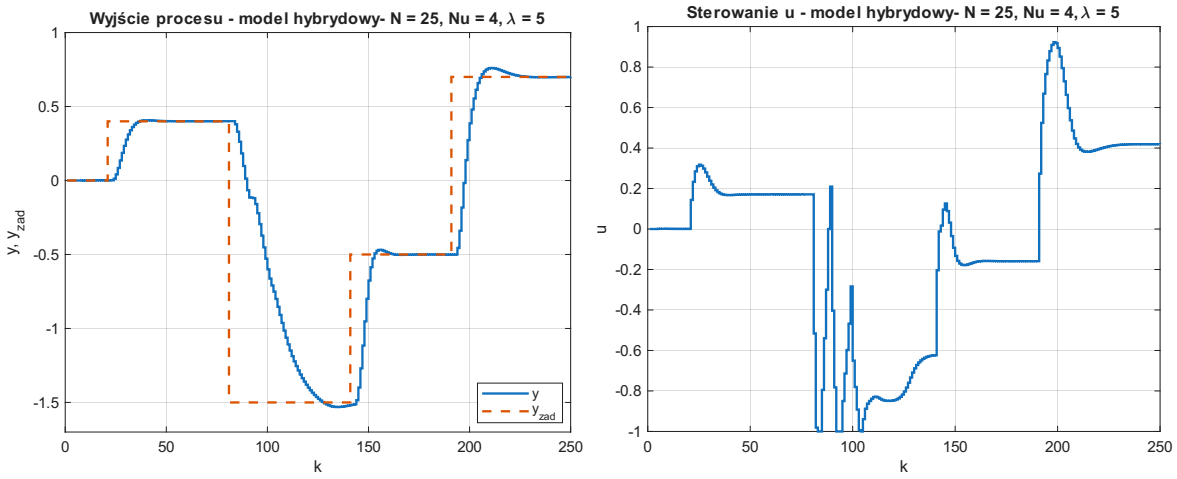
Wartość $\lambda = 5$ została dobrana jako kompromis pomiędzy szybkością reakcji a dynamiką zmian sterowania. Końcowe parametry, dobrane podczas procesu strojenia, zestawiono w tab. 6.1.

Tab. 6.1. Dobrane parametry regulatora NPL

Parametr	Wartość
N	25
N_u	4
λ	5

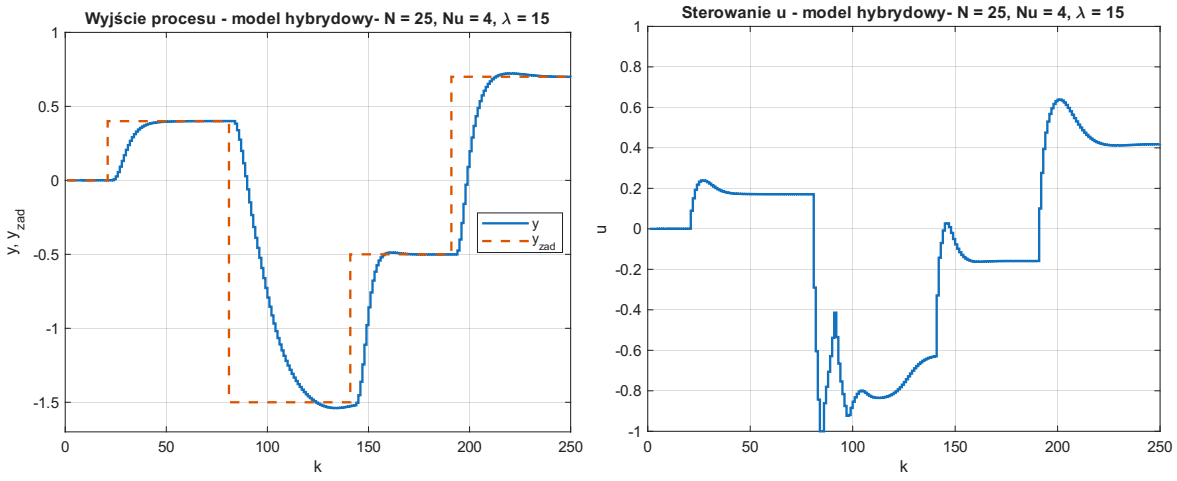
6.3. Regulacja z wykorzystaniem modelu hybrydowego

Na rys. 6.19 przedstawiono przebiegi sygnału wyjściowego $y(k)$, wartości zadanej oraz przebieg sygnału sterującego $u(k)$ dla regulatora opartego o model hybrydowy.



Rys. 6.19. Przebieg wyjścia oraz sterowania (Model hybrydowy)

Ze względu na niższą dokładność modelu hybrydowego, parametry dobrane pierwotnie dla modelu ELM okazały się nieoptymalne. W konsekwencji zdecydowano o zwiększeniu wartości współczynnika kary λ do 15. Wyniki eksperymentu przedstawiono na rys. 6.20.



Rys. 6.20. Przebieg wyjścia oraz sterowania dla $\lambda = 15$ (Model hybrydowy)

Zwiększenie parametru λ pozwoliło na złagodzenie sygnału sterującego o bardzo dynamicznych zmianach do dużo bardziej łagodnego, co pozwoliło zamaskować niedokładności wynikające z modelowania oraz osiągnąć satysfakcjonującą jakość regulacji.

7. Podsumowanie

Celem zrealizowanego projektu była identyfikacja nieliniowego procesu dynamicznego z wykorzystaniem metod sztucznej inteligencji oraz zaprojektowanie algorytmu regulacji predykcyjnej z Nieliniową Predykcją i Linearyzacją (NPL). Przeprowadzone badania symulacyjne pozwoliły na sformułowanie następujących wniosków dotyczących modelowania oraz sterowania procesem.

7.1. Wnioski z identyfikacji procesu

Analiza porównawcza modeli liniowych i nieliniowych wykazała fundamentalne różnice w ich zdolności do odwzorowania dynamiki obiektu:

— **Model liniowy (ARX):** Chociaż w predykcji jednokrokowej osiągnął niskie błędy ($MSE \approx 10^{-4}$), to w trybie rekurencyjnym jego jakość drastycznie spadła ($MSE \approx 10^{-2}$). Potwierdziło to silnie nieliniowy charakter procesu, którego nie da się poprawnie zamodelować modelem liniowym.
— **Sieci ELM:** Zastosowanie sieci typu Extreme Learning Machine pozwoliło na skuteczne odwzorowanie nieliniowości. Badania struktury wykazały, że optymalną liczbą neuronów ukrytych jest $K = 70$. Model ten zapewnił najlepszy kompromis między dokładnością na zbiorze uczącym a zdolnością generalizacji (brak przewymiarowania) w trybie rekurencyjnym.
— **Model hybrydowy:** Zastosowanie architektury sieci neuronowej, wzbogaconej o dodatkowe składniki nieliniowe (funkcje sinus i cosinus), nie przełożyło się na uzyskanie lepszych rezultatów. Mimo optymalizacji struktury (wybór $K = 4$), model ten osiągnął gorsze wyniki końcowe, generując znacznie większe błędy modelowania niż w przypadku sieci ELM. W trybie rekurencyjnym najżniższy błąd średniokwadratowy modelu hybrydowego wynosił około 10^{-4} , podczas gdy dla ELM był on o dwa rzędy wielkości mniejszy 10^{-6}.

7.2. Wnioski z regulacji predykcyjnej (NPL)

Implementacja algorytmu NPL potwierdziła skuteczność sterowania predykcyjnego dla badanej klasy obiektów nieliniowych. Kluczowe obserwacje z etapu strojenia regulatora to:

— **Dobór horyzontów:** Zredukowanie horyzontu predykcji do $N = 25$ oraz horyzontu sterowania do $N_u = 4$ pozwoliło na znaczne zmniejszenie złożoności obliczeniowej oraz poprawy jakośći regulacji. Dalsze zmniejszanie tych parametrów prowadziło do pogorszenia działania regulatora.
— **Współczynnik kary λ:** Dla regulatora opartego na dokładniejszym modelu ELM przyjęto wartość $\lambda = 5$, co zapewniło szybką reakcję układu przy akceptowalnej dynamice sygnału sterującego.
— **Model hybrydowy w pętli regulacji:** Regulator wykorzystujący model z Deep Learning Toolbox wykazywał gorszą jakość predykcji w porównaniu do ELM. Aby uzyskać stabilną pracę układu i zamaskować niedokładności modelu, konieczne było zwiększenie współczynnika kary do wartości $\lambda = 15$, co skutkowało bardziej zachowawczym (wygładzonym) przebiegiem sterowania.

Podsumowując, zastosowanie nieliniowych modeli neuronowych (szczególnie ELM) umożliwiło precyzyjną identyfikację procesu, co z kolei przełożyło się na skuteczne działanie algorytmu regulacji predykcyjnej NPL, zapewniając poprawne śledzenie zadanej trajektorii wyjścia.