UNIVERSITAS INDONESIA

IDENTIFIKASI SISTEM MENGGUNAKAN NEURAL NETWORK

TUGAS MAKALAH UTS

SISTEM BERBASIS PENGETAHUAN

DEVIN EZEKIEL PURBA

2106701583

FAKULTAS TEKNIK

PROGRAM STUDI TEKNIK ELEKTRO

DEPOK

OKTOBER 2024

BAB 1
PENDAHULUAN

1.1. Latar Belakang

Sistem dinamik memiliki peran yang sangat penting dalam berbagai bidang, mulai dari kontrol industri, otomasi, hingga pemodelan fenomena alam yang kompleks. Sistem-sistem ini, yang dapat berupa mesin-mesin industri, kendaraan otomatis, maupun perangkat elektronik lainnya, beroperasi secara berkesinambungan dan membutuhkan kontrol yang tepat agar tetap dapat bekerja secara optimal. Oleh karena itu, untuk dapat mengendalikan sistem-sistem tersebut dengan baik, diperlukan pemahaman yang mendalam mengenai bagaimana sistem tersebut bekerja. Dua langkah yang tidak dapat dilepaskan dalam proses ini adalah pemodelan dan identifikasi sistem dinamik.

Pemodelan sistem dinamik melibatkan pembuatan model matematis yang merepresentasikan perilaku dan respons sistem terhadap berbagai masukan. Sedangkan identifikasi sistem adalah proses menentukan parameter-parameter dalam model tersebut berdasarkan data yang diperoleh dari pengamatan atau pengujian sistem nyata. Identifikasi ini sangat penting karena memungkinkan kita untuk memahami bagaimana sistem akan bereaksi terhadap kondisi tertentu dan bagaimana mengontrolnya agar dapat mencapai tujuan yang diinginkan.

Salah satu metode yang semakin populer dalam melakukan pemodelan dan identifikasi sistem adalah penggunaan jaringan saraf tiruan atau neural network. Neural network memiliki kemampuan luar biasa dalam mempelajari dan mengenali pola-pola yang ada dalam data, termasuk pola hubungan non-linear antara variabel input dan output. Hal ini memungkinkan neural network untuk memodelkan perilaku sistem yang kompleks tanpa memerlukan persamaan matematis yang eksplisit dan sulit. Dengan neural network, model sistem dapat dibangun berdasarkan data, di mana hubungan antara input dan output dipelajari melalui proses pelatihan.

Penggunaan neural network menjadi sangat relevan ketika berhadapan dengan sistem dinamik yang memiliki sifat non-linear dan kompleks, yang sulit atau bahkan tidak mungkin dimodelkan dengan metode konvensional. Kemampuan neural network untuk mengatasi masalah non-linearitas ini menjadikannya alat yang sangat efektif

dalam proses identifikasi sistem. Neural network dapat mempelajari pola dan hubungan antara masukan dan keluaran yang tidak selalu terlihat secara jelas oleh metode tradisional.

Dalam makalah ini, akan dibahas bagaimana neural network dapat digunakan untuk melakukan identifikasi sistem dinamik berdasarkan persamaan sistem diskrit. Persamaan sistem diskrit yang digunakan memiliki karakteristik umpan balik (feedback), di mana keluaran sistem saat ini (y[k]) tidak hanya dipengaruhi oleh masukan saat ini (u[k]), tetapi juga oleh keluaran sebelumnya (y[k-1]) dan masukan pada waktu sebelumnya (u[k-1]). Dengan adanya komponen umpan balik ini, sistem menjadi lebih kompleks dan memerlukan pendekatan yang lebih canggih untuk dapat dipahami dan dimodelkan dengan tepat. Neural network akan dilatih untuk mempelajari hubungan ini, sehingga dapat digunakan untuk memprediksi perilaku sistem berdasarkan input yang diberikan.

Tujuan dari proses identifikasi ini adalah untuk membangun model yang dapat meniru perilaku sistem asli dengan akurat, sehingga dapat dipergunakan untuk berbagai keperluan, seperti kontrol otomatis, simulasi, dan pengembangan algoritma prediksi. Dengan memahami bagaimana neural network dapat diaplikasikan untuk mengidentifikasi sistem-sistem dengan sifat umpan balik, diharapkan hasil dari penelitian ini dapat memberikan wawasan yang lebih luas tentang penerapan teknik pembelajaran mesin dalam bidang kontrol dan otomasi.

1.2. Rumusan Masalah

1.2.1. Bagaimana cara mengidentifikasi sistem dinamik diskrit menggunakan neural network?

1.2.2. Bagaimana neural network mempelajari hubungan input dan output pada sistem dengan umpan balik?

1.2.3. Seberapa akurat neural network dalam memprediksi keluaran sistem berdasarkan input?

1.2.4. Apakah neural network efektif untuk memodelkan sistem non-linear dan kompleks?

1.3. Tujuan

1.3.1. Mengidentifikasi sistem dinamik diskrit menggunakan jaringan saraf tiruan (neural network) berdasarkan persamaan yang diberikan.

1.3.2. Memahami bagaimana neural network dapat digunakan untuk mempelajari hubungan antara input dan output pada sistem yang memiliki sifat umpan balik.

1.3.3. Mengevaluasi kinerja neural network dalam melakukan prediksi keluaran sistem berdasarkan input yang diberikan.

1.3.4. Memberikan solusi pemodelan yang efektif untuk sistem-sistem serupa yang memiliki karakteristik non-linear dan kompleks.

1.4. Pembatasan Masalah

1.4.1. Identifikasi sistem yang dilakukan hanya terbatas pada sistem dinamik diskrit dengan persamaan yang memiliki sifat umpan balik, di mana keluaran saat ini bergantung pada keluaran dan masukan sebelumnya.

1.4.2. Neural network yang digunakan dalam penelitian ini terbatas pada model jaringan saraf tiruan feedforward dengan metode pembelajaran supervised learning.

1.4.3. Data yang digunakan untuk pelatihan dan pengujian neural network diambil dari simulasi sistem yang telah ditentukan, tanpa melibatkan data nyata dari sistem fisik.

1.4.4. Evaluasi kinerja neural network akan difokuskan pada tingkat akurasi prediksi dan kemampuan model dalam mempelajari pola hubungan input-output saja

1.4.5. Dalam makalah ini, hanya dilakukan identifikasi sistem menggunakan jaringan saraf tiruan saja, belum sampai pada desain pengendali untuk memperbaiki respon sistem non-linear

BAB 2
STATE OF THE ART

2.1. Artificial Neural Network (ANN)

Artificial Neural Network (ANN) adalah model komputasi yang terinspirasi oleh cara kerja otak manusia. ANN terdiri dari neuron-neuron buatan yang dihubungkan oleh bobot (weights) yang dapat diatur. ANN dapat belajar dari data yang diberikan melalui proses pelatihan (training), dan berfungsi untuk mengenali pola atau memetakan input ke output tertentu.

Secara umum, struktur ANN dapat dibagi menjadi tiga bagian utama:

- **Lapisan Input (Input Layer):** Lapisan ini menerima data mentah dari luar jaringan dan meneruskannya ke lapisan berikutnya.
- **Lapisan Tersembunyi (Hidden Layers):** Lapisan ini melakukan pemrosesan terhadap data dan mengekstrak fitur-fitur yang relevan. Lapisan ini dapat terdiri dari satu atau lebih lapisan.
- **Lapisan Output (Output Layer):** Lapisan ini menghasilkan prediksi akhir berdasarkan pemrosesan yang dilakukan oleh lapisan-lapisan tersembunyi.

Pada setiap neuron, output dihasilkan melalui persamaan:

$$a_j = f\left(\sum_{i=1}^{n} w_{ij} \cdot x_i + b_j\right)$$

di mana:

- w_{ij} adalah bobot antara neuron ke-iii di lapisan sebelumnya dengan neuron ke-jjj di lapisan saat ini
- x_i adalah input
- b_j adalah bias
- f adalah fungsi aktivasi

2.2. Supervised Learning

Supervised learning adalah metode pembelajaran mesin di mana model dilatih menggunakan data input-output yang sudah diberi label. Pada ANN, data input dipasangkan dengan output yang diharapkan. Selama proses pelatihan, jaringan belajar untuk memetakan input ke output yang benar dengan meminimalkan error (loss) antara output jaringan dan output yang diharapkan.

Contoh loss function yang sering digunakan adalah Mean Squared Error (MSE):

$$J = \frac{1}{N}\sum_{k=1}^{N}(y_k - \widehat{y_k})^2$$

di mana:

- J adalah nilai Loss atau error
- y_k adalah nilai output yang diharapkan
- $\widehat{y_k}$ adalah nilai output yang diprediksi oleh jaringan

2.3. Forward Propagation

Forward propagation adalah proses di mana data input diteruskan melalui jaringan, dari lapisan input ke lapisan output, untuk menghasilkan prediksi akhir. Pada setiap lapisan, hasil kalkulasi berupa dot product antara input dan bobot, ditambahkan dengan bias, kemudian diproses melalui fungsi aktivasi.

Persamaan forward propagation:

$$z_j = \sum_{i=1}^{n} w_{ij} \cdot x_i + b_j$$

$$a_j = f(z_j)$$

di mana:

- z_j adalah input bersih yang masuk ke neuron
- a_j adalah output neuron setelah fungsi aktivasi
- f adalah fungsi aktivasi seperti sigmoid atau tanh

Untuk aplikasi sistem kendali, fungsi aktivasi yang biasanya digunakan adalah **fungsi tanh (hyperbolic tangent).** Fungsi tanh memiliki range output antara -1 dan 1, yang cocok untuk sistem kendali yang umumnya bekerja pada rentang nilai negatif dan positif. Persamaan fungsi tanh adalah:

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

2.4. Backward Propagation

Backward propagation adalah langkah penting dalam pelatihan *neural network* yang digunakan untuk memperbarui parameter-parameter jaringan, yaitu *weights* dan *biases*, sehingga jaringan dapat belajar dari kesalahan yang terjadi selama proses pelatihan. Proses ini memungkinkan jaringan saraf tiruan untuk meminimalkan fungsi kerugian (loss function) dengan cara menghitung gradien dan memperbarui parameter menggunakan algoritma optimisasi, seperti *Gradient Descent*.

Pada dasarnya, *backward propagation* bekerja dengan cara sebagai berikut:

1. Mencari *error* atau kesalahan dari prediksi jaringan dibandingkan dengan nilai sebenarnya.
2. Menghitung gradien (turunan) dari fungsi kerugian terhadap setiap *weight* dan *bias* dengan menggunakan *Chain Rule* dari kalkulus.
3. Memperbarui *weights* dan *biases* berdasarkan gradien tersebut untuk meminimalkan kesalahan pada iterasi berikutnya.

Persamaan pembaruan bobot yang menggunakan algoritma **Gradient Descent Optimizer**:

$$w_{ij} = w_{ij} - \alpha \frac{\partial J}{\partial w_{ij}}$$

$$b_j = b_j - \alpha \frac{\partial J}{\partial b_j}$$

Di mana:

- w_{ij} adalah *weight* yang menghubungkan neuron ke-i pada lapisan sebelumnya dengan neuron ke-j pada lapisan saat ini

- b_j adalah *bias* neuron ke-j
- α adalah *learning rate*, yang menentukan seberapa besar pembaruan yang dilakukan
- $\frac{\partial J}{\partial w_{ij}}$ adalah gradien dari fungsi kerugian J terhadap *weight* w_{ij}.
- $\frac{\partial J}{\partial b_j}$ adalah gradien dari fungsi kerugian J terhadap *bias* b_j

Gradien tersebut diperoleh dengan menggunakan turunan parsial dari fungsi kerugian dan menerapkan *Chain Rule*. Perhitungan gradien ini memungkinkan jaringan untuk mengetahui arah di mana kesalahan dapat diminimalkan dan oleh karena itu, dapat memperbarui parameter-parameter jaringan untuk menghasilkan prediksi yang lebih akurat. Selama proses *backward propagation*, turunan dari *activation function* memainkan peran penting. Turunan ini diperlukan untuk menghitung gradien dari fungsi kerugian terhadap *weights* dan *biases*. Misalnya, ketika menggunakan fungsi aktivasi *tanh*, turunan dari fungsi *tanh* digunakan untuk mengukur sensitivitas kesalahan terhadap perubahan pada neuron. Turunan dari fungsi *tanh* adalah:

$$f'(x) = 1 - \tanh^2(x)$$

Turunan ini digunakan dalam proses *backward propagation* untuk menghitung perubahan yang perlu diterapkan pada setiap parameter jaringan. Jika keluaran dari neuron adalah A dan fungsi aktivasi adalah f, maka turunan dari keluaran neuron tersebut adalah $f'(Z)$, di mana Z adalah nilai sebelum diterapkan fungsi aktivasi:

$$dZ = dA \cdot f'(Z)$$

di sini:

- dZ adalah gradien dari fungsi kerugian terhadap Z,
- dA adalah gradien dari fungsi kerugian terhadap keluaran neuron, dan
- $f'(Z)$ adalah turunan dari fungsi aktivasi (misalnya *tanh*).

2.5. Normalisasi Min-Max Scaler

Normalisasi adalah teknik untuk menskalakan data sehingga berada dalam rentang tertentu, misalnya [0, 1] atau [-1, 1]. Hal ini membantu dalam mempercepat proses pelatihan karena memastikan bahwa nilai input berada dalam rentang yang konsisten. Persamaan normalisasi:

$$x_{norm} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

Denormalisasi adalah proses mengembalikan data yang telah dinormalisasi ke skala aslinya:

$$x = x_{norm} \cdot (\max(x) - \min(x)) + \min(x)$$

2.6. Inisialisasi Bobot Nguyen-Widrow

Metode Nguyen-Widrow adalah teknik inisialisasi bobot untuk mempercepat proses pelatihan jaringan. Inisialisasi yang baik dapat membantu jaringan untuk konvergen lebih cepat. Persamaan Nguyen-Widrow:

$$W = \beta \frac{W_{random}}{||W\_random||}$$

dengan $\beta = 0.7 \cdot (output\ size)^{\frac{1}{input\ size}}$

2.7. L2 Regularization

Overfitting terjadi ketika model belajar terlalu baik terhadap data pelatihan sehingga tidak mampu menangani data baru dengan baik. Underfitting terjadi saat model gagal belajar pola-pola penting dari data pelatihan.

L2 Regularization adalah teknik untuk mencegah overfitting dengan menambahkan penalti terhadap bobot besar. Fungsi loss dengan L2 Regularization adalah:

$$J_{total} = J_{original} + \lambda \sum_j w_j^2$$

Dimana λ adalah parameter regularisasi.

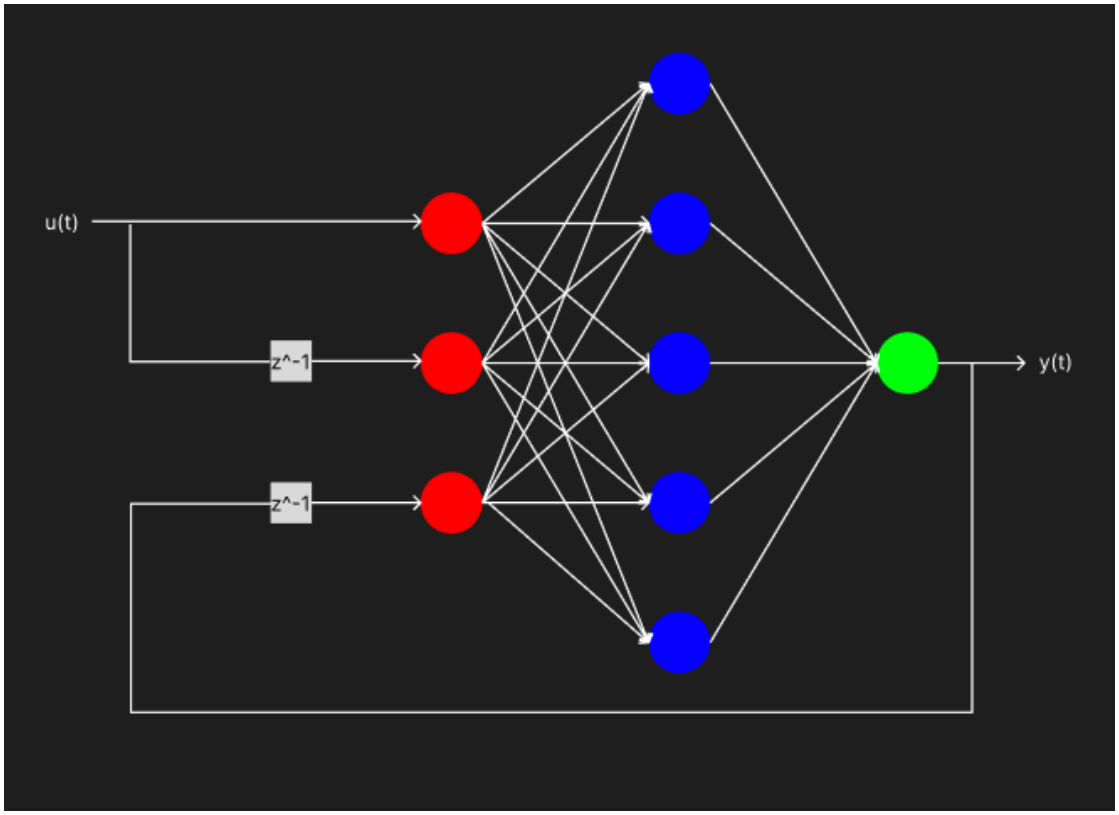
2.8. Identifikasi Sistem dengan Metode Output Feedback Secara Vektoral pada ANN

Metode ini menggunakan feedback dari output sistem sebagai input tambahan ke jaringan. ANN menerima input dari beberapa langkah waktu sebelumnya, memungkinkan untuk memodelkan sistem dinamis:

$$y[k] = f(X[k])$$

$$X[k] = (y[k-1], y[k-2], u[k], u[k-1], u[k-2])$$

Metode ini membuat jaringan dapat belajar pola feedback dan memprediksi output berdasarkan input masa lalu yang umumnya digunakan dalam sistem kendali. Hal ini membuat pelatihan dalam matriks tidak memungkinkan karena data input yang dilakukan umpan balik akan berubah terhadap waktu dan terhadap sistemnya sehingga input data pada sistem harus berupa vektoral.



BAB 3
METODOLOGI

3.1. Data Collection

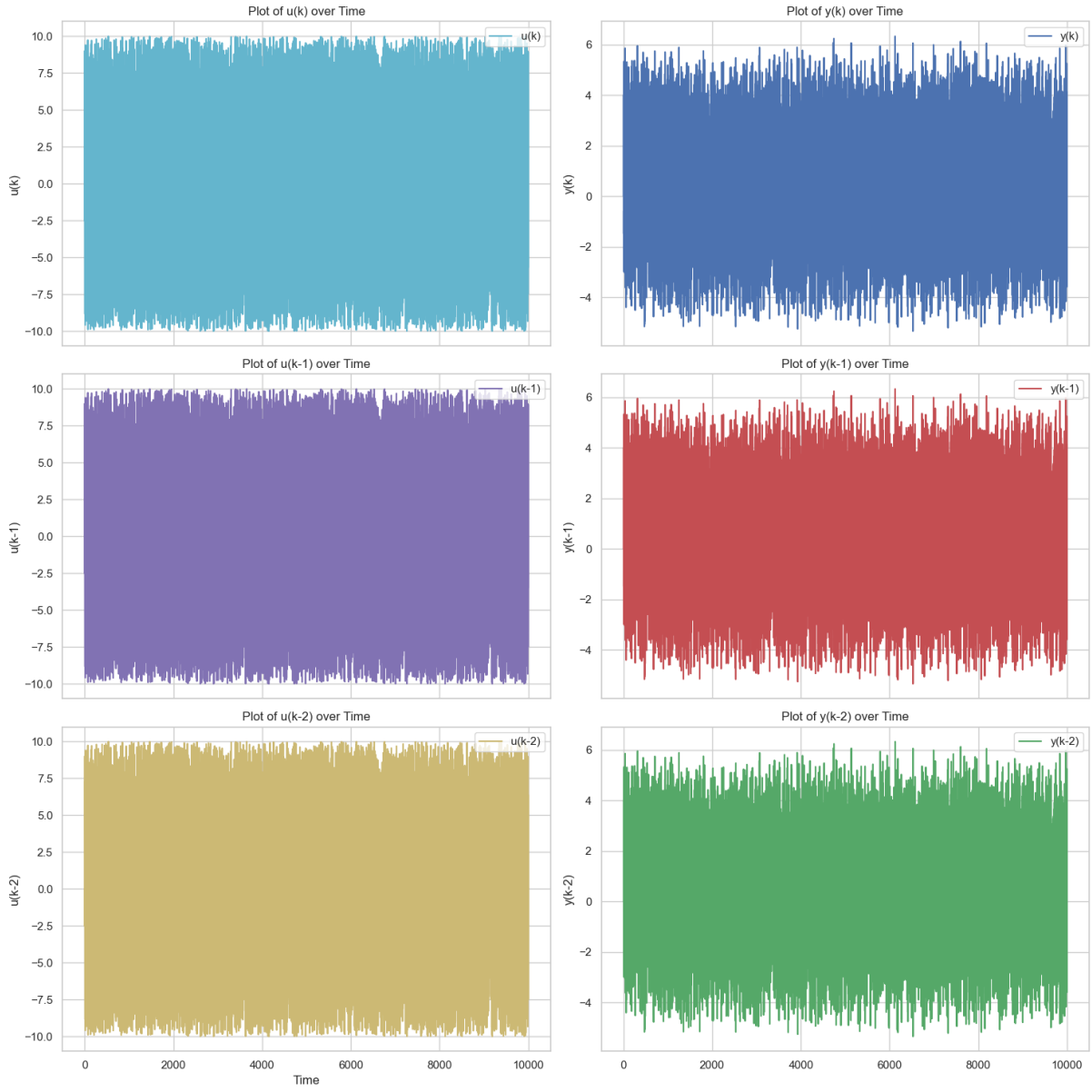
Pada tahap Pengumpulan Data dalam program ini, proses dimulai dengan mengimpor beberapa pustaka penting seperti numpy untuk operasi numerik, matplotlib.pyplot untuk visualisasi grafik, pandas untuk manipulasi data (meskipun tidak digunakan secara eksplisit pada bagian ini), seaborn untuk memperindah tampilan grafik, serta sklearn.preprocessing dan scipy.signal untuk tugas penskalaan dan pemrosesan sinyal. Langkah pertama adalah menetapkan random seed menggunakan np.random.seed(42) untuk memastikan hasil yang konsisten setiap kali program dijalankan. Selanjutnya, program mendefinisikan variabel waktu dengan k_max = 10000 yang menunjukkan jumlah total sampel data. Array t dibuat menggunakan np.linspace(0, k_max, k_max), yang menghasilkan nilai yang terdistribusi secara linear dari 0 hingga k_max. Variabel t ini berfungsi sebagai sumbu waktu untuk semua grafik. Program kemudian menghasilkan sinyal input acak u(k) menggunakan u = 2 * np.random.uniform(-5, 5, k_max). Sinyal ini memiliki rentang nilai antara -10 hingga 10, yang mencerminkan berbagai kondisi yang bisa memengaruhi sistem, memberikan variasi data yang kaya. Setelah itu, array y diinisialisasi dengan y = np.zeros(k_max), yang berfungsi untuk menyimpan nilai keluaran sistem pada setiap indeks waktu k. Pada tahap ini, array y diisi dengan nilai nol sebagai langkah awal. Penghitungan keluaran sistem y(k) dilakukan melalui sebuah loop, di mana nilai y(k) ditentukan berdasarkan nilai y(k-1) (keluaran sebelumnya) serta input saat ini u(k) dan input sebelumnya u(k-1). Persamaan yang digunakan adalah:

$$y[k] = \frac{1}{(1 + (y[k-1])^2)} + 0.25u[k] - 0.3u[k-1]$$

Persamaan ini menggambarkan sistem dinamik non-linear di mana keluaran saat ini y(k) dipengaruhi oleh keluaran dan input dari langkah-langkah sebelumnya. Untuk melengkapi pengumpulan data, program membuat beberapa array yang berisi shifted values atau nilai geser, yaitu y(k-1), y(k-2), u(k-1), dan u(k-2). Array ini dibuat dengan tujuan untuk menyimpan keluaran dan input dari satu hingga dua langkah sebelumnya, yang penting untuk analisis data deret waktu. Proses ini dilakukan dengan menggeser data dalam array:

- y_k_1[1:] = y[:-1] menyimpan nilai y(k-1)
- y_k_2[2:] = y[:-2] menyimpan nilai y(k-2)
- Array u_k_1 dan u_k_2 dibuat dengan metode serupa untuk input u(k-1) dan u(k-2).

Sebagai langkah akhir dari pengumpulan data, program membuat visualisasi menggunakan grid 3x2 untuk menunjukkan hubungan input-output. Pada baris pertama, program menampilkan grafik u(k) dan y(k) untuk memperlihatkan sinyal input dan output saat ini. Baris kedua menampilkan u(k-1) dan y(k-1) untuk menunjukkan bagaimana sinyal input dan output di masa lalu berperilaku, sementara baris ketiga menggambarkan u(k-2) dan y(k-2) yang menunjukkan keadaan input dan output pada dua langkah sebelumnya. Setiap grafik dilengkapi dengan judul, label sumbu, dan legenda untuk memperjelas interpretasi data. Terakhir, plt.tight_layout() digunakan untuk memastikan tata letak grafik tidak saling tumpang tindih, dan plt.show() menampilkan seluruh grafik tersebut. Tahap pengumpulan data ini bertujuan untuk menghasilkan data sintetik yang mencerminkan perilaku sistem dinamik. Dengan menciptakan sinyal input acak dan menghitung keluaran berdasarkan model non-linear, serta membuat visualisasi yang informatif, program ini menyediakan data dasar yang dapat digunakan untuk analisis lebih lanjut atau pemodelan.



3.2. Data Preprocessing

Pada tahap Pra-pemrosesan Data dalam program ini, proses dimulai dengan mengatur data mentah yang telah dikumpulkan menjadi bentuk yang siap untuk digunakan dalam pelatihan model. Langkah pertama adalah menggabungkan beberapa variabel menjadi satu matriks fitur (X). Matriks X ini terdiri dari data yang telah dihasilkan sebelumnya, yaitu y(k-1), y(k-2), u(k), u(k-1), dan u(k-2), yang disusun menggunakan np.array([y_k_1, y_k_2, u, u_k_1, u_k_2]).T. Variabel target y juga diubah menjadi bentuk kolom dengan y.reshape(-1, 1), sementara sinyal input u disusun dengan cara yang sama.

Selanjutnya, proses normalisasi dilakukan untuk memastikan bahwa semua fitur berada dalam rentang nilai yang sama, yang sangat penting untuk mempercepat

konvergensi selama pelatihan model dan meningkatkan kinerja. Program menggunakan MinMaxScaler dari pustaka sklearn.preprocessing untuk merubah nilai fitur ke dalam rentang antara -1 dan 1. Langkah ini bertujuan untuk menghindari dominasi fitur dengan skala besar dan memastikan bahwa semua fitur memiliki dampak yang seimbang pada model. Normalisasi dilakukan pada variabel input u, u(k-1), dan u(k-2), serta pada output y, y(k-1), dan y(k-2). Setelah fitting menggunakan data input asli, program melakukan transformasi untuk mengubah data ke dalam skala yang dinormalisasi. Matriks fitur yang telah dinormalisasi, X_norm, kemudian disusun menggunakan array np.array([y_k_1_norm.flatten(), y_k_2_norm.flatten(), u_norm.flatten(), u_k_1_norm.flatten(), u_k_2_norm.flatten()]).T. Data ini kini siap untuk digunakan dalam pelatihan dan validasi model.

Sebagai langkah berikutnya, program melakukan pembagian data menjadi dua bagian, yaitu data pelatihan dan data validasi. Sebanyak 80% dari total data digunakan sebagai data pelatihan (X_train dan y_train), sedangkan 20% sisanya digunakan untuk validasi (X_val dan y_val). Pembagian ini juga diterapkan pada data yang telah dinormalisasi (X_train_norm, y_train_norm, X_val_norm, dan y_val_norm). Dengan demikian, program memastikan bahwa model dapat dilatih dengan baik menggunakan data yang cukup banyak, sementara validasi dengan data yang terpisah akan membantu menilai kemampuan generalisasi model pada data baru yang tidak dilihat selama proses pelatihan. Tahap pra-pemrosesan data ini memastikan bahwa data input dan output berada dalam format yang sesuai, dinormalisasi, dan tersegmentasi dengan baik untuk melatih dan memvalidasi model secara efektif, sehingga meningkatkan kualitas dan keakuratan proses pembelajaran mesin.

3.3. Training dan Validasi Tahap 1

Pada tahap Training dan Validasi Tahap 1, program menggunakan kelas MLP (Multi-Layer Perceptron) untuk membangun, melatih, dan memvalidasi model jaringan saraf tiruan yang dirancang untuk memprediksi perilaku sistem berdasarkan input-output yang telah dinormalisasi. Kelas MLP diinisialisasi dengan beberapa parameter, seperti jumlah lapisan (layers), fungsi aktivasi (activations), jenis optimizer (optimizer), laju pembelajaran (learning_rate), dan parameter regulasi L2 (l2_lambda). Pada awalnya, bobot diinisialisasi menggunakan metode Nguyen-Widrow yang membantu mempercepat konvergensi dengan mengatur bobot awal secara efisien. Selain itu, bias diinisialisasi dengan nilai nol.

Selama proses forward pass, input diteruskan melalui setiap lapisan jaringan untuk menghasilkan output prediksi, di mana setiap input dikalikan dengan bobot, ditambahkan bias, dan diproses menggunakan fungsi aktivasi seperti tanh atau linear. Kemudian, pada backward pass, program menghitung gradien kerugian menggunakan metode backpropagation untuk menentukan perubahan yang diperlukan pada bobot dan bias, sambil menerapkan regulasi L2 untuk mengurangi risiko overfitting. Pembaruan bobot dilakukan menggunakan algoritma optimasi yang dipilih seperti adam, rmsprop, atau SGD, yang mempertimbangkan parameter seperti momentum dan rata-rata kuadrat tertimbang.

Selama proses pelatihan, model dilatih melalui beberapa epoch. Pada setiap epoch, prediksi dihasilkan dari data pelatihan menggunakan forward pass, kemudian gradien dihitung melalui backward pass, dan bobot diperbarui. Jika data validasi tersedia, model juga dievaluasi pada data tersebut, dan nilai kerugian serta akurasi disimpan untuk memantau kinerja selama pelatihan. Program juga menyediakan visualisasi berupa grafik yang menampilkan perbandingan kerugian dan akurasi antara data pelatihan dan validasi, menggunakan skala logaritmik pada sumbu epoch untuk memudahkan pemantauan perubahan tren selama pelatihan.

Setelah pelatihan selesai, model digunakan untuk prediksi pada data baru atau data yang telah dilatih, di mana hasil prediksi dibandingkan dengan nilai aktual untuk melihat seberapa baik model dapat menggeneralisasi pola yang dipelajari. Program juga menyajikan grafik perbandingan antara nilai prediksi dan nilai aktual untuk data pelatihan dan validasi, memberikan gambaran visual tentang akurasi model dalam mereplikasi dinamika sistem yang dipelajari. Secara keseluruhan, proses ini bertujuan untuk menghasilkan model jaringan saraf tiruan yang stabil, akurat, dan mampu menggeneralisasi dengan baik pada data yang belum pernah dilihat sebelumnya.

3.4. Training dan Validasi Tahap 2

Pada tahap Training dan Validasi Tahap 2, program menggunakan kelas MLPVectoral untuk melatih dan memvalidasi model jaringan saraf tiruan yang dirancang dengan mekanisme umpan balik (feedback). Kelas ini mirip dengan MLP pada tahap pertama, tetapi dengan penekanan lebih pada integrasi feedback dari prediksi sebelumnya untuk meningkatkan akurasi prediksi berurutan. Kelas MLPVectoral diinisialisasi dengan parameter seperti jumlah lapisan (layers), fungsi

aktivasi (activations), jenis optimizer (optimizer), laju pembelajaran (learning_rate), dan parameter regulasi L2 (l2_lambda). Kelas ini juga memungkinkan untuk menggunakan bobot dan bias yang telah ditentukan sebelumnya, yang dapat menghemat waktu pelatihan jika model sudah pernah dilatih. Salah satu perbedaan utama dari tahap pertama adalah penggunaan mekanisme umpan balik dalam proses pelatihan. Pada bagian ini, selama proses training, nilai prediksi yang dihasilkan (y_pred) pada setiap langkah digunakan sebagai input untuk langkah berikutnya (y(k-1) dan y(k-2)). Hal ini menciptakan dinamika yang lebih realistis karena model dapat belajar dari pola temporal yang ada dalam data, menjadikannya lebih efektif dalam memodelkan sistem yang membutuhkan prediksi berurutan.

Selama proses forward pass, model menerapkan fungsi aktivasi seperti tanh atau linear pada setiap lapisan. Input diolah dengan bobot dan bias, kemudian diaktifkan untuk menghasilkan output pada lapisan berikutnya. Ketika masuk ke dalam backward pass, gradien kerugian dihitung dan digunakan untuk meng-update bobot dan bias melalui algoritma optimasi seperti adam, rmsprop, atau SGD, serupa dengan metode pada tahap pertama. Namun, perhitungan gradien di sini memperhitungkan pengaruh feedback, yang membantu dalam melatih model untuk menangkap pola sekuensial dalam data. Selama proses pelatihan, model dilatih dalam beberapa epoch, dan untuk setiap epoch, data training diolah dengan feedback yang memperbarui input berdasarkan hasil prediksi sebelumnya. Setelah menyelesaikan satu epoch, model juga mengevaluasi kinerja pada data validasi untuk menghitung metrik seperti kerugian dan akurasi, yang kemudian dicatat untuk pemantauan. Program juga menyediakan visualisasi untuk melihat tren kerugian dan akurasi dari pelatihan dan validasi, yang membantu dalam memahami apakah model mengalami overfitting atau underfitting.

Perbedaan mendasar antara tahap pertama dan tahap kedua adalah bahwa MLP pada tahap pertama melakukan pelatihan tanpa feedback eksplisit, artinya setiap prediksi independen dari prediksi sebelumnya. Sementara itu, MLPVectoral pada tahap kedua memanfaatkan mekanisme umpan balik yang memungkinkan model untuk belajar dari urutan data, meningkatkan kemampuan prediksi dalam situasi di mana terdapat pola temporal. Dengan demikian, tahap kedua lebih cocok untuk masalah yang memerlukan analisis sekuensial atau prediksi yang bergantung pada kondisi sebelumnya, memberikan hasil yang lebih akurat dan konsisten pada data yang memiliki ketergantungan waktu. Training dan validasi tahap kedua dengan

MLPVectoral meningkatkan kemampuan model dalam mengidentifikasi pola sekuensial dengan menggunakan feedback, sehingga lebih unggul dalam menangani data berurutan dan menghasilkan prediksi yang lebih cermat dibandingkan dengan model yang dilatih pada tahap pertama tanpa feedback.

3.5. Testing Hasil Identifikasi Sistem

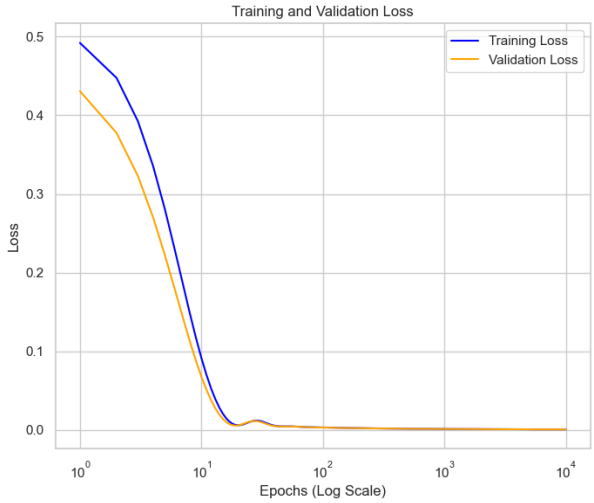
Pada tahap Testing Hasil Identifikasi Sistem, program bertujuan untuk mengevaluasi kemampuan model dalam memprediksi keluaran sistem berdasarkan berbagai sinyal input baru. Tahap ini melibatkan pembuatan sinyal uji, seperti gelombang sinusoidal, kotak, dan segitiga, dengan parameter tertentu (frekuensi dan amplitudo) yang kemudian digunakan sebagai input untuk sistem. Setiap sinyal input diolah untuk menghasilkan keluaran sistem y_test dengan menggunakan persamaan yang sama seperti yang dipakai saat pelatihan. Data y_test dan u_test ini kemudian dinormalisasi menggunakan MinMaxScaler yang telah dilatih dengan data pelatihan, memastikan skala yang konsisten antara data uji dan data pelatihan. Setelah normalisasi, matriks input X_test dibentuk dari nilai-nilai y(k-1), y(k-2), u(k), u(k-1), dan u(k-2), yang kemudian digunakan untuk menghasilkan prediksi keluaran menggunakan model nn (dari Tahap 1) dan nnv (dari Tahap 2).

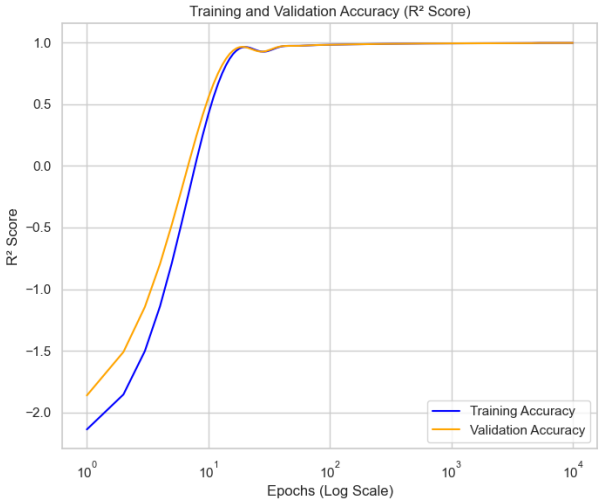
Program membuat prediksi keluaran (y_pred) dan membandingkannya dengan nilai y_test aktual untuk menilai akurasi model. Selain itu, hasil uji divisualisasikan dalam grafik yang menampilkan perbandingan antara sinyal input (gelombang sinusoidal, kotak, atau segitiga) dengan keluaran yang dihasilkan, serta perbandingan antara nilai aktual dan prediksi dari model Tahap 1 dan Tahap 2. Penggunaan tiga jenis sinyal input ini memungkinkan pengujian yang lebih komprehensif terhadap model, memastikan model tidak hanya mengenali satu pola tetapi dapat menggeneralisasi dan mengidentifikasi berbagai pola dinamis. Grafik perbandingan membantu memberikan gambaran visual tentang seberapa akurat model dalam mereplikasi perilaku sistem yang sebenarnya. Secara keseluruhan, tahap ini penting untuk menilai efektivitas model yang dilatih dan memastikan bahwa model mampu memberikan prediksi yang konsisten dan akurat untuk berbagai skenario input.

BAB 4
HASIL DAN ANALISIS

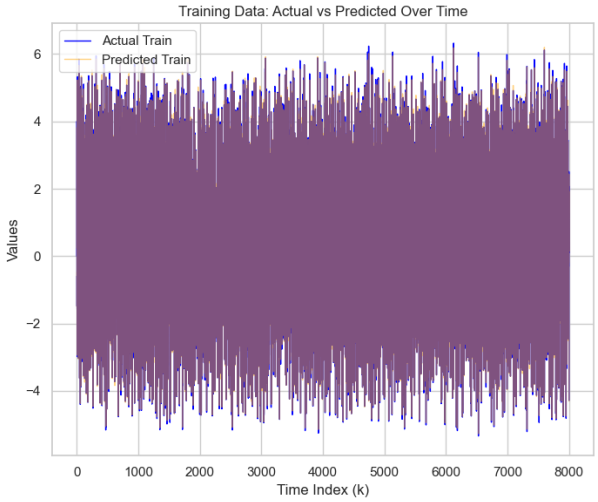
4.1. 1 Hidden Layer dan 7 Neuron

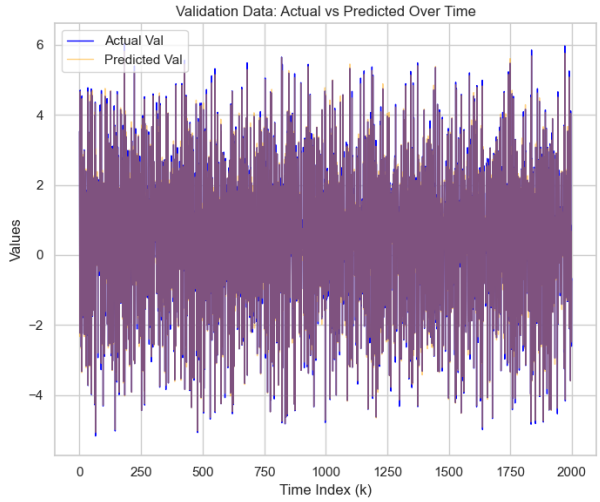
4.1.1. Hasil Tahap 1



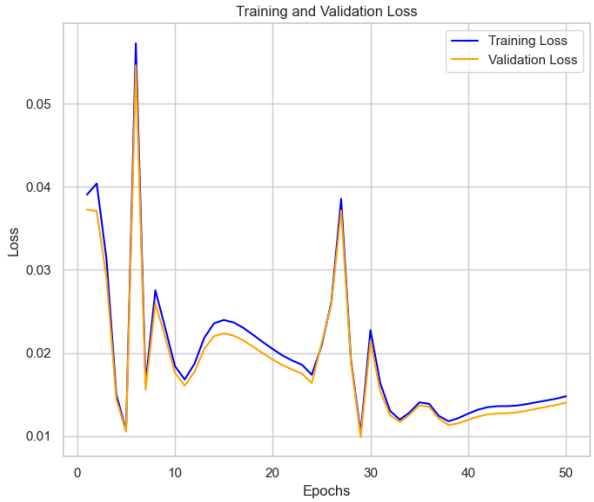


Train Loss: 0.0006, Train Acc: 0.9987, Val Loss: 0.0006, Val Acc: 0.9986

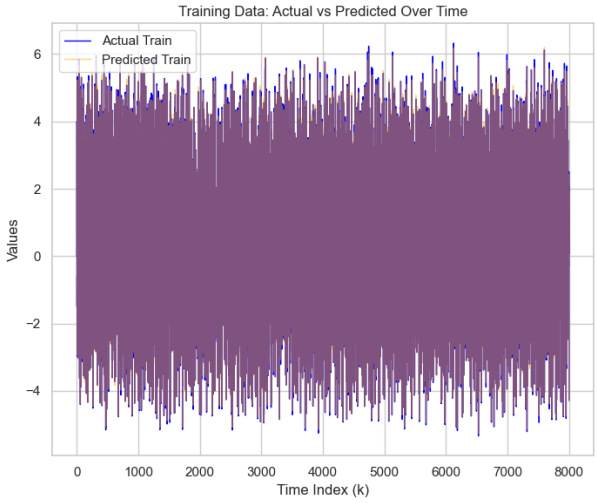


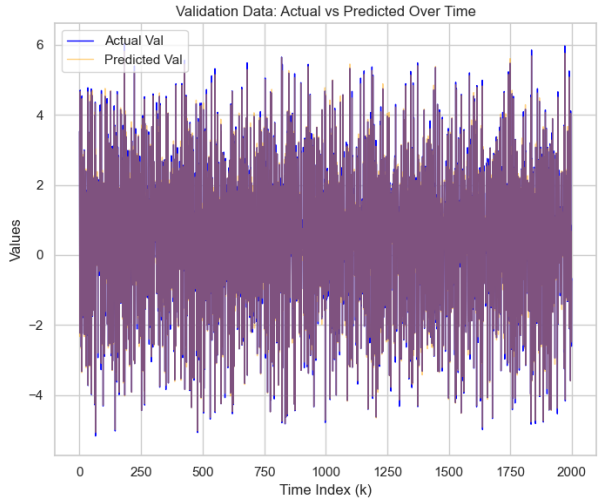


4.1.2. Hasil Tahap 2

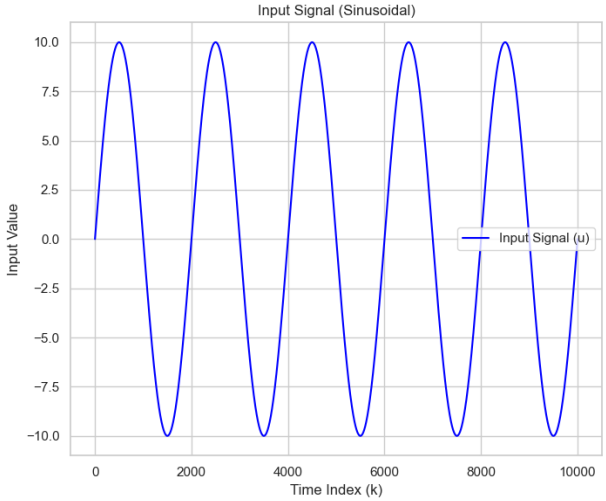


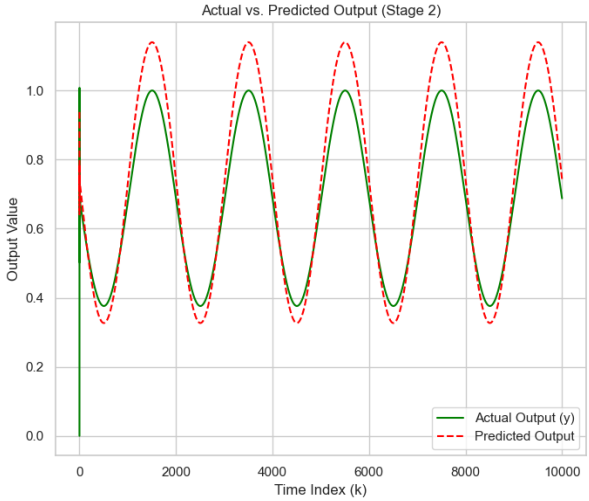
Train Loss: 0.0148, Train Acc: 0.9973, Val Loss: 0.0140, Val Acc: 0.9973

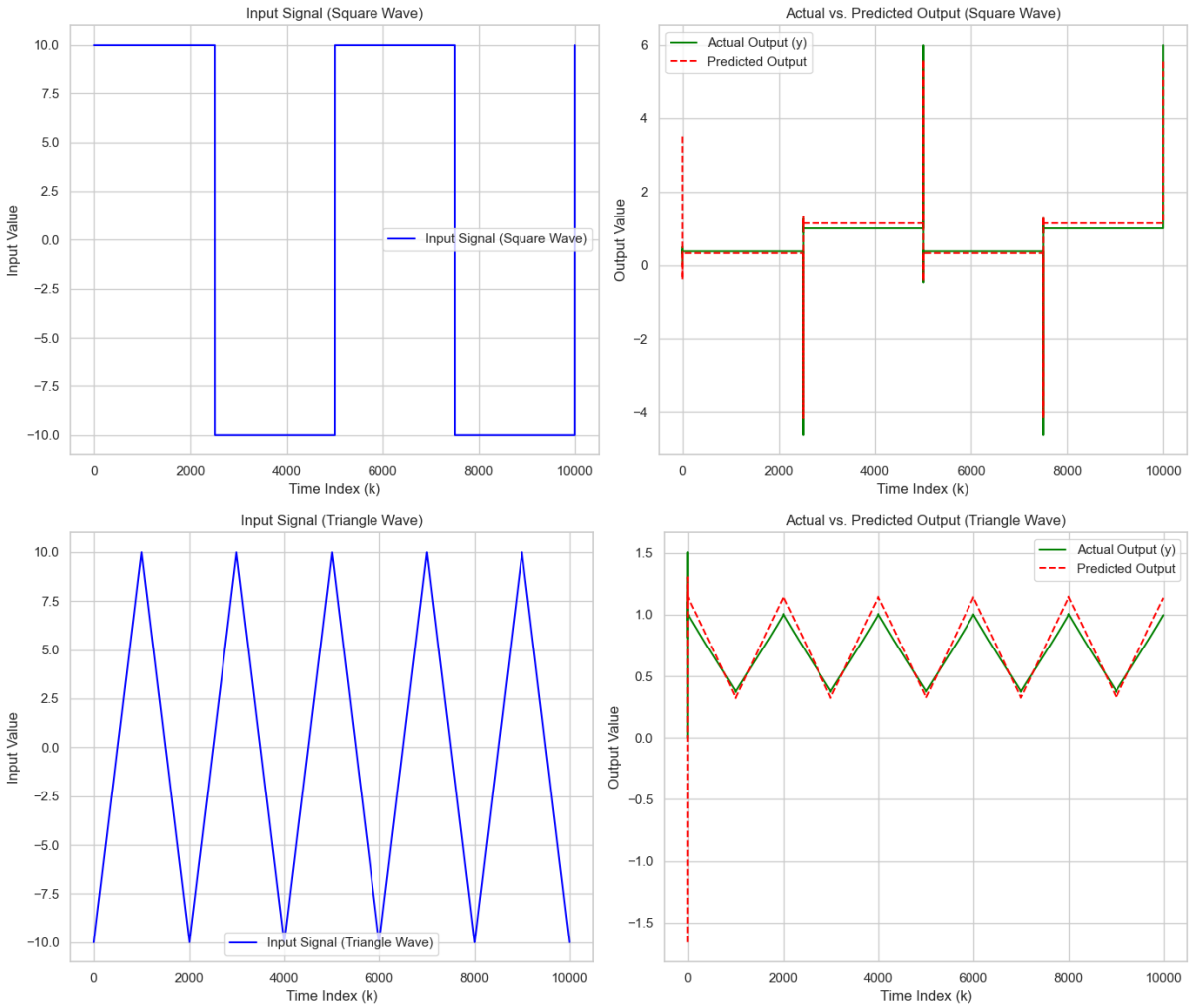




4.1.3. Hasil Testing

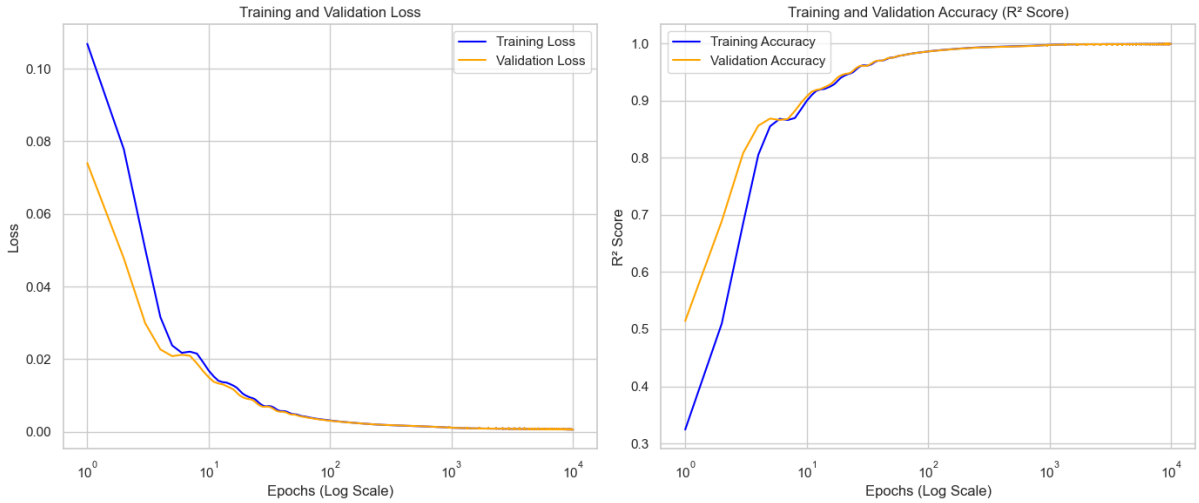




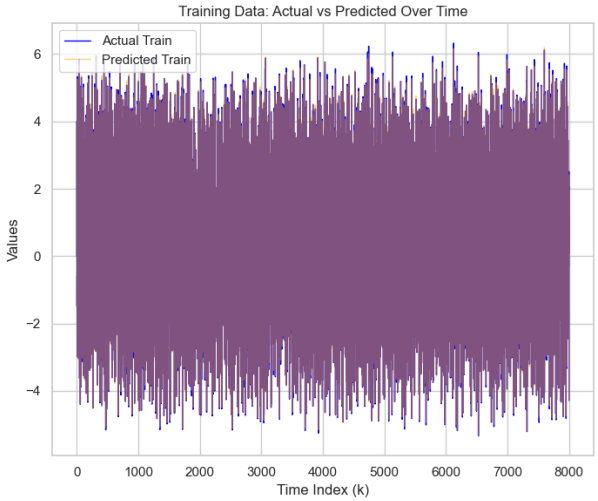


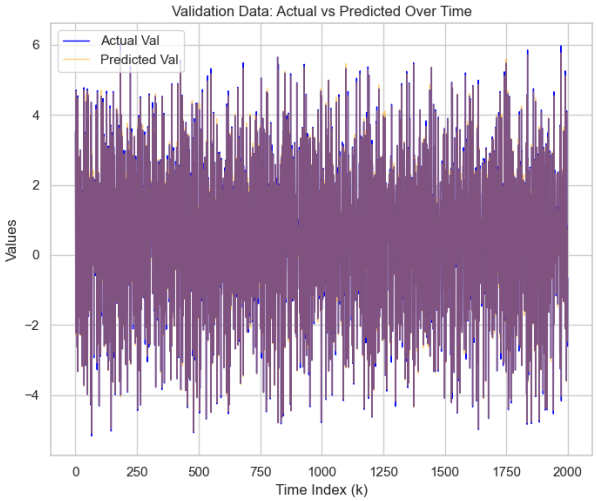
4.2. 3 Hidden Layer dan 7 Neuron

4.2.1. Hasil Tahap 1

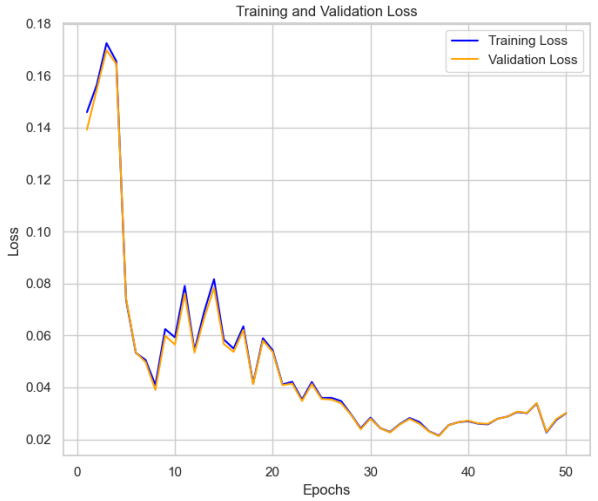


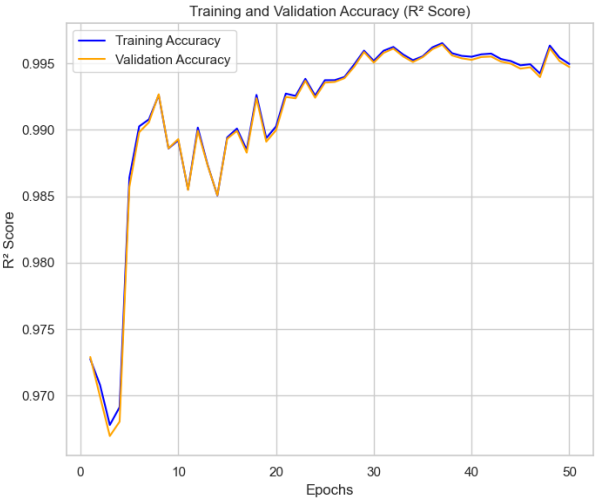
Train Loss: 0.0006, Train Acc: 0.9992, Val Loss: 0.0006, Val Acc: 0.9992



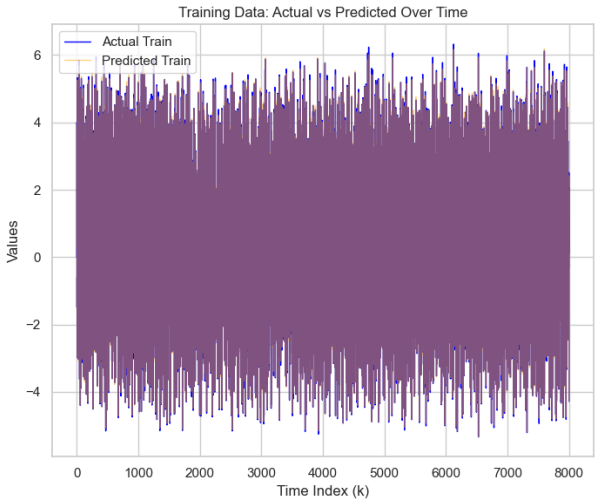


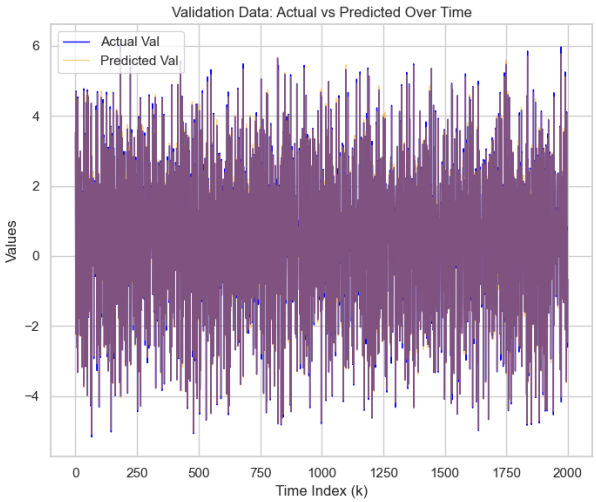
4.2.2. Hasil Tahap 2



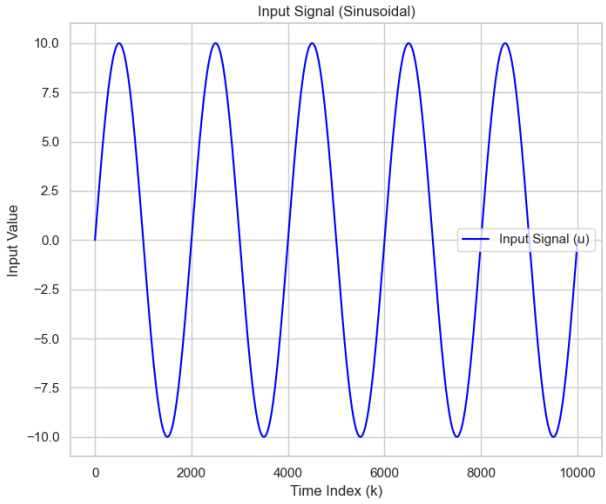


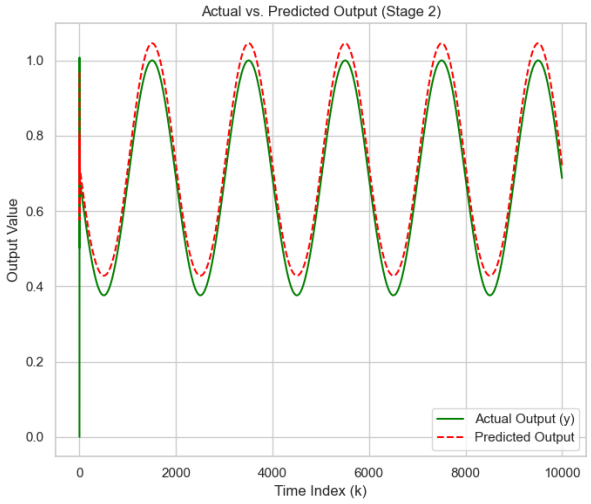
Train Loss: 0.0300, Train Acc: 0.9950, Val Loss: 0.0301, Val Acc: 0.9947

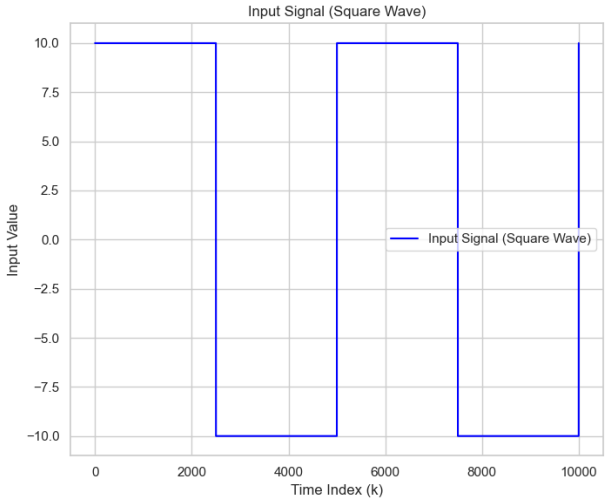


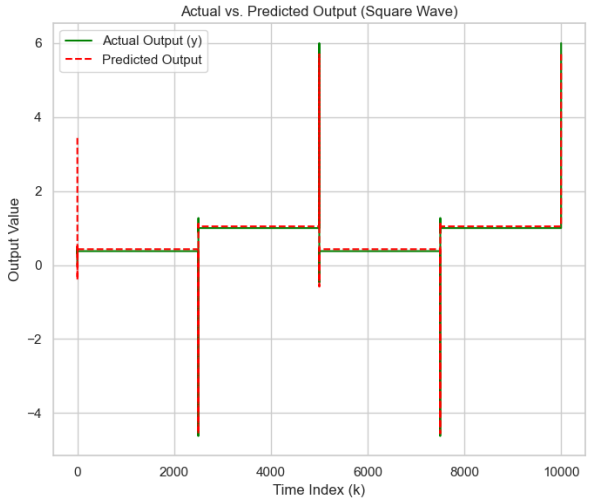


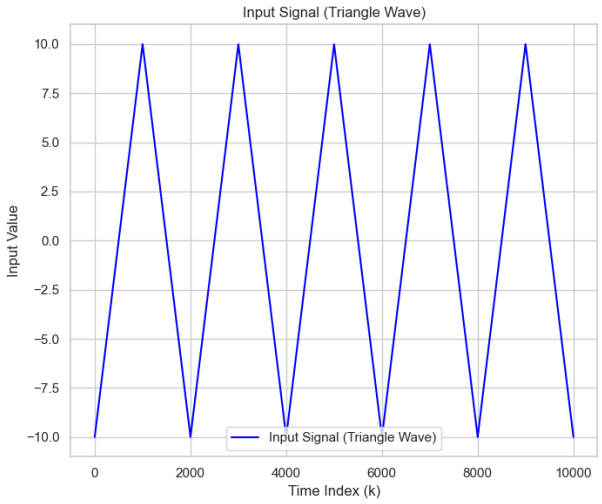
4.2.3. Hasil Testing

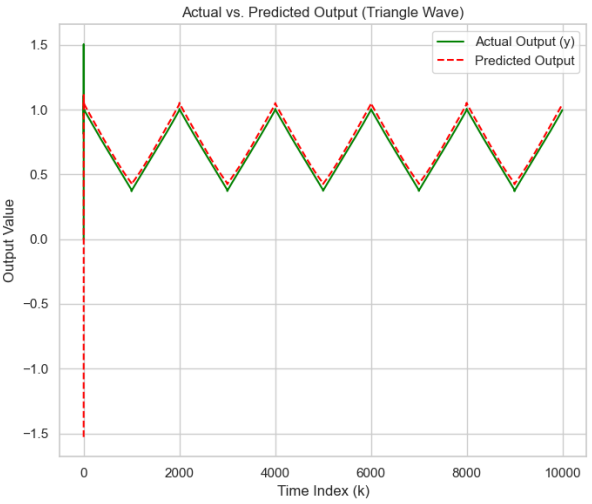






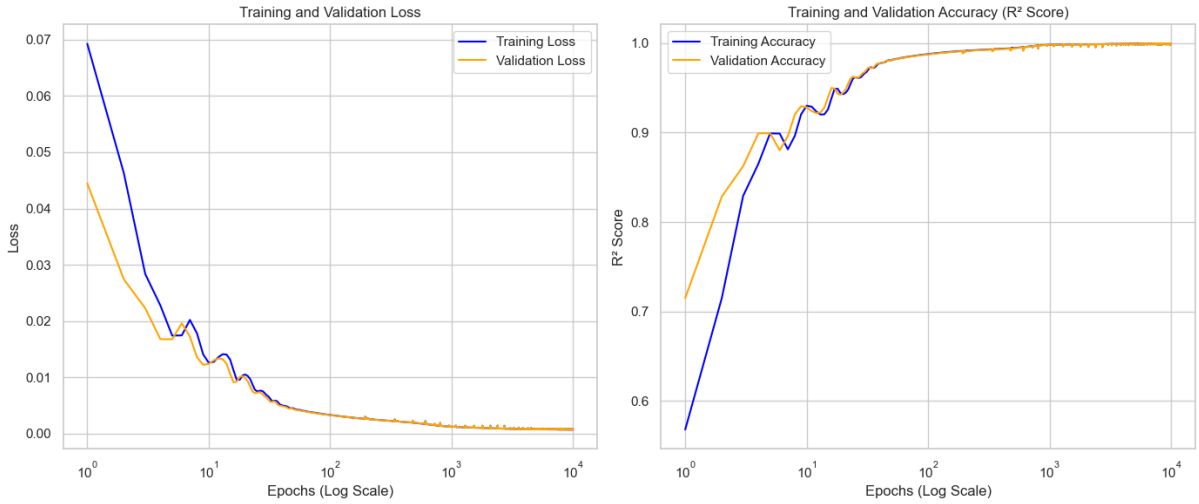




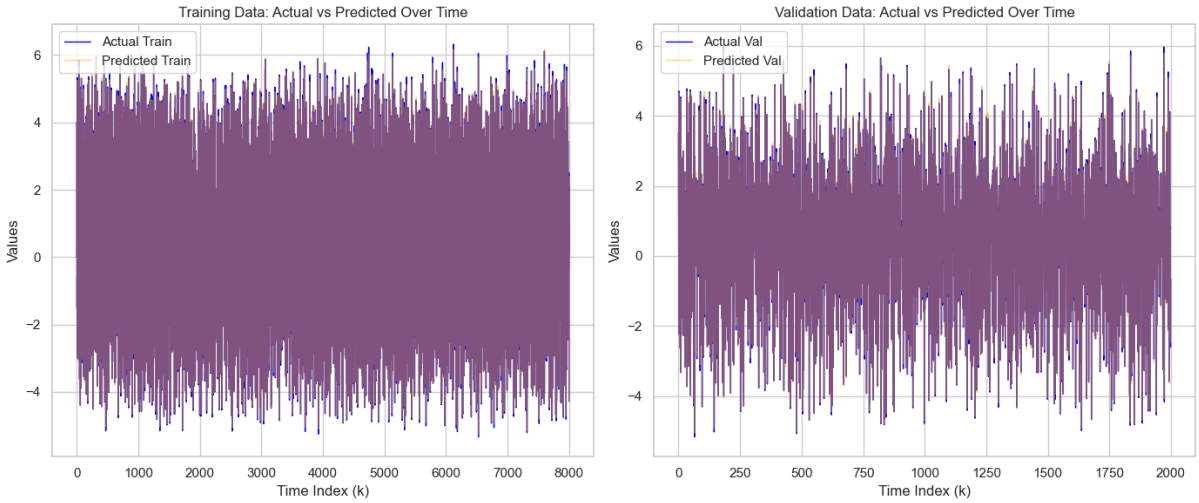


4.3. 5 Hidden Layer dan 7 Neuron

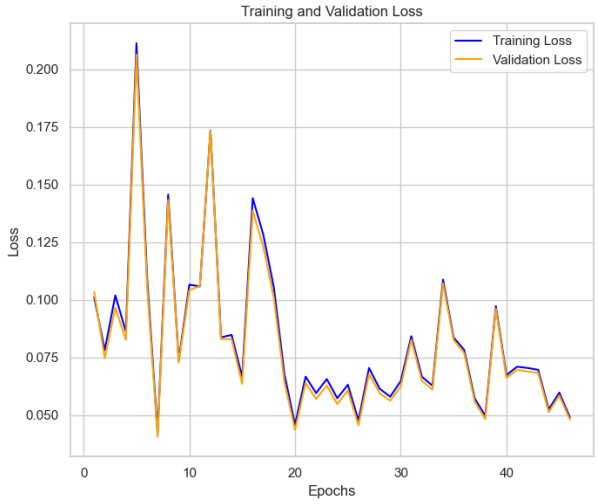
4.3.1. Hasil Tahap 1

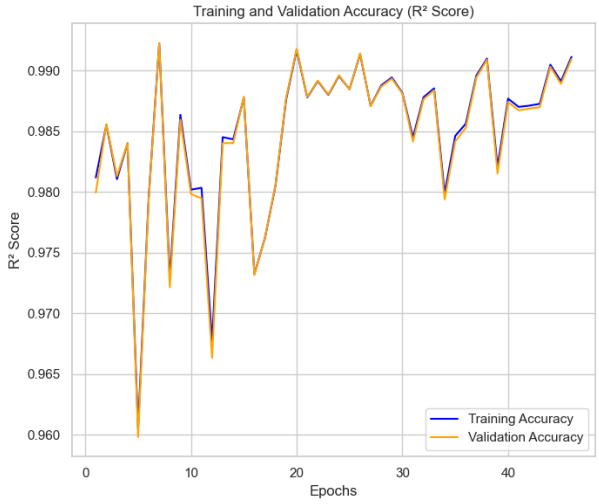


Train Loss: 0.0007, Train Acc: 0.9992, Val Loss: 0.0007, Val Acc: 0.9992

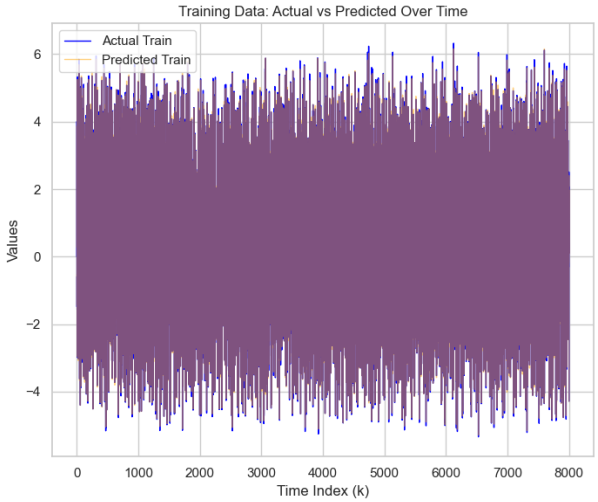


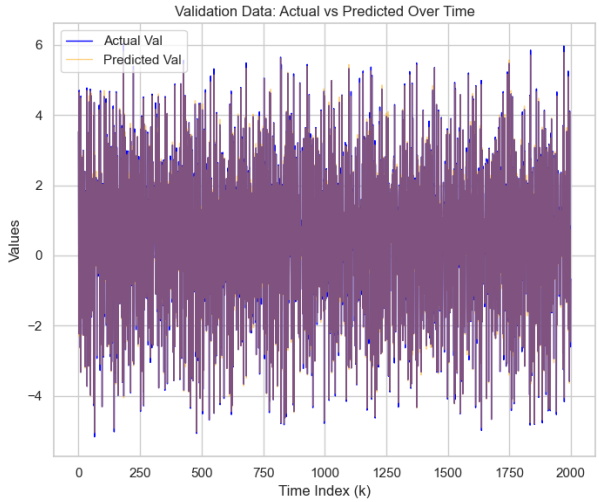
4.3.2. Hasil Tahap 2



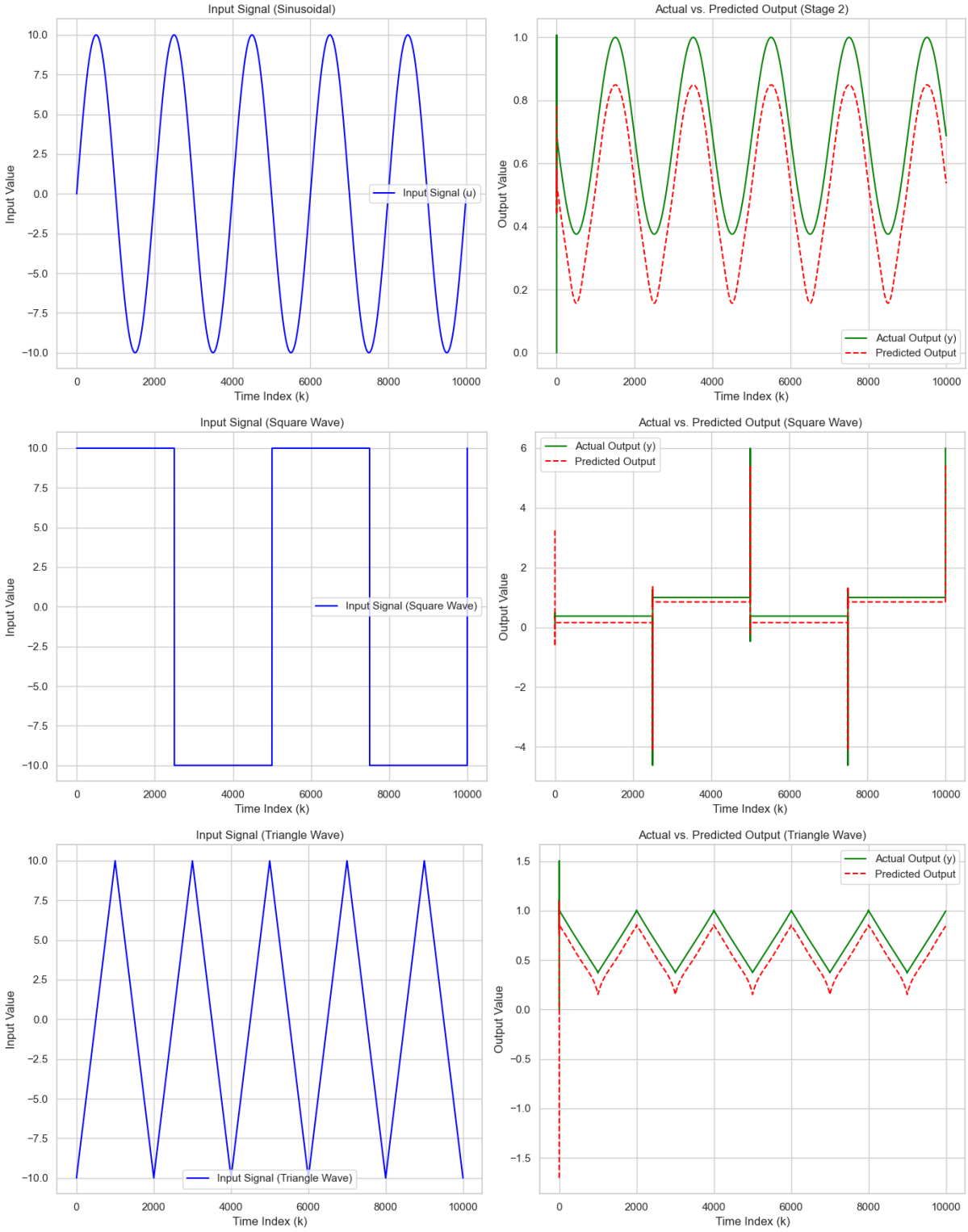


Train Loss: 0.0490, Train Acc: 0.9911, Val Loss: 0.0480, Val Acc: 0.9910





4.3.3. Hasil Testing



4.4. Analisis Perbandingan Hasil Training, Validasi, dan Testing

Pada ketiga percobaan yang dilakukan, penulis menggunakan 10000 data dan 7 neuron pada hidden layer. Perbedaan percobaan 1, 2, dan 3 adalah jumlah hidden layer yang digunakan. Untuk semua hyperparameter dibuat sama yaitu epoch tahap 1 sebanyak

10000, epoch tahap 2 sebanyak 50, learning rate sama 0.01, hyperparameter tuning pada optimizer yang digunakan yaitu Adam Optimizer juga diinisialisasikan dengan cara yang sama, dan koefisien dari L2 Regularization yang juga sama yaitu 0.0001.

Analisis dari eksperimen pelatihan, validasi, dan pengujian menunjukkan wawasan penting tentang bagaimana kompleksitas model, hiperparameter, dan karakteristik sinyal memengaruhi kinerja jaringan saraf dalam tugas identifikasi sistem. Tiga uji coba dilakukan, masing-masing dengan variasi jumlah lapisan tersembunyi (1, 3, dan 5) sambil mempertahankan hiperparameter yang konsisten, termasuk jumlah epoch, laju pembelajaran, pengaturan optimizer, dan teknik regularisasi. Eksperimen ini mengungkap aspek penting seperti keseimbangan antara kapasitas pembelajaran dan overfitting, serta seberapa baik model dapat melakukan generalisasi terhadap pola sinyal yang berbeda.

Percobaan 1 adalah uji coba dengan 1 hidden layer. Konfigurasi model paling sederhana dengan satu lapisan tersembunyi menunjukkan kinerja terbaik pada pola sederhana seperti gelombang sinus. Model ini mencapai nilai loss yang rendah pada pelatihan dan validasi (0,0006) dengan akurasi tinggi sekitar 99,87% untuk pelatihan dan 99,86% untuk validasi. Hal ini menunjukkan bahwa model dapat menangkap fitur utama data secara efisien tanpa mengalami overfitting. Kesederhanaan jaringan membantu model dalam melakukan generalisasi dengan baik karena tidak memiliki kapasitas berlebih untuk mempelajari detail yang tidak relevan atau noise.

Percobaan 2 adalah uji coba dengan 3 hidden layer. Percobaan ini menambahkan lebih banyak lapisan tersembunyi meningkatkan kapasitas pembelajaran model, tetapi juga menyebabkan sedikit peningkatan loss (Train Loss: 0,0148, Val Loss: 0,0140), yang mengindikasikan mulai terjadinya overfitting. Dengan lebih banyak neuron dan lapisan, model dapat mempelajari pola yang lebih kompleks, yang berguna untuk menangani sinyal yang lebih rumit seperti gelombang kotak dan segitiga. Namun, hal ini juga berarti model mungkin menghafal pola atau noise dari data pelatihan yang tidak dapat digeneralisasi dengan baik ke data baru, yang terlihat dari sedikit penurunan akurasi dan peningkatan loss dibandingkan model 1 lapisan.

Percobaan 3 adalah uji coba dengan 5 hidden layer. Model ini adalah paling kompleks dengan 5 lapisan tersembunyi menunjukkan nilai loss tertinggi (Train Loss: 0,0490, Val Loss: 0,0480) dengan akurasi pelatihan dan validasi yang sedikit lebih rendah (sekitar

99,11%). Ini adalah tanda jelas dari overfitting, di mana kompleksitas model memungkinkan pembelajaran detail yang terlalu rumit, termasuk noise, daripada pola umum. Meskipun model memiliki kapasitas untuk menangkap lebih banyak fitur, performanya kurang kuat saat dihadapkan dengan data baru, mencerminkan ketidakmampuannya untuk melakukan generalisasi dengan baik.

Pengamatan ini menunjukkan bahwa peningkatan kompleksitas model dapat meningkatkan kemampuan untuk mempelajari pola kompleks, tetapi juga meningkatkan risiko overfitting. Overfitting terjadi ketika model mempelajari tidak hanya pola pada data, tetapi juga noise, yang mengarah pada kinerja yang lebih buruk pada data yang tidak terlihat sebelumnya. Teknik regularisasi, seperti L2 regularisasi yang digunakan dalam uji coba ini, membantu mengurangi risiko ini, tetapi mungkin tidak cukup jika model terlalu kompleks.

Pelatihan dilakukan dalam dua tahap, dengan 10.000 epoch pada Tahap 1 dan 50 epoch pada Tahap 2. Durasi pelatihan yang lama pada Tahap 1 memastikan model memiliki cukup waktu untuk mempelajari pola dasar, sedangkan Tahap 2 dengan epoch yang lebih sedikit dirancang untuk memperhalus prediksi menggunakan mekanisme umpan balik. Meskipun jumlah epoch yang tinggi dapat memastikan pelatihan menyeluruh, hal ini juga meningkatkan risiko overfitting, karena model memiliki lebih banyak kesempatan untuk menghafal data. Penggunaan monitoring selama pelatihan, seperti early stopping, dapat membantu mencegah hal ini.

Optimizer Adam digunakan di semua uji coba, yang dikenal karena laju pembelajarannya yang adaptif dan membantu mencapai konvergensi yang lebih cepat dan stabil. Meskipun berkontribusi pada akurasi tinggi yang diamati, optimizer ini kurang efektif dalam mencegah overfitting pada model dengan kompleksitas lebih tinggi. Eksplorasi optimizer alternatif seperti RMSProp atau pengaturan momentum adaptif mungkin memberikan manfaat lebih lanjut.

Pilihan laju pembelajaran yaitu 0.01 merupakan laju pembelajaran standar memungkinkan model untuk menyeimbangkan antara pembelajaran yang cepat dan stabil. Laju pembelajaran yang lebih kecil dapat mengurangi pembaruan yang tidak stabil, meningkatkan performa untuk model yang lebih kompleks. Sebaliknya, penyesuaian laju pembelajaran secara dinamis (menurun seiring waktu) mungkin lebih

menyempurnakan proses pelatihan, terutama di Tahap 2, untuk membantu model berkonsentrasi tanpa melampaui target konvergensi.

Fase pengujian menggunakan tiga jenis sinyal—gelombang sinus, kotak, dan segitiga—yang masing-masing menghadirkan tantangan unik bagi model:

- **Gelombang Sinus**: Model 1 lapisan unggul dalam melacak gelombang sinus yang halus dan kontinu. Struktur yang lebih sederhana memungkinkan model untuk melakukan generalisasi dengan baik, menghasilkan prediksi yang halus dan akurat yang sesuai dengan bentuk gelombang asli. Model dengan 3 dan 5 lapisan, meskipun lebih kompleks, menunjukkan sedikit ketidakstabilan dan noise, yang mengindikasikan overfitting. Kompleksitas tambahan tidak memberikan manfaat untuk pola yang sederhana dan dapat diprediksi seperti ini.
- **Gelombang Kotak**: Gelombang kotak memiliki transisi tajam dan tiba-tiba antara nilai tinggi dan rendah, yang menjadi tantangan bagi model. Model 1 lapisan kesulitan di sini, menghasilkan puncak yang membulat dan meratakan transisi tajam, yang menunjukkan bahwa model tidak mampu secara efektif mempelajari perubahan yang mendadak. Model 3 lapisan menunjukkan perbaikan dengan transisi yang lebih tajam, meskipun masih ada sedikit keterlambatan. Model 5 lapisan melacak tepi tajam dengan lebih baik, tetapi ada sedikit noise dan overshoot, menunjukkan bahwa sementara kompleksitas meningkatkan performa, hal itu juga membawa potensi overfitting.
- **Gelombang Segitiga**: Gelombang segitiga memiliki kenaikan dan penurunan linier, yang membutuhkan model untuk menangkap perilaku kontinu non-linier. Model 1 lapisan menunjukkan keterlambatan dan perataan, tidak mampu secara akurat mengikuti pola linier. Model 3 lapisan lebih baik dalam menangkap pola ini, mengurangi keterlambatan, sedangkan model 5 lapisan memberikan aproksimasi paling dekat, mencerminkan transisi linier dengan lebih akurat. Namun, osilasi kecil di sekitar puncak menunjukkan bahwa model yang paling kompleks masih mengalami kecenderungan overfitting.

Dari analisis ini, terlihat bahwa peningkatan kompleksitas model dapat meningkatkan kemampuan untuk mempelajari pola rumit, yang berguna untuk menangani sinyal seperti gelombang kotak dan segitiga. Namun, hal ini juga membawa risiko overfitting, terutama jika pelatihan dilakukan dalam banyak epoch tanpa cukup regularisasi. Teknik regularisasi

tambahan, seperti dropout atau penalti L2 yang lebih agresif, mungkin membantu mengurangi masalah ini. Pengujian dengan berbagai sinyal input menyoroti pentingnya generalisasi model yang baik di berbagai skenario, mengungkap kekuatan dan keterbatasan model. Model yang lebih sederhana (1 lapisan) bekerja terbaik pada sinyal yang halus dan sederhana, sementara model yang lebih kompleks (3 dan 5 lapisan) lebih baik menangani pola yang rumit tetapi menunjukkan tanda-tanda overfitting. Berdasarkan analisis tersebut, sebaiknya pada percobaan berikutnya penulis mempertimbangkan penggunaan laju pembelajaran dinamis, early stopping, dan metode regularisasi yang lebih beragam untuk mengoptimalkan performa. Selain itu, uji coba yang lebih ketat dengan sinyal berbeda menekankan kebutuhan akan arsitektur model yang seimbang, yang dapat melakukan generalisasi dengan baik di berbagai situasi, memastikan identifikasi sistem yang konsisten dan andal. Penambahan pengendali berbasis neural network juga dapat menjadi kelanjutan untuk memperbaiki model identifikasi sistem yang sudah dirancang, hal ini karena model sistem identifikasi pada tahap kedua merupakan model menggunakan feedback loop dimana feedback loop yang digunakan adalah unity feedback loop atau dapat dikatakan sebagai sistem kendali loop tertutup atau closed loop control yang belum memiliki controller dan masih hanya plant saja. Oleh karena itu, memungkinkan muncul fenomena seperti overshoot, undershoot, steady state error, dan lainnya pada response sistem plant tersebut.

BAB 5
KESIMPULAN

5.1. Kesimpulan

5.1.1. Neural network berhasil mengidentifikasi sistem dinamik diskrit secara efektif. Model dapat mempelajari pola input-output dengan baik, terutama pada sinyal yang sederhana dan kontinu seperti gelombang sinus.

5.1.2. Neural network mampu memahami hubungan input-output pada sistem dengan umpan balik. Eksperimen di Tahap 2 menunjukkan bahwa mekanisme umpan balik membantu meningkatkan akurasi prediksi, terutama untuk pola dinamis yang kompleks.

5.1.3. Model menunjukkan akurasi tinggi (>99%) dalam memprediksi sinyal uji, tetapi peningkatan kompleksitas model (lebih banyak lapisan) tidak selalu meningkatkan performa. Model sederhana (1 lapisan) lebih baik dalam generalisasi sinyal sederhana, sedangkan model lebih kompleks (3-5 lapisan) menangani pola yang lebih kompleks tetapi rentan terhadap overfitting.

5.1.4. Neural network efektif dalam memodelkan sistem non-linear dan kompleks, terbukti dari kemampuannya memprediksi sinyal seperti gelombang kotak dan segitiga. Namun, pengelolaan kompleksitas model penting untuk menghindari overfitting dan memastikan kemampuan generalisasi yang baik.

5.2. Saran

5.2.1. Optimalisasi Arsitektur Model: Perlu dilakukan eksplorasi lebih lanjut terhadap konfigurasi arsitektur neural network, seperti jumlah lapisan dan neuron, untuk menemukan keseimbangan optimal antara kompleksitas dan kemampuan generalisasi. Penggunaan teknik seperti dropout atau batch normalization dapat membantu mengurangi overfitting.

5.2.2. Pengaturan Hiperparameter yang Lebih Baik: Disarankan untuk mencoba teknik pengaturan laju pembelajaran dinamis (learning rate schedules) atau algoritma optimasi lainnya, seperti RMSProp atau SGD dengan momentum, untuk meningkatkan efisiensi pelatihan dan stabilitas model. Penggunaan early stopping juga dapat mencegah pelatihan berlebihan (overfitting).

5.2.3. Eksplorasi Teknik Regularisasi: Untuk mengatasi overfitting, penggunaan regularisasi yang lebih kuat, seperti L1 regularisasi atau dropout, dapat dieksplorasi. Hal ini akan membantu model untuk lebih fokus pada pola yang relevan dan menghindari belajar dari noise dalam data.

5.2.4. Pengujian dengan Data yang Lebih Beragam: Perluasan jenis sinyal uji, termasuk sinyal yang lebih kompleks atau non-periodik, akan memberikan pemahaman lebih mendalam tentang kemampuan model untuk melakukan generalisasi. Hal ini juga dapat membantu mengidentifikasi keterbatasan model dalam memodelkan pola dinamis yang lebih rumit.

5.2.5. Penggunaan Teknik Feedback yang Lebih Canggih: Mempertimbangkan penggunaan teknik feedback yang lebih canggih, seperti Long Short-Term Memory (LSTM) atau Recurrent Neural Networks (RNNs), dapat meningkatkan kemampuan model dalam menangani pola dinamis yang bergantung pada nilai sebelumnya. Ini akan sangat berguna untuk aplikasi di mana ketergantungan temporal sangat kuat.

5.2.6. Pengembangan Pengendali Berbasis Neural Network: Mengembangkan pengendali berbasis neural network dapat menjadi langkah lanjut yang menarik, karena model saat ini masih berupa unity feedback loop control. Dengan menambahkan pengendali yang memanfaatkan hasil identifikasi sistem, model neural network dapat digunakan untuk mengatur parameter kontrol secara adaptif dan lebih responsif terhadap perubahan kondisi sistem. Hal ini berpotensi meningkatkan performa keseluruhan sistem dinamik, terutama pada aplikasi yang membutuhkan respons adaptif dan prediktif.

DAFTAR PUSTAKA

S. Haykin, "Neural Networks: A Comprehensive Foundation," 2nd ed., Prentice Hall, 1999.

T. M. Mitchell, "Machine Learning," McGraw Hill, 1997.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," Nature, vol. 323, no. 6088, pp. 533-536, Oct. 1986.

Y. LeCun, L. Bottou, G. B. Orr, and K. R. Müller, "Efficient BackProp," in Neural Networks: Tricks of the Trade, Springer, 2012, pp. 9-48.

I. Goodfellow, Y. Bengio, and A. Courville, "Deep Learning," MIT Press, 2016.

D. Nguyen and B. Widrow, "Improving the Learning Speed of 2-Layer Neural Networks by Choosing Initial Values of the Adaptive Weights," in Proceedings of the IJCNN, 1990, vol. 3, pp. 21-26.

S. Ruder, "An overview of gradient descent optimization algorithms," arXiv preprint arXiv:1609.04747, 2016.

A. Ng, "Machine Learning Yearning," Deeplearning.ai, 2019.

K. S. Narendra and K. Parthasarathy, "Identification and Control of Dynamical Systems Using Neural Networks," IEEE Transactions on Neural Networks, vol. 1, no. 1, pp. 4-27, 1990.

LAMPIRAN KODE PROGRAM

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from scipy import signal

# Set a random seed for reproducibility
np.random.seed(42)

# Define the number of samples
k_max = 10000
t = np.linspace(0, k_max, k_max)  # Time variable

# Generate random inputs
u = 2 * np.random.uniform(-5, 5, k_max)

# Initialize y(k) array
y = np.zeros(k_max)

# Define y(k) calculations
for k in range(1, k_max):
    y[k] = 1 / (1 + (y[k-1])**2) + 0.25 * u[k] - 0.3 * u[k-1]

# Shift arrays for y(k-1), y(k-2), u(k-1), u(k-2)
y_k_1 = np.zeros(k_max)
y_k_2 = np.zeros(k_max)
u_k_1 = np.zeros(k_max)
u_k_2 = np.zeros(k_max)

y_k_1[1:] = y[:-1]
y_k_2[2:] = y[:-2]
u_k_1[1:] = u[:-1]
u_k_2[2:] = u[:-2]

# Set up Seaborn style
sns.set_theme(style="whitegrid")

# Create a single figure with a 3x2 grid of subplots
fig, ax = plt.subplots(3, 2, figsize=(15, 15), sharex=True)

# Plot u(k) over time
ax[0, 0].plot(t, u, label='u(k)', color='c')
ax[0, 0].set_ylabel('u(k)')
ax[0, 0].set_title('Plot of u(k) over Time')
ax[0, 0].legend()
ax[0, 0].grid(True)

# Plot y(k) over time
ax[0, 1].plot(t, y, label='y(k)', color='b')
ax[0, 1].set_ylabel('y(k)')
ax[0, 1].set_title('Plot of y(k) over Time')
ax[0, 1].legend()
ax[0, 1].grid(True)

# Plot u(k-1) over time
ax[1, 0].plot(t, u_k_1, label='u(k-1)', color='m')
ax[1, 0].set_ylabel('u(k-1)')
ax[1, 0].set_title('Plot of u(k-1) over Time')
ax[1, 0].legend()
ax[1, 0].grid(True)

# Plot y(k-1) over time
ax[1, 1].plot(t, y_k_1, label='y(k-1)', color='r')
ax[1, 1].set_ylabel('y(k-1)')
ax[1, 1].set_title('Plot of y(k-1) over Time')
ax[1, 1].legend()
ax[1, 1].grid(True)

# Plot u(k-2) over time
ax[2, 0].plot(t, u_k_2, label='u(k-2)', color='y')
ax[2, 0].set_xlabel('Time')
ax[2, 0].set_ylabel('u(k-2)')
ax[2, 0].set_title('Plot of u(k-2) over Time')
ax[2, 0].legend()
```

```
ax[2, 0].grid(True)

# Plot y(k-2) over time
ax[2, 1].plot(t, y_k_2, label='y(k-2)', color='g')
ax[2, 1].set_ylabel('y(k-2)')
ax[2, 1].set_title('Plot of y(k-2) over Time')
ax[2, 1].legend()
ax[2, 1].grid(True)

# Show the combined plot
plt.tight_layout()
plt.show()

X = np.array([y_k_1, y_k_2, u, u_k_1, u_k_2]).T
y = y.reshape(-1, 1)
u = u.reshape(-1, 1)

scaler = MinMaxScaler(feature_range=(-1, 1))

scaler.fit(u)
u_norm = scaler.transform(u.reshape(-1, 1))
u_k_1_norm = scaler.transform(u_k_1.reshape(-1, 1))
u_k_2_norm = scaler.transform(u_k_2.reshape(-1, 1))

scaler.fit(y)
y_norm = scaler.transform(y.reshape(-1, 1))
y_k_1_norm = scaler.transform(y_k_1.reshape(-1, 1))
y_k_2_norm = scaler.transform(y_k_2.reshape(-1, 1))

X_norm = np.array([y_k_1_norm.flatten(), y_k_2_norm.flatten(), u_norm.flatten(),
u_k_1_norm.flatten(), u_k_2_norm.flatten()]).T

# Assuming X and y are your feature matrix and target vector respectively
train_size = int(0.8 * len(X))  # 80% of the data

# First 80% of the data for training
X_train = X[:train_size]
y_train = y[:train_size]

# Last 20% of the data for validation
X_val = X[train_size:]
y_val = y[train_size:]

# For the normalized data
X_train_norm = X_norm[:train_size]
y_train_norm = y_norm[:train_size]
X_val_norm = X_norm[train_size:]
y_val_norm = y_norm[train_size:]


class MLP:
    def __init__(self, layers, activations, optimizer='adam', learning_rate=0.001,
l2_lambda=0.01):
        self.layers = layers
        self.activations = activations
        self.learning_rate = learning_rate
        self.l2_lambda = l2_lambda  # Regularization parameter
        self.weights, self.biases = self.initialize_weights()
        self.optimizer = optimizer
        self.momentum_w, self.momentum_b = None, None
        self.v_w, self.v_b = None, None
        self.initialize_optimizer_parameters()
        # Arrays to store training and validation loss/accuracy
        self.training_losses = []
        self.validation_losses = []
        self.training_accuracies = []
        self.validation_accuracies = []

    def initialize_weights(self):
        weights = []
        biases = []
        for i in range(len(self.layers) - 1):
            input_dim = self.layers[i]
            output_dim = self.layers[i + 1]

            # Nguyen-Widrow initialization
            # Step 1: Randomly initialize weights from a uniform distribution
```

```
            w = np.random.uniform(-1, 1, (input_dim, output_dim))

            # Step 2: Compute the scaling factor `beta`
            beta = 0.7 * output_dim ** (1.0 / input_dim)

            # Step 3: Normalize the weights to have unit length for each column
            norm = np.linalg.norm(w, axis=0)
            w = beta * (w / norm)

            # Initialize biases to zero
            b = np.zeros((1, output_dim))

            weights.append(w)
            biases.append(b)

        return weights, biases

    def initialize_optimizer_parameters(self):
        if self.optimizer in ['adam', 'rmsprop']:
            self.momentum_w = [np.zeros_like(w) for w in self.weights]
            self.momentum_b = [np.zeros_like(b) for b in self.biases]
            self.v_w = [np.zeros_like(w) for w in self.weights]
            self.v_b = [np.zeros_like(b) for b in self.biases]

        self.beta1 = 0.9
        self.beta2 = 0.999
        self.epsilon = 1e-8

    def activation(self, x, func):
        if func == 'tanh':
            return np.tanh(x)
        elif func == 'linear':
            return x

    def activation_derivative(self, x, func):
        if func == 'tanh':
            return 1 - np.tanh(x) ** 2
        elif func == 'linear':
            return np.ones_like(x)

    def forward(self, x):
        self.z_list = []
        self.a_list = [x]

        for i in range(len(self.weights)):
            z = np.dot(x, self.weights[i]) + self.biases[i]
            x = self.activation(z, self.activations[i])
            self.z_list.append(z)
            self.a_list.append(x)
        return x

    def backward(self, y_true, y_pred):
        gradients_w = []
        gradients_b = []
        loss_derivative = 2 * (y_pred - y_true) / y_true.shape[0]

        # Output layer
        delta = loss_derivative * self.activation_derivative(self.z_list[-1],
self.activations[-1])
        grad_w = np.dot(self.a_list[-2].T, delta) + self.l2_lambda * self.weights[-1]
        grad_b = np.sum(delta, axis=0, keepdims=True)

        gradients_w.insert(0, grad_w)
        gradients_b.insert(0, grad_b)

        # Hidden layers
        for i in range(len(self.layers) - 3, -1, -1):
            delta = np.dot(delta, self.weights[i + 1].T) *
self.activation_derivative(self.z_list[i], self.activations[i])
            grad_w = np.dot(self.a_list[i].T, delta) + self.l2_lambda * self.weights[i]
            grad_b = np.sum(delta, axis=0, keepdims=True)
            gradients_w.insert(0, grad_w)
            gradients_b.insert(0, grad_b)

        return gradients_w, gradients_b

    def update_weights(self, gradients_w, gradients_b):
```

```
        if self.optimizer == 'gradient_descent':
            for i in range(len(self.weights)):
                self.weights[i] -= self.learning_rate * gradients_w[i]
                self.biases[i] -= self.learning_rate * gradients_b[i]
        elif self.optimizer == 'sgd':
            for i in range(len(self.weights)):
                self.weights[i] -= self.learning_rate * gradients_w[i]
                self.biases[i] -= self.learning_rate * gradients_b[i]
        elif self.optimizer == 'rmsprop':
            for i in range(len(self.weights)):
                self.momentum_w[i] = self.beta2 * self.momentum_w[i] + (1 - self.beta2) *
gradients_w[i]**2
                self.momentum_b[i] = self.beta2 * self.momentum_b[i] + (1 - self.beta2) *
gradients_b[i]**2

                self.weights[i] -= self.learning_rate * gradients_w[i] /
(np.sqrt(self.momentum_w[i]) + self.epsilon)
                self.biases[i] -= self.learning_rate * gradients_b[i] /
(np.sqrt(self.momentum_b[i]) + self.epsilon)
        elif self.optimizer == 'adam':
            for i in range(len(self.weights)):
                self.momentum_w[i] = self.beta1 * self.momentum_w[i] + (1 - self.beta1) *
gradients_w[i]
                self.momentum_b[i] = self.beta1 * self.momentum_b[i] + (1 - self.beta1) *
gradients_b[i]

                self.v_w[i] = self.beta2 * self.v_w[i] + (1 - self.beta2) * gradients_w[i]**2
                self.v_b[i] = self.beta2 * self.v_b[i] + (1 - self.beta2) * gradients_b[i]**2

                m_w_corrected = self.momentum_w[i] / (1 - self.beta1)
                m_b_corrected = self.momentum_b[i] / (1 - self.beta1)
                v_w_corrected = self.v_w[i] / (1 - self.beta2)
                v_b_corrected = self.v_b[i] / (1 - self.beta2)

                self.weights[i] -= self.learning_rate * m_w_corrected /
(np.sqrt(v_w_corrected) + self.epsilon)
                self.biases[i] -= self.learning_rate * m_b_corrected / (np.sqrt(v_b_corrected)
+ self.epsilon)

    def calculate_loss(self, y_true, y_pred):
        return np.mean((y_pred - y_true) ** 2) + (self.l2_lambda / 2) * sum([np.sum(w ** 2)
for w in self.weights])

    def calculate_accuracy(self, y_true, y_pred):
        ss_res = np.sum((y_true - y_pred) ** 2)
        ss_tot = np.sum((y_true - np.mean(y_true)) ** 2)
        r2_score = 1 - (ss_res / ss_tot)
        return r2_score

    def train(self, x_train, y_train, x_val=None, y_val=None, epochs=100):
        for epoch in range(epochs):
            # Training phase
            y_pred_train = self.forward(x_train)
            gradients_w, gradients_b = self.backward(y_train, y_pred_train)
            self.update_weights(gradients_w, gradients_b)

            # Calculate training loss and accuracy
            train_loss = self.calculate_loss(y_train, y_pred_train)
            train_acc = self.calculate_accuracy(y_train, y_pred_train)
            self.training_losses.append(train_loss)
            self.training_accuracies.append(train_acc)

            # Validation phase (if validation data provided)
            if x_val is not None and y_val is not None:
                y_pred_val = self.forward(x_val)
                val_loss = self.calculate_loss(y_val, y_pred_val)
                val_acc = self.calculate_accuracy(y_val, y_pred_val)
                self.validation_losses.append(val_loss)
                self.validation_accuracies.append(val_acc)

                print(f"Epoch {epoch + 1}/{epochs} - Train Loss: {train_loss:.4f}, Train Acc:
{train_acc:.4f}, "
                      f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}")
            else:
                print(f"Epoch {epoch + 1}/{epochs} - Train Loss: {train_loss:.4f}, Train Acc:
{train_acc:.4f}")
```

```
    def predict(self, x):
        return self.forward(x)

    def get_weights(self):
        return self.weights

    def get_biases(self):
        return self.biases

    def set_weights(self, weights):
        self.weights = weights

    def set_biases(self, biases):
        self.biases = biases

nn = MLP(layers=[X.shape[-1], 7, 7, 7, 7, 7, 1], activations=['tanh', 'tanh', 'tanh', 'tanh',
'tanh', 'linear'], optimizer='adam', learning_rate=0.01, l2_lambda=0.0001)

nn.train(X_train_norm, y_train_norm, x_val=X_val_norm, y_val=y_val_norm, epochs=10000)

training_losses = nn.training_losses
validation_losses = nn.validation_losses
training_accuracies = nn.training_accuracies
validation_accuracies = nn.validation_accuracies

# Create a figure with 1 row and 2 columns for loss and accuracy plots
fig, ax = plt.subplots(1, 2, figsize=(14, 6))

# Create a range of epochs for the x-axis
epochs = range(1, len(training_losses) + 1)

# Plot Loss with Logarithmic X-Axis
sns.lineplot(x=epochs, y=training_losses, ax=ax[0], label='Training Loss', color='blue')
sns.lineplot(x=epochs, y=validation_losses, ax=ax[0], label='Validation Loss', color='orange')
ax[0].set_xscale('log')  # Set x-axis to logarithmic scale
ax[0].set_xlabel('Epochs (Log Scale)')
ax[0].set_ylabel('Loss')
ax[0].set_title('Training and Validation Loss')
ax[0].legend()

# Plot Accuracy with Logarithmic X-Axis
sns.lineplot(x=epochs, y=training_accuracies, ax=ax[1], label='Training Accuracy',
color='blue')
sns.lineplot(x=epochs, y=validation_accuracies, ax=ax[1], label='Validation Accuracy',
color='orange')
ax[1].set_xscale('log')  # Set x-axis to logarithmic scale
ax[1].set_xlabel('Epochs (Log Scale)')
ax[1].set_ylabel('R² Score')
ax[1].set_title('Training and Validation Accuracy (R² Score)')
ax[1].legend()

# Adjust layout and show the plots
plt.tight_layout()
plt.show()


y_pred_train_norm = nn.predict(X_train_norm)
y_pred_val_norm = nn.predict(X_val_norm)
y_pred_train = scaler.inverse_transform(y_pred_train_norm)
y_pred_val = scaler.inverse_transform(y_pred_val_norm)

# Assuming y_train, y_pred_train, y_val, and y_pred_val are already defined
fig, ax = plt.subplots(1, 2, figsize=(14, 6))

# Generate time indices for training and validation
train_indices = range(len(y_train))
val_indices = range(len(y_val))

# Plot y_pred_train and y_train over time (k) with reduced alpha for predicted line
sns.lineplot(x=train_indices, y=y_train.flatten(), ax=ax[0], color='blue', label='Actual
Train', linewidth=1)
sns.lineplot(x=train_indices, y=y_pred_train.flatten(), ax=ax[0], color='orange',
label='Predicted Train', alpha=0.5, linewidth=1)
ax[0].set_xlabel('Time Index (k)')
ax[0].set_ylabel('Values')
ax[0].set_title('Training Data: Actual vs Predicted Over Time')
ax[0].legend()
```

```
# Plot y_pred_val and y_val over time (k) with reduced alpha for predicted line
sns.lineplot(x=val_indices, y=y_val.flatten(), ax=ax[1], color='blue', label='Actual Val',
linewidth=1)
sns.lineplot(x=val_indices, y=y_pred_val.flatten(), ax=ax[1], color='orange', label='Predicted
Val', alpha=0.5, linewidth=1)
ax[1].set_xlabel('Time Index (k)')
ax[1].set_ylabel('Values')
ax[1].set_title('Validation Data: Actual vs Predicted Over Time')
ax[1].legend()

# Show the updated plots
plt.tight_layout()
plt.show()

weights = nn.get_weights()
biases = nn.get_biases()

class MLPVectoral:
    def __init__(self, layers, activations, optimizer='adam', learning_rate=0.001,
l2_lambda=0.01, weights=None, biases=None):
        self.layers = layers
        self.activations = activations
        self.learning_rate = learning_rate
        self.l2_lambda = l2_lambda  # Regularization parameter
        self.weights = weights if weights else self.initialize_weights()[0]
        self.biases = biases if biases else self.initialize_weights()[1]
        self.optimizer = optimizer
        self.momentum_w, self.momentum_b = None, None
        self.v_w, self.v_b = None, None
        self.initialize_optimizer_parameters()
        # Arrays to store training and validation loss/accuracy
        self.training_losses = []
        self.validation_losses = []
        self.training_accuracies = []
        self.validation_accuracies = []

    def initialize_weights(self):
        weights = []
        biases = []
        for i in range(len(self.layers) - 1):
            input_dim = self.layers[i]
            output_dim = self.layers[i + 1]

            # Nguyen-Widrow initialization
            w = np.random.uniform(-1, 1, (input_dim, output_dim))
            beta = 0.7 * output_dim ** (1.0 / input_dim)
            norm = np.linalg.norm(w, axis=0)
            w = beta * (w / norm)

            b = np.zeros((1, output_dim))
            weights.append(w)
            biases.append(b)

        return weights, biases

    def initialize_optimizer_parameters(self):
        if self.optimizer in ['adam', 'rmsprop']:
            self.momentum_w = [np.zeros_like(w) for w in self.weights]
            self.momentum_b = [np.zeros_like(b) for b in self.biases]
            self.v_w = [np.zeros_like(w) for w in self.weights]
            self.v_b = [np.zeros_like(b) for b in self.biases]

        self.beta1 = 0.9
        self.beta2 = 0.999
        self.epsilon = 1e-8

    def activation(self, x, func):
        if func == 'tanh':
            return np.tanh(x)
        elif func == 'linear':
            return x

    def activation_derivative(self, x, func):
        if func == 'tanh':
            return 1 - np.tanh(x) ** 2
        elif func == 'linear':
```

```
            return np.ones_like(x)

    def forward(self, x):
        self.z_list = []
        self.a_list = [x]

        for i in range(len(self.weights)):
            z = np.dot(x, self.weights[i]) + self.biases[i]
            x = self.activation(z, self.activations[i])
            self.z_list.append(z)
            self.a_list.append(x)
        return x

    def backward(self, y_true, y_pred):
        gradients_w = []
        gradients_b = []
        loss_derivative = 2 * (y_pred - y_true)

        # Output layer
        delta = loss_derivative * self.activation_derivative(self.z_list[-1],
self.activations[-1])
        grad_w = np.dot(self.a_list[-2].T, delta) + self.l2_lambda * self.weights[-1]
        grad_b = np.sum(delta, axis=0, keepdims=True)

        gradients_w.insert(0, grad_w)
        gradients_b.insert(0, grad_b)

        # Hidden layers
        for i in range(len(self.layers) - 3, -1, -1):
            delta = np.dot(delta, self.weights[i + 1].T) *
self.activation_derivative(self.z_list[i], self.activations[i])
            grad_w = np.dot(self.a_list[i].T, delta) + self.l2_lambda * self.weights[i]
            grad_b = np.sum(delta, axis=0, keepdims=True)
            gradients_w.insert(0, grad_w)
            gradients_b.insert(0, grad_b)

        return gradients_w, gradients_b

    def update_weights(self, gradients_w, gradients_b):
        if self.optimizer == 'gradient_descent' or self.optimizer == 'sgd':
            for i in range(len(self.weights)):
                self.weights[i] -= self.learning_rate * gradients_w[i]
                self.biases[i] -= self.learning_rate * gradients_b[i]
        elif self.optimizer == 'rmsprop':
            for i in range(len(self.weights)):
                self.momentum_w[i] = self.beta2 * self.momentum_w[i] + (1 - self.beta2) *
gradients_w[i]**2
                self.momentum_b[i] = self.beta2 * self.momentum_b[i] + (1 - self.beta2) *
gradients_b[i]**2

                self.weights[i] -= self.learning_rate * gradients_w[i] /
(np.sqrt(self.momentum_w[i]) + self.epsilon)
                self.biases[i] -= self.learning_rate * gradients_b[i] /
(np.sqrt(self.momentum_b[i]) + self.epsilon)
        elif self.optimizer == 'adam':
            for i in range(len(self.weights)):
                self.momentum_w[i] = self.beta1 * self.momentum_w[i] + (1 - self.beta1) *
gradients_w[i]
                self.momentum_b[i] = self.beta1 * self.momentum_b[i] + (1 - self.beta1) *
gradients_b[i]

                self.v_w[i] = self.beta2 * self.v_w[i] + (1 - self.beta2) * gradients_w[i]**2
                self.v_b[i] = self.beta2 * self.v_b[i] + (1 - self.beta2) * gradients_b[i]**2

                m_w_corrected = self.momentum_w[i] / (1 - self.beta1)
                m_b_corrected = self.momentum_b[i] / (1 - self.beta1)
                v_w_corrected = self.v_w[i] / (1 - self.beta2)
                v_b_corrected = self.v_b[i] / (1 - self.beta2)

                self.weights[i] -= self.learning_rate * m_w_corrected /
(np.sqrt(v_w_corrected) + self.epsilon)
                self.biases[i] -= self.learning_rate * m_b_corrected / (np.sqrt(v_b_corrected)
+ self.epsilon)

    def calculate_loss(self, y_true, y_pred):
        return np.mean((y_pred - y_true) ** 2) + (self.l2_lambda / 2) * sum([np.sum(w ** 2)
for w in self.weights])
```

```
    def calculate_accuracy(self, y_true, y_pred):
        ss_res = np.sum((y_true - y_pred) ** 2)
        ss_tot = np.sum((y_true - np.mean(y_true)) ** 2)
        r2_score = 1 - (ss_res / ss_tot)
        return r2_score

    def train(self, x_train, y_train, x_val=None, y_val=None, epochs=100):
        for epoch in range(epochs):
            # Training with feedback
            y_train_pred_prev = 0 # Initial value for y(k-2)
            y_pred_train = 0  # Initial value for y(k-1)

            for k in range(len(x_train)):
                # Prepare the input data with feedback
                if k == 0:
                    x_train[k][0] = y_pred_train   # Initial value for y_train(k-1)
                    x_train[k][1] = y_train_pred_prev   # Initial value for y_train(k-2)
                elif k == 1:
                    x_train[k][0] = y_pred_train   # Use y(k-1) from k=0
                    x_train[k][1] = y_train_pred_prev    # Keep the initial y(k-2)
                else:
                    x_train[k][0] = y_pred_train    # Feedback y_pred_train(k-1) as y_train(k-
1)
                    x_train[k][1] = y_train_pred_prev   # Feedback y_pred_train(k-2)

                # Forward pass and backpropagation
                y_pred_train = self.forward(x_train[k].reshape(1, -1))
                gradients_w, gradients_b = self.backward(y_train[k], y_pred_train)
                self.update_weights(gradients_w, gradients_b)

                # Store y_train_pred for feedback at next step
                y_train_pred_prev = y_pred_train[0]

            # After epoch: Evaluate on training set for metrics
            y_pred_epoch_train = []
            y_train_pred_prev = 0  # Reset feedback state for epoch evaluation
            y_pred_train = 0

            for k in range(len(x_train)):
                # Use feedback during epoch evaluation
                if k == 0:
                    x_train[k][0] = y_pred_train   # Initial value for y(k-1)
                    x_train[k][1] = y_train_pred_prev   # Initial value for y(k-2)
                elif k == 1:
                    x_train[k][0] = y_pred_train
                    x_train[k][1] = y_train_pred_prev
                else:
                    x_train[k][0] = y_pred_epoch_train[-1]   # Feedback from previous
predictions
                    x_train[k][1] = y_train_pred_prev

                y_pred_epoch_train.append(self.forward(x_train[k].reshape(1, -1))[0])
                y_train_pred_prev = y_pred_epoch_train[-1]

            # Calculate training metrics
            y_pred_epoch_train = np.array(y_pred_epoch_train)
            train_loss = self.calculate_loss(y_train, y_pred_epoch_train)
            train_acc = self.calculate_accuracy(y_train, y_pred_epoch_train)
            self.training_losses.append(train_loss)
            self.training_accuracies.append(train_acc)

            # Validation with feedback
            if x_val is not None and y_val is not None:
                y_pred_epoch_val = []
                y_val_pred_prev = 0  # Initial value for validation feedback
                y_pred_val = 0

                for k in range(len(x_val)):
                    if k == 0:
                        x_val[k][0] = y_pred_val   # Initial value for y_val(k-1)
                        x_val[k][1] = y_val_pred_prev   # Initial value for y_val(k-2)
                    elif k == 1:
                        x_val[k][0] = y_pred_val
                        x_val[k][1] = y_val_pred_prev
                    else:
                        x_val[k][0] = y_pred_epoch_val[-1]
```

```
                        x_val[k][1] = y_val_pred_prev

                    y_pred_epoch_val.append(self.forward(x_val[k].reshape(1, -1))[0])
                    y_val_pred_prev = y_pred_epoch_val[-1]

                # Calculate validation metrics
                y_pred_epoch_val = np.array(y_pred_epoch_val)
                val_loss = self.calculate_loss(y_val, y_pred_epoch_val)
                val_acc = self.calculate_accuracy(y_val, y_pred_epoch_val)
                self.validation_losses.append(val_loss)
                self.validation_accuracies.append(val_acc)

                # Print metrics
                print(f"Epoch {epoch + 1}/{epochs} - "
                      f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.4f}, "
                      f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}")
            else:
                # Print metrics without validation
                print(f"Epoch {epoch + 1}/{epochs} - "
                      f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.4f}")

    def predict(self, x):
        y_pred = []
        y_pred_prev = 0
        for k in range(len(x)):
            if k > 0:
                x[k][0] = y_pred[-1]  # Feedback from the last prediction
                x[k][1] = y_pred_prev
            current_pred = self.forward(x[k].reshape(1, -1))[0]
            y_pred.append(current_pred)
            y_pred_prev = current_pred
        return np.array(y_pred)

    def get_weights(self):
        return self.weights

    def get_biases(self):
        return self.biases

    def set_weights(self, weights):
        self.weights = weights

    def set_biases(self, biases):
        self.biases = biases

nnv = MLPVectoral(layers=[X.shape[-1], 7, 7, 7, 7, 7, 1], activations=['tanh', 'tanh', 'tanh',
'tanh', 'tanh', 'linear'], optimizer='adam', learning_rate=0.01, l2_lambda=0.0001,
weights=weights, biases=biases)

nnv.train(X_train, y_train, x_val=X_val, y_val=y_val, epochs=50)

training_losses = nnv.training_losses
validation_losses = nnv.validation_losses
training_accuracies = nnv.training_accuracies
validation_accuracies = nnv.validation_accuracies

# Create a figure with 1 row and 2 columns for loss and accuracy plots
fig, ax = plt.subplots(1, 2, figsize=(14, 6))

# Create a range of epochs for the x-axis
epochs = range(1, len(training_losses) + 1)

# Plot Loss with Logarithmic X-Axis
sns.lineplot(x=epochs, y=training_losses, ax=ax[0], label='Training Loss', color='blue')
sns.lineplot(x=epochs, y=validation_losses, ax=ax[0], label='Validation Loss', color='orange')
# ax[0].set_xscale('log')  # Set x-axis to logarithmic scale
ax[0].set_xlabel('Epochs')
ax[0].set_ylabel('Loss')
ax[0].set_title('Training and Validation Loss')
ax[0].legend()

# Plot Accuracy with Logarithmic X-Axis
sns.lineplot(x=epochs, y=training_accuracies, ax=ax[1], label='Training Accuracy',
color='blue')
sns.lineplot(x=epochs, y=validation_accuracies, ax=ax[1], label='Validation Accuracy',
color='orange')
# ax[1].set_xscale('log')  # Set x-axis to logarithmic scale
```

```
ax[1].set_xlabel('Epochs')
ax[1].set_ylabel('R² Score')
ax[1].set_title('Training and Validation Accuracy (R² Score)')
ax[1].legend()

# Adjust layout and show the plots
plt.tight_layout()
plt.show()

y_train_pred_norm = nnv.predict(X_train_norm)
y_val_pred_norm = nnv.predict(X_val_norm)
y_train_pred = scaler.inverse_transform(y_train_pred_norm)
y_val_pred = scaler.inverse_transform(y_val_pred_norm)

# Assuming y_train, y_pred_train, y_val, and y_pred_val are already defined
fig, ax = plt.subplots(1, 2, figsize=(14, 6))

# Generate time indices for training and validation
train_indices = range(len(y_train))
val_indices = range(len(y_val))

# Plot y_pred_train and y_train over time (k) with reduced alpha for predicted line
sns.lineplot(x=train_indices, y=y_train.flatten(), ax=ax[0], color='blue', label='Actual
Train', linewidth=1)
sns.lineplot(x=train_indices, y=y_pred_train.flatten(), ax=ax[0], color='orange',
label='Predicted Train', alpha=0.5, linewidth=1)
ax[0].set_xlabel('Time Index (k)')
ax[0].set_ylabel('Values')
ax[0].set_title('Training Data: Actual vs Predicted Over Time')
ax[0].legend()

# Plot y_pred_val and y_val over time (k) with reduced alpha for predicted line
sns.lineplot(x=val_indices, y=y_val.flatten(), ax=ax[1], color='blue', label='Actual Val',
linewidth=1)
sns.lineplot(x=val_indices, y=y_pred_val.flatten(), ax=ax[1], color='orange', label='Predicted
Val', alpha=0.5, linewidth=1)
ax[1].set_xlabel('Time Index (k)')
ax[1].set_ylabel('Values')
ax[1].set_title('Validation Data: Actual vs Predicted Over Time')
ax[1].legend()

# Show the updated plots
plt.tight_layout()
plt.show()

t_test = np.linspace(0, k_max, k_max)  # Time variable (consistent across inputs)

# Generate sinusoidal input for u_test_sin(k)
frequency = 5    # Frequency of the sine wave
amplitude = 10   # Amplitude of the sine wave
phase = 0        # Phase shift of the sine wave
u_test_sin = amplitude * np.sin(2 * np.pi * frequency * t_test + phase)

# Initialize y_test_sin(k) array and calculate values based on the given formula
y_test_sin = np.zeros(k_max)
for k in range(1, k_max):
    y_test_sin[k] = 1 / (1 + (y_test_sin[k-1])**2) + 0.25 * u_test_sin[k] - 0.3 *
u_test_sin[k-1]

# Shift arrays for y_test_sin(k-1), y_test_sin(k-2), u_test_sin(k-1), u_test_sin(k-2)
y_k_1_test_sin = np.zeros(k_max)
y_k_2_test_sin = np.zeros(k_max)
u_k_1_test_sin = np.zeros(k_max)
u_k_2_test_sin = np.zeros(k_max)

y_k_1_test_sin[1:] = y_test_sin[:-1]
y_k_2_test_sin[2:] = y_test_sin[:-2]
u_k_1_test_sin[1:] = u_test_sin[:-1]
u_k_2_test_sin[2:] = u_test_sin[:-2]

# Initialize MinMaxScaler to scale between -1 and 1
scaler_y = MinMaxScaler(feature_range=(-1, 1))
scaler_u = MinMaxScaler(feature_range=(-1, 1))

# Fit the scalers on training data (assuming y and u are defined)
scaler_y.fit(y.reshape(-1, 1))  # Fit on y
scaler_u.fit(u.reshape(-1, 1))  # Fit on u
```

```
# Normalize the testing data using the fitted scalers
y_test_sin_norm = scaler_y.transform(y_test_sin.reshape(-1, 1)).flatten()
y_k_1_test_sin_norm = scaler_y.transform(y_k_1_test_sin.reshape(-1, 1)).flatten()
y_k_2_test_sin_norm = scaler_y.transform(y_k_2_test_sin.reshape(-1, 1)).flatten()

u_test_sin_norm = scaler_u.transform(u_test_sin.reshape(-1, 1)).flatten()
u_k_1_test_sin_norm = scaler_u.transform(u_k_1_test_sin.reshape(-1, 1)).flatten()
u_k_2_test_sin_norm = scaler_u.transform(u_k_2_test_sin.reshape(-1, 1)).flatten()

# Prepare input matrices for the models
X_test_sin_norm = np.array([y_k_1_test_sin_norm, y_k_2_test_sin_norm, u_test_sin_norm,
u_k_1_test_sin_norm, u_k_2_test_sin_norm]).T
X_test_sin = np.array([y_k_1_test_sin, y_k_2_test_sin, u_test_sin, u_k_1_test_sin,
u_k_2_test_sin]).T

# Make predictions using the models
y_pred_test_sin_t2 = nnv.predict(X_test_sin).flatten()  # Predictions from model nnv
y_pred_test_sin_t1 = nn.predict(X_test_sin).flatten()   # Predictions from model nn

# Create a figure with 1 row and 2 columns
fig, ax = plt.subplots(1, 2, figsize=(14, 6))

# Plot the input signal (u_test_sin) in the first subplot
sns.lineplot(x=range(len(u_test_sin)), y=u_test_sin, ax=ax[0], color='blue', label='Input
Signal (u)')
ax[0].set_xlabel('Time Index (k)')
ax[0].set_ylabel('Input Value')
ax[0].set_title('Input Signal (Sinusoidal)')
ax[0].legend()

# Plot actual vs. predicted output (y_test_sin vs. y_pred_test_sin_t1) in the second subplot
sns.lineplot(x=range(len(y_test_sin)), y=y_test_sin, ax=ax[1], color='green', label='Actual
Output (y)')
sns.lineplot(x=range(len(y_test_sin)), y=y_pred_test_sin_t2, ax=ax[1], color='red',
linestyle='--', label='Predicted Output')
ax[1].set_xlabel('Time Index (k)')
ax[1].set_ylabel('Output Value')
ax[1].set_title('Actual vs. Predicted Output (Stage 2)')
ax[1].legend()

# Adjust the layout and show the plots
plt.tight_layout()
plt.show()


# Generate square wave input for u_test_square(k)
frequency = 2    # Frequency of the square wave
amplitude = 10   # Amplitude of the square wave
u_test_square = amplitude * signal.square(2 * np.pi * frequency * t_test)

# Initialize y_test_square(k) array and calculate values based on the given formula
y_test_square = np.zeros(k_max)
for k in range(1, k_max):
    y_test_square[k] = 1 / (1 + (y_test_square[k-1])**2) + 0.25 * u_test_square[k] - 0.3 *
u_test_square[k-1]

# Shift arrays for y_test_square(k-1), y_test_square(k-2), u_test_square(k-1),
u_test_square(k-2)
y_k_1_test_square = np.zeros(k_max)
y_k_2_test_square = np.zeros(k_max)
u_k_1_test_square = np.zeros(k_max)
u_k_2_test_square = np.zeros(k_max)

y_k_1_test_square[1:] = y_test_square[:-1]
y_k_2_test_square[2:] = y_test_square[:-2]
u_k_1_test_square[1:] = u_test_square[:-1]
u_k_2_test_square[2:] = u_test_square[:-2]

# Initialize MinMaxScaler to scale between -1 and 1
scaler_y = MinMaxScaler(feature_range=(-1, 1))
scaler_u = MinMaxScaler(feature_range=(-1, 1))

# Fit the scalers on training data (assuming y and u are defined)
scaler_y.fit(y.reshape(-1, 1))  # Fit on y
scaler_u.fit(u.reshape(-1, 1))  # Fit on u
```

```
# Normalize the testing data using the fitted scalers
y_test_square_norm = scaler_y.transform(y_test_square.reshape(-1, 1)).flatten()
y_k_1_test_square_norm = scaler_y.transform(y_k_1_test_square.reshape(-1, 1)).flatten()
y_k_2_test_square_norm = scaler_y.transform(y_k_2_test_square.reshape(-1, 1)).flatten()

u_test_square_norm = scaler_u.transform(u_test_square.reshape(-1, 1)).flatten()
u_k_1_test_square_norm = scaler_u.transform(u_k_1_test_square.reshape(-1, 1)).flatten()
u_k_2_test_square_norm = scaler_u.transform(u_k_2_test_square.reshape(-1, 1)).flatten()

# Prepare input matrices for the models
X_test_square_norm = np.array([y_k_1_test_square_norm, y_k_2_test_square_norm,
u_test_square_norm, u_k_1_test_square_norm, u_k_2_test_square_norm]).T
X_test_square = np.array([y_k_1_test_square, y_k_2_test_square, u_test_square,
u_k_1_test_square, u_k_2_test_square]).T

# Make predictions using the models
y_pred_test_square_t2 = nnv.predict(X_test_square).flatten()  # Predictions from model nnv
y_pred_test_square_t1 = nn.predict(X_test_square).flatten()   # Predictions from model nn

# Create a figure with 1 row and 2 columns to visualize square wave input and model output
fig, ax = plt.subplots(1, 2, figsize=(14, 6))

# Plot the input signal (u_test_square) in the first subplot
sns.lineplot(x=range(len(u_test_square)), y=u_test_square, ax=ax[0], color='blue',
label='Input Signal (Square Wave)')
ax[0].set_xlabel('Time Index (k)')
ax[0].set_ylabel('Input Value')
ax[0].set_title('Input Signal (Square Wave)')
ax[0].legend()

# Plot actual vs. predicted output (y_test_square vs. y_pred_test_square_t1) in the second
subplot
sns.lineplot(x=range(len(y_test_square)), y=y_test_square, ax=ax[1], color='green',
label='Actual Output (y)')
sns.lineplot(x=range(len(y_test_square)), y=y_pred_test_square_t2, ax=ax[1], color='red',
linestyle='--', label='Predicted Output')
ax[1].set_xlabel('Time Index (k)')
ax[1].set_ylabel('Output Value')
ax[1].set_title('Actual vs. Predicted Output (Square Wave)')
ax[1].legend()

# Adjust the layout and show the plots
plt.tight_layout()
plt.show()

# Generate triangle wave input for u_test_triangle(k)
frequency = 5    # Frequency of the triangle wave
amplitude = 10   # Amplitude of the triangle wave
u_test_triangle = amplitude * signal.sawtooth(2 * np.pi * frequency * t_test, 0.5)  # 0.5 duty
cycle for a triangle wave

# Initialize y_test_triangle(k) array and calculate values based on the given formula
y_test_triangle = np.zeros(k_max)
for k in range(1, k_max):
    y_test_triangle[k] = 1 / (1 + (y_test_triangle[k-1])**2) + 0.25 * u_test_triangle[k] - 0.3
* u_test_triangle[k-1]

# Shift arrays for y_test_triangle(k-1), y_test_triangle(k-2), u_test_triangle(k-1),
u_test_triangle(k-2)
y_k_1_test_triangle = np.zeros(k_max)
y_k_2_test_triangle = np.zeros(k_max)
u_k_1_test_triangle = np.zeros(k_max)
u_k_2_test_triangle = np.zeros(k_max)

y_k_1_test_triangle[1:] = y_test_triangle[:-1]
y_k_2_test_triangle[2:] = y_test_triangle[:-2]
u_k_1_test_triangle[1:] = u_test_triangle[:-1]
u_k_2_test_triangle[2:] = u_test_triangle[:-2]

# Initialize MinMaxScaler to scale between -1 and 1
scaler_y = MinMaxScaler(feature_range=(-1, 1))
scaler_u = MinMaxScaler(feature_range=(-1, 1))

# Fit the scalers on training data (assuming y and u are defined)
scaler_y.fit(y.reshape(-1, 1))  # Fit on y
scaler_u.fit(u.reshape(-1, 1))  # Fit on u
```

```
# Normalize the testing data using the fitted scalers
y_test_triangle_norm = scaler_y.transform(y_test_triangle.reshape(-1, 1)).flatten()
y_k_1_test_triangle_norm = scaler_y.transform(y_k_1_test_triangle.reshape(-1, 1)).flatten()
y_k_2_test_triangle_norm = scaler_y.transform(y_k_2_test_triangle.reshape(-1, 1)).flatten()

u_test_triangle_norm = scaler_u.transform(u_test_triangle.reshape(-1, 1)).flatten()
u_k_1_test_triangle_norm = scaler_u.transform(u_k_1_test_triangle.reshape(-1, 1)).flatten()
u_k_2_test_triangle_norm = scaler_u.transform(u_k_2_test_triangle.reshape(-1, 1)).flatten()

# Prepare input matrices for the models
X_test_triangle_norm = np.array([y_k_1_test_triangle_norm, y_k_2_test_triangle_norm,
u_test_triangle_norm, u_k_1_test_triangle_norm, u_k_2_test_triangle_norm]).T
X_test_triangle = np.array([y_k_1_test_triangle, y_k_2_test_triangle, u_test_triangle,
u_k_1_test_triangle, u_k_2_test_triangle]).T

# Make predictions using the models
y_pred_test_triangle_t2 = nnv.predict(X_test_triangle).flatten()  # Predictions from model nnv
y_pred_test_triangle_t1 = nn.predict(X_test_triangle).flatten()   # Predictions from model nn

# Create a figure with 1 row and 2 columns to visualize triangle wave input and model output
fig, ax = plt.subplots(1, 2, figsize=(14, 6))

# Plot the input signal (u_test_triangle) in the first subplot
sns.lineplot(x=range(len(u_test_triangle)), y=u_test_triangle, ax=ax[0], color='blue',
label='Input Signal (Triangle Wave)')
ax[0].set_xlabel('Time Index (k)')
ax[0].set_ylabel('Input Value')
ax[0].set_title('Input Signal (Triangle Wave)')
ax[0].legend()

# Plot actual vs. predicted output (y_test_triangle vs. y_pred_test_triangle_t1) in the second
subplot
sns.lineplot(x=range(len(y_test_triangle)), y=y_test_triangle, ax=ax[1], color='green',
label='Actual Output (y)')
sns.lineplot(x=range(len(y_test_triangle)), y=y_pred_test_triangle_t2, ax=ax[1], color='red',
linestyle='--', label='Predicted Output')
ax[1].set_xlabel('Time Index (k)')
ax[1].set_ylabel('Output Value')
ax[1].set_title('Actual vs. Predicted Output (Triangle Wave)')
ax[1].legend()

# Adjust the layout and show the plots
plt.tight_layout()
plt.show()
```