Neural Networks – Project

Katarina Dimić, Marko Milinković
University of Belgrade, Faculty of Electrical Engineering
2022/0362, 2022/0564

October 23, 2025

Abstract

The project encompasses the design and analysis of artificial neural networks and fuzzy controllers. In the first part, a multilayer fully connected network for data classification was developed, with hyperparameter optimization and performance evaluation. The second part covers a convolutional network for image classification, with emphasis on preprocessing and results analysis. In the third part, a fuzzy controller was designed. The results demonstrate successful application of neural networks and fuzzy logic in modeling and controlling nonlinear systems.

Contents

1 Design of Fully Connected Neural Network

1.1 Problem Description and Dataset

For the implementation of this task, the `CTG.csv` dataset (Cardiotocography dataset) was used, which contains results of cardiotocographic measurements, i.e., combined recordings of fetal heart rate (FHR) and uterine contractions in pregnant women during the third trimester of pregnancy. Cardiotocography represents a standard diagnostic method in obstetrics for assessing fetal condition, where changes in heart rate are observed in relation to uterine contractions, in order to determine whether there is a risk of fetal distress based on these parameters.

The goal of the task is to perform **automatic classification of fetal status** using a fully connected neural network based on quantitative parameters. The model should learn relationships between different physiological parameters and fetal status in order to predict the diagnostic category for new, unknown examples.

The dataset contains a total of **2126 samples**, where each sample is described by **21 numerical features (attributes)** that quantitatively represent different characteristics of fetal rhythm and contractions (e.g., beat variability, number of accelerations, number of decelerations, movement index, etc.). The target variable is indicated by the `CLASS` column, which contains three classes:

- **1 – Normal fetal status:** regular heart rate, no signs of stress;
- **2 – Suspicious status:** possible appearance of minor irregularities requiring additional monitoring;
- **3 – Pathological status:** pronounced irregularities that may indicate fetal distress.

The problem is thus defined as a **multi-class supervised classification task** (*supervised multi-class classification*), in which input quantities (attributes) and their corresponding classes (labels) are known. The task of the neural network is to approximate a nonlinear function during training that maps the attribute space to discrete fetal status classes, with maximum accuracy and generalization capability.

The figure shows a histogram of sample distribution by classes. A significant **dataset imbalance** is observed – class **1 (normal fetal status)** is predominantly represented with over 1600 samples, while classes **2 (suspicious)** and **3 (pathological)** have significantly fewer instances. Such distribution can lead to model bias toward the majority class, which is why **balanced accuracy** metric and **stratified dataset splitting** were used during training to maintain proportional representation of all classes.

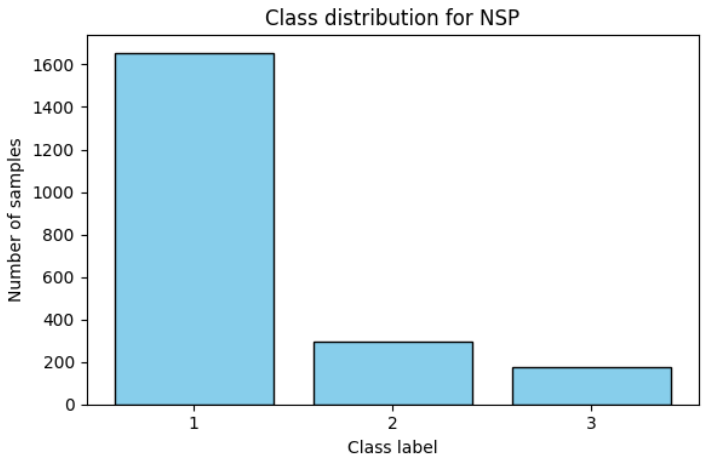

Figure 1: Histogram of sample distribution by classes in the `CTG.csv` dataset.

1.2 Data Preprocessing

Before starting neural network training, it was necessary to conduct data preprocessing to ensure reliable input data and stable model convergence during training.

First, all records with missing values were removed from the dataset. This step is important because the presence of *NaN* values can lead to incorrect weight estimation during learning, and thus to degradation of model performance. Since the `CTG.csv` dataset contains real clinical data collected through measurements, unrealistic or incomplete values may result from sensor errors or manual entry. Their removal ensures dataset homogeneity and reduces noise in the data.

Then standardization of all numerical features was performed using the StandardScaler method. Standardization transforms each attribute so that it has zero mean and unit variance, which prevents attributes with larger numerical ranges (e.g., number of decelerations) from dominating over attributes with smaller ranges (e.g., mean variation). This accelerates and stabilizes the learning process.

The dependent variable (`CLASS`) was encoded using `LabelEncoder`, to convert symbolic class labels (1, 2, 3) into numerical values suitable for model input. Although the classes are already numerically labeled, the encoder ensures consistent data type and uniform interpretation when evaluating and generating classification reports.

After processing, the data were split into **training set (80% of samples)** and **test set (20% of samples)**. The training set is used for network training and validation, while the test set serves for independent verification of the model's ability to generalize to new data. The split was performed with **stratification by classes**, which means that the same class ratio was preserved in both subsets. This procedure is necessary because the `CTG.csv` dataset is not completely balanced – normal conditions are more numerous than suspicious and pathological ones. Without stratification, the model could learn a biased distribution and favor the dominant class.

1.3 Optimization and Cross-Validation

To find the optimal neural network configuration, systematic hyperparameter optimization was conducted using **cross-validation** and grid search. The goal of this procedure was to find the parameter combination that provides the best balance between accuracy, stability, and model generalization capability.

Cross-validation was implemented through a *5-fold stratified CV* procedure, which means the entire dataset was divided into five equal parts (folds) while preserving class distribution in each part. During training, the network is trained on four data parts, while the fifth part is used for validation. The process is repeated five times, so that each subgroup serves once as a validation set. This provides a more reliable estimate of model performance, since each instance from the dataset is used for both training and validation. This approach significantly reduces the risk of overfitting and dependence on a specific data split.

Hyperparameter search was conducted using the `GridSearchCV` function, which systematically examines all possible combinations of specified hyperparameter values. The model is trained and evaluated in each combination using five-fold cross-validation, and **balanced accuracy** metric is used as the comparison criterion. This metric represents the average accuracy per class and is suitable for imbalanced datasets, like `CTG.csv`, where normal states are more represented than pathological ones.

Within the experiment, three key hyperparameters were optimized:

- **Number of neurons per layer** (`hidden_layer_sizes`) – determines network capacity, i.e., its ability to learn complex nonlinear relationships between input data. Tested configurations were (8), (16), (32), and $(16, 8)$. A smaller number of neurons can lead to insufficient model ability to learn patterns in data (*underfitting*), while too many neurons increase the risk of overfitting. The combined configuration $(16, 8)$ allows the network to capture global characteristics in the first layer, and abstract more detailed structures in the second layer.
- **Initial learning rate** (`learning_rate_init`) – controls the step size when updating weights during learning. Tested values were 10^{-2}, 10^{-3}, and 10^{-4}. Larger values accelerate convergence but can cause oscillations around the loss function minimum, while smaller values enable more stable but slower learning. The optimal value of 10^{-2} proved to be the best compromise between speed and stability.
- **Mini-batch size** (`batch_size`) – defines the number of samples processed before each weight update iteration. Tested values were 16, 32, and 64. Smaller batches enable more frequent updates and greater stochasticity in learning, which can help avoid local minima, while larger batches provide more stable gradient estimation but with greater memory consumption. In this project, the optimal balance was achieved with a batch size of 16.

Cross-validation results showed that the best combination consists of:

`hidden_layer_sizes = (16, 8)`, `learning_rate_init = 0.01`, `batch_size = 16`.

This configuration provides a high degree of accuracy and stability, with fast and reliable loss function convergence. The choice of `GridSearchCV` algorithm combined with stratified cross-validation enabled objective and reproducible comparison of different network architectures and training parameters.

1.4 Loss Function, Activation Functions, and Optimization Method

For training the multilayer perceptron (*MLP*), a **Cross-Entropy loss function** was used since this is a multi-class classification task. Cross-Entropy measures the difference between the probability distribution predicted by the model and the actual class distribution. This function is particularly suitable for classification tasks because:

- it penalizes incorrect predictions proportionally to their probability,
- it enables stable training and faster convergence compared to MSE,
- it combines well with the softmax layer which provides interpretable probabilities per class.

The activation function used in hidden layers is **ReLU** (*Rectified Linear Unit*). The ReLU function was chosen due to:

- simple and efficient implementation,
- mitigation of the vanishing gradient problem in deep networks,
- acceleration of convergence compared to classical sigmoid functions.

At the output layer, **softmax** activation is used, which provides a probability distribution over classes and is directly connected to the Cross-Entropy loss function.

For loss function optimization, **stochastic gradient descent** (*SGD*) with the **Adam** algorithm was used, which represents an adaptive variant of the SGD method. Adam uses *momentum* and adaptive learning rate per parameter, which:

- enables more stable and faster training compared to classical SGD,
- reduces the need for manual learning rate adjustment,
- accelerates convergence especially in problems with noise or nonlinearities.

In addition, an **early stopping** mechanism was used, which interrupts training if there is no improvement on the validation set. This prevents overfitting and contributes to model generalization.

1.5 Training Results and Model Evaluation

Accuracy and Loss Function Graphs

Loss function graph (Figure 2) shows how the loss function value decreases during training epochs. Already after the first epoch, a sharp drop in the loss function is observed, indicating that the model very quickly began learning relevant patterns from the data. After the initial steep drop, the curve becomes gentle and stable, oscillations are minimal and remain at a low level – indicating that **the convergence process was successfully achieved**. Such behavior suggests that the initial learning rate was properly chosen and that the model had no problems with gradient explosion or vanishing.

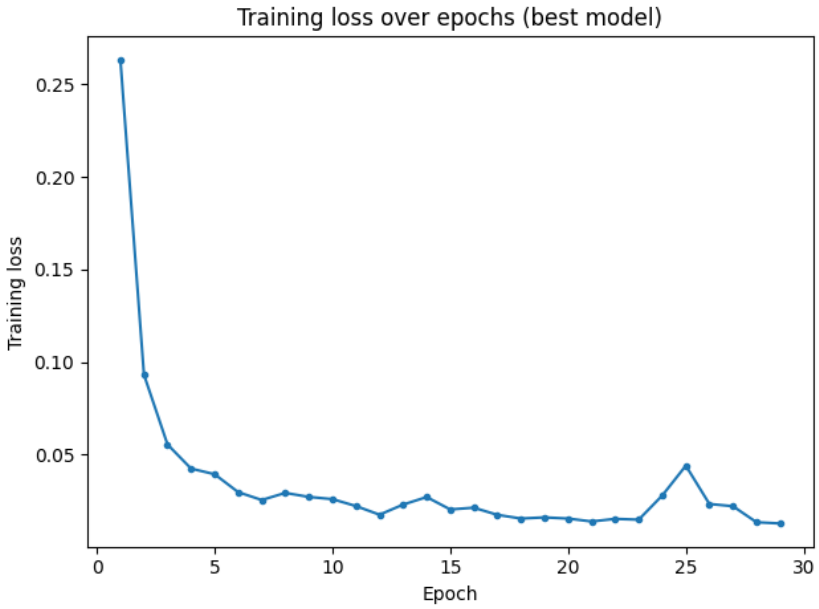

Figure 2: Loss function progression during training.

Validation accuracy graph (Figure 3) shows how model performance on the validation set evolves during epochs. Accuracy is high from the very beginning (above 0.97) and remains stable throughout the training process. Occasional minor oscillations in accuracy values are a consequence of stochasticity in learning and changes in mini-batches, but the general trend shows **stable and reliable model generalization**. The absence of a sharp accuracy drop in later epochs further confirms that overfitting did not occur.

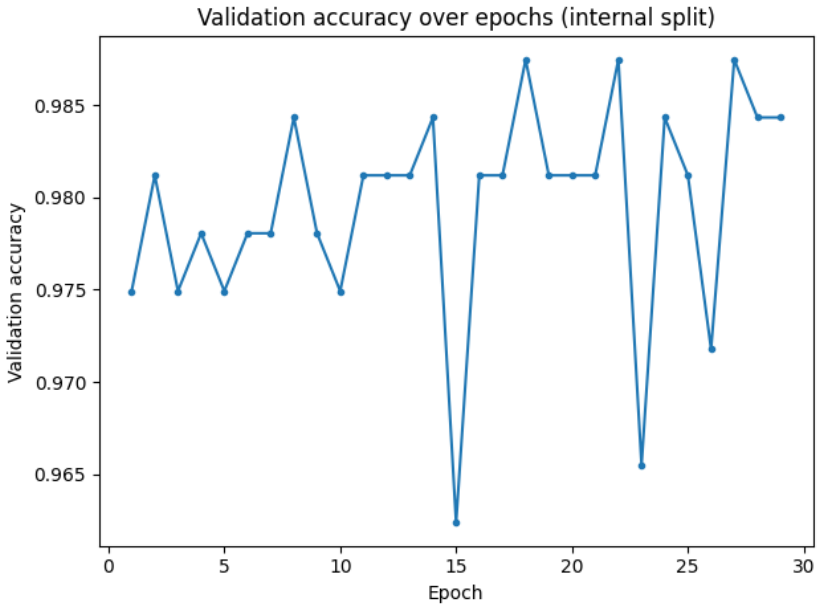

Figure 3: Validation accuracy progression by epochs.

In conclusion, the combination of a stable validation accuracy curve and rapidly decreasing loss function indicates **efficient model training**, good convergence, and satisfactory generalization capability without overfitting.

Model Performance Analysis

Model evaluation results on the training and test sets show exceptionally high classification performance. The best hyperparameter set obtained through cross-validation is:

`hidden_layer_sizes = (16, 8)`, `learning_rate_init = 0.01`, `batch_size = 16`

with an average accuracy on trained folds of **98.37%**.

Accuracy. The model achieved an overall accuracy of **99.47%** on the training set and **99.77%** on the test set. High accuracy on both sets indicates exceptional generalization capability and model stability. There is no significant difference between training and test accuracy, which suggests that overfitting did not occur.

Precision. Precision is **0.9947** on training and **0.9977** on test, meaning that false positive classifications are minimal. The model very rarely marks a sample as belonging to a class if that is not correct. Such properties are particularly important in medical applications where misclassified cases can have serious consequences.

Sensitivity (Recall). Model sensitivity of **0.9947** on training and **0.9977** on test shows that the model successfully detects most positive cases. High recall for classes 2 and 3 shows that the model recognizes less represented classes well, which is crucial for imbalanced datasets.

F1-score. F1-score is **0.9947** on training and **0.9977** on test, indicating excellent balance between precision and recall. The high value of this metric confirms model consistency across all classes and absence of significant trade-offs between accuracy and coverage.

Conclusion. High values of all key metrics (accuracy, precision, recall, and F1-score) show that the model is very reliable and stable. The choice of hyperparameters contributed to efficient learning, while high test set accuracy confirms the model's ability to generalize to new data.

```
Best hyperparameters: {'mlp__batch_size': 16, 'mlp__hidden_layer_sizes': (16, 8), 'mlp__learning_rate_init': 0.01}
CV mean accuracy (train folds): 0.9836517608733288

TRAIN metrics:
Accuracy: 0.9947 | Precision(w): 0.9947 | Recall/Sensitivity(w): 0.9947 | F1(w): 0.9947
              precision    recall  f1-score   support

           1       1.00      1.00      1.00      1323
           2       0.97      0.99      0.98       236
           3       1.00      0.99      1.00       141

    accuracy                           0.99      1700
   macro avg       0.99      0.99      0.99      1700
weighted avg       0.99      0.99      0.99      1700


TEST metrics:
Accuracy: 0.9977 | Precision(w): 0.9977 | Recall/Sensitivity(w): 0.9977 | F1(w): 0.9977
              precision    recall  f1-score   support

           1       1.00      1.00      1.00       332
           2       0.98      1.00      0.99        59
           3       1.00      1.00      1.00        35

    accuracy                           1.00       426
   macro avg       0.99      1.00      1.00       426
weighted avg       1.00      1.00      1.00       426


Hyperparameters tested:
hidden_layer_sizes: [(8,), (16,), (32,), (16, 8)]
learning_rate_init: [0.01, 0.001, 0.0001]
batch_size: [16, 32, 64]
```

Figure 4: Display of performance metric values on training and test sets for the best hyperparameter configuration.

Confusion Matrix (training set)

The confusion matrix for the training set shows that the model correctly classifies almost all samples, with minimal errors between adjacent classes. Most examples from class *0 (normal)* and *2 (pathological)* are correctly recognized, while a smaller number are incorrectly classified into similar categories (*e.g., suspicious*). This shows that the model successfully learned patterns characteristic of each class without significant bias.

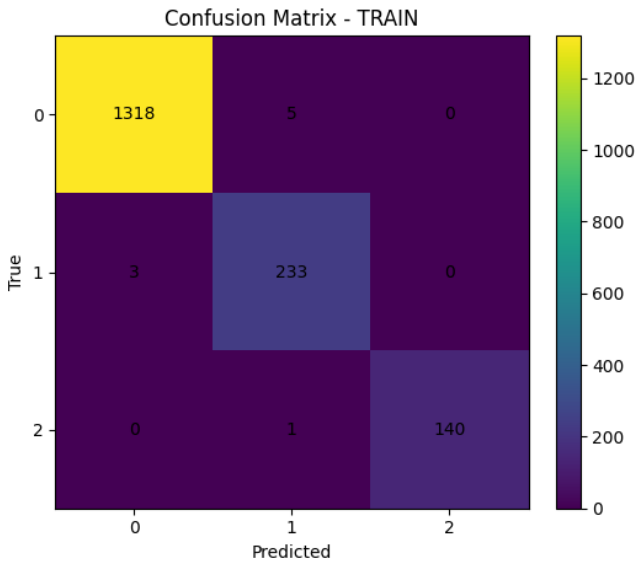

Figure 5: Confusion matrix on training set.

Confusion Matrix (test set)

On the test matrix, the model maintains excellent performance – the majority of examples are classified into the correct class (331, 59, and 35 correct classifications per class). Errors are minimal (e.g., only one misclassified example in class 1), which confirms that the model **generalizes very well** and is not overly adapted to the training set.

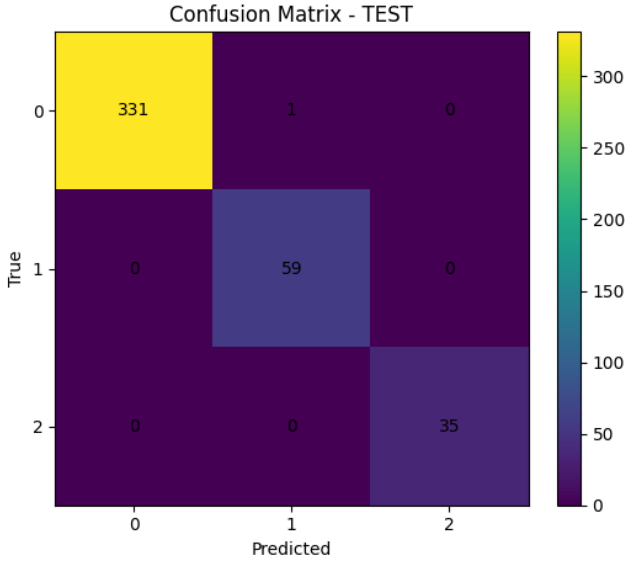

Figure 6: Confusion matrix on test set.

1.6 Conclusion

Overall, the model successfully learned relationships between input parameters and fetal diagnostic classes, achieving high values of accuracy, precision, and sensitivity metrics.

The obtained loss function values and stable convergence during training confirm that the chosen architecture is well adapted to the nature of the problem. The choice of optimized hyperparameters contributed to stable learning without pronounced oscillations and without overfitting, which is also confirmed by the consistency of performance on training and test sets. Confusion matrix analysis shows that most errors occur between similar classes (suspicious and pathological states), which is expected given the overlap of their clinical characteristics.

In conclusion, it can be said that the developed model reliably classifies fetal states based on cardiotocographic signals, representing a foundation for further improvement and integration into intelligent medical diagnostic systems.

2 Design of Fully Convolutional Neural Network

2.1 Problem Description and Dataset

Within this task, a fully convolutional neural network (CNN) was developed to solve the problem of digit classification in images from the **SVHN** (Street View House Numbers) dataset. This dataset comes from a real environment – digitized images of house numbers from streets and represents a challenging task due to the presence of noise, lighting changes, digit occlusion, and variations in position and size.

The `SVHN` set contains images with dimensions of 32×32 pixels in color (RGB format), and each image shows one digit from the set $\{0, 1, 2, \ldots, 9\}$. There are approximately **73,257** images in the training set and **26,032** in the test set, while part of the training set (˜10%) is set aside for validation during training. This defines a **multi-class classification problem with 10 classes**, where each class corresponds to one digit:

`0, 1, 2, 3, 4, 5, 6, 7, 8, 9.`

The input data are low-resolution images, which requires extraction of hierarchical features using convolutional layers. The network's task is to learn representations that are translation-invariant and resistant to changes in lighting and object position, in order to correctly classify each digit.

Figure 7 shows a histogram of sample distribution by classes. It is evident that the classes are relatively balanced, which enables fair evaluation of model performance without the need for additional data balancing.

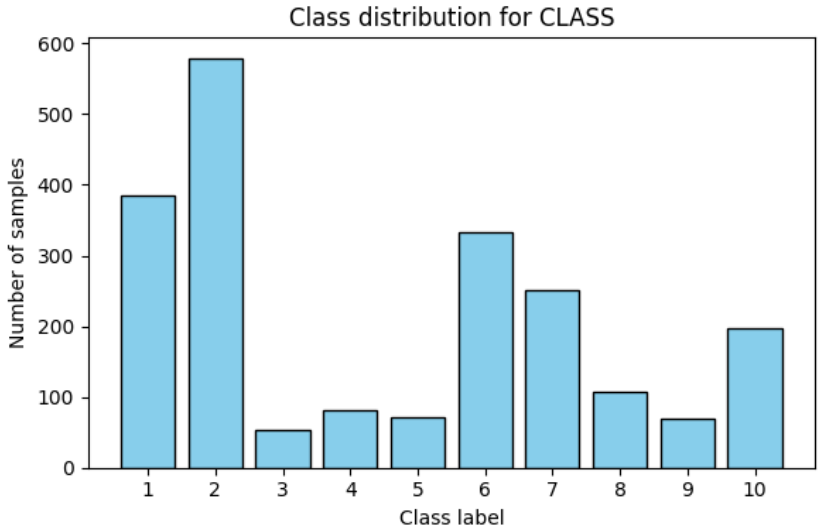

Figure 7: Distribution of number of samples by classes in the SVHN dataset.

2.2 Data Preprocessing and Augmentation

Before model training, detailed preprocessing of input images was conducted, with the goal of preparing data for efficient learning and ensuring better network generalization capability. In this project, data from the `SVHN` set were used, whose images have dimensions of 32×32 pixels in color (RGB format).

During data preparation, the following transformations were applied:

- **Random Cropping:** On the training set, augmentation in the form of random cropping of 32×32 pixel images with additional **padding of 4 pixels** was applied. This transformation simulates small translations and shifts of the object in the frame, which increases model robustness to changes in digit positions and reduces the risk of overfitting. This transformation was not applied to the validation and test sets, so that evaluation would be consistent and representative.
- **Conversion to Tensor (ToTensor):** All images were converted from `PIL.Image` format to PyTorch tensors with dimensions $(3, 32, 32)$ and values normalized to the interval $[0, 1]$. This transformation enables working with data within PyTorch's computational graph.
- **Normalization (Normalize):** After conversion, each image was normalized by channels based on mean values and standard deviations of the SVHN set:

  $$\mu = (0.4377, 0.4438, 0.4728), \quad \sigma = (0.1980, 0.2010, 0.1970)$$

  Normalization centers the distribution of pixel values and scales them so that all channels have approximately zero mean and unit variance. This accelerates convergence during training, prevents one channel from dominating over others, and ensures more stable gradient flow.

For model training, validation, and testing, PyTorch `DataLoader`s were used, with **sample shuffling (shuffle=True)** on the training set. Ten percent of training data was set aside for validation (`VAL_RATIO = 0.1`) to enable performance monitoring during learning and application of early stopping.

This procedure ensured that the network learns on diverse and properly scaled data, which contributes to its ability to generalize to real digit images that differ in position, lighting, and contrast.

2.3 Model Architecture

The developed model is a medium-depth convolutional neural network. The architecture consists of three convolutional blocks and two fully connected layers, where each block contains a convolutional layer, batch normalization, nonlinear activation, and spatial dimension reduction. Such structure allows the network to learn local edges and textures in early layers, while deeper layers abstract more complex shapes and patterns characteristic of different digits.

- **Block 1:** `Conv2d(3, 32, kernel=3, padding=1)` → `BatchNorm2d(32)` → `ReLU()` → `MaxPool2d(2)` → `Dropout(0.25)`
- **Block 2:** `Conv2d(32, 64, kernel=3, padding=1)` → `BatchNorm2d(64)` → `ReLU()` → `MaxPool2d(2)` → `Dropout(0.25)`
- **Block 3:** `Conv2d(64, 128, kernel=3, padding=1)` → `BatchNorm2d(128)` → `ReLU()` → `MaxPool2d(2)` → `Dropout(0.25)`
- **Fully connected layers:** `Linear(128*4*4, 512)` → `ReLU()` → `Dropout(0.5)` → `Linear(512, 10)` → `Softmax()`

The total number of model parameters is approximately **4.3 million**, which allows the network to efficiently learn complex visual features, while remaining compact enough for training on a standard GPU device.

Loss Function

As the loss function, **CrossEntropyLoss** was used, which combines *log-softmax* activation and negative log-likelihood. This function measures the difference between the predicted probability distribution and the actual class, and is particularly suitable for multi-class classification tasks.

This function was chosen precisely because it enables stable model training in situations when the neural network output represents a probability distribution over multiple classes, as is the case in the digit recognition problem (0–9). Unlike simpler loss functions, such as mean squared error (MSE), **CrossEntropyLoss** takes into account not only the accuracy of the predicted class, but also the prediction confidence – that is, it penalizes the network more when it is confident in a wrong answer. Thus, the model learns not only to make correct decisions, but also to be "moderately confident" in its predictions.

Additionally, Cross Entropy is characterized by good numerical stability and smooth gradients, which contributes to faster and more reliable convergence during training. In combination with the `softmax` activation in the final layer, this function allows network outputs to be directly interpreted as class membership probabilities, which facilitates analysis and evaluation of model results.

Activation Function

For all layers, the **ReLU (Rectified Linear Unit)** activation function was used, defined as $f(x) = \max(0, x)$. ReLU was chosen due to its simplicity, efficiency, and ability to eliminate the vanishing gradient problem, which is common with sigmoid and tanh functions. Additionally, the nonlinearity of ReLU enables the model to learn complex relationships between inputs and outputs without increasing computational burden.

Optimization Method

For loss function minimization, the **Adam** optimizer was used (*Adaptive Moment Estimation*) with initial learning rate $\eta = 0.001$. Adam combines the advantages of adaptive gradient scaling (as in RMSProp) and momentum (as in SGD with momentum), which achieves fast and stable convergence. During training, Adam automatically adjusts learning steps for each weight, which enables more efficient operation and better resistance to gradient noise.

Regularization

Dropout layers with dropout probability $p = 0.25$ in convolutional blocks and $p = 0.5$ in fully connected layers were applied in the model. This technique reduces the risk of overfitting by preventing neurons from depending too much on each other during training, thereby improving the network's generalization capabilities.

2.4 Training Results

Below are analyzed key graphs and confusion matrices obtained during CNN model training and evaluation on the SVHN set.

Accuracy change by epochs - Figure 8 shows accuracy curves for training and validation sets during 10 epochs. Training set accuracy grows from ∼0.64 to ∼0.91, while validation accuracy grows from ∼0.86 to ∼0.93–0.94. The consistently higher validation curve compared to training is expected due to applied regularization techniques (*dropout* and *batch normalization*) and augmentation. The close distance of the curves and absence of divergence indicate no signs of overfitting and that the model generalizes well.

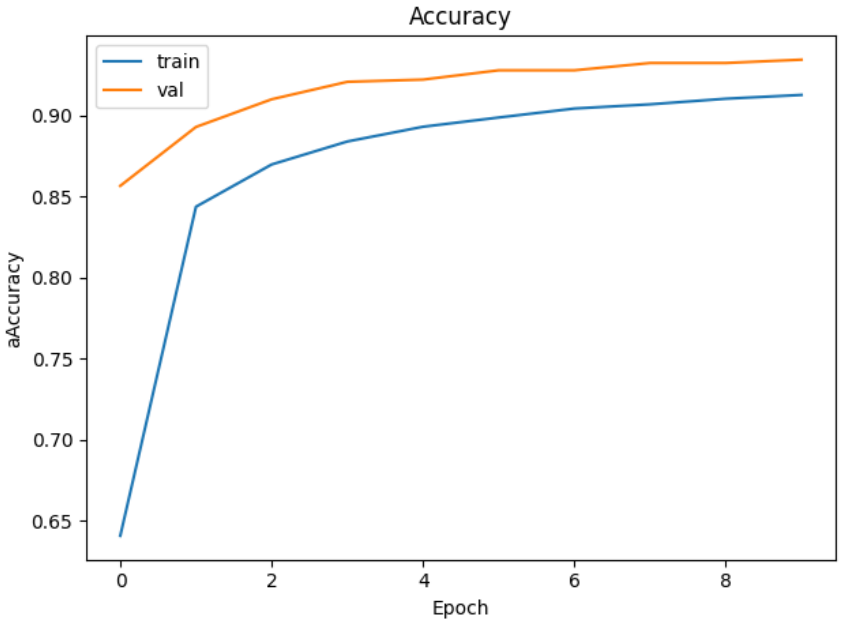

Figure 8: Accuracy (train/val) during epochs.

Loss change by epochs - Figure 9 shows monotonic decrease of loss function on both sets: training loss drops from ∼1.1 to ∼0.28, and validation from ∼0.46 to ∼0.22. Lower validation loss than training loss is consistent with previous explanation: dropout and augmentation make the task harder during learning, so the "noisy" training-loss is higher, while validation measures performance in deterministic mode (dropout disabled). Stabilization around epochs 8–10 suggests that the learning rate is appropriate and the model has reached a plateau.

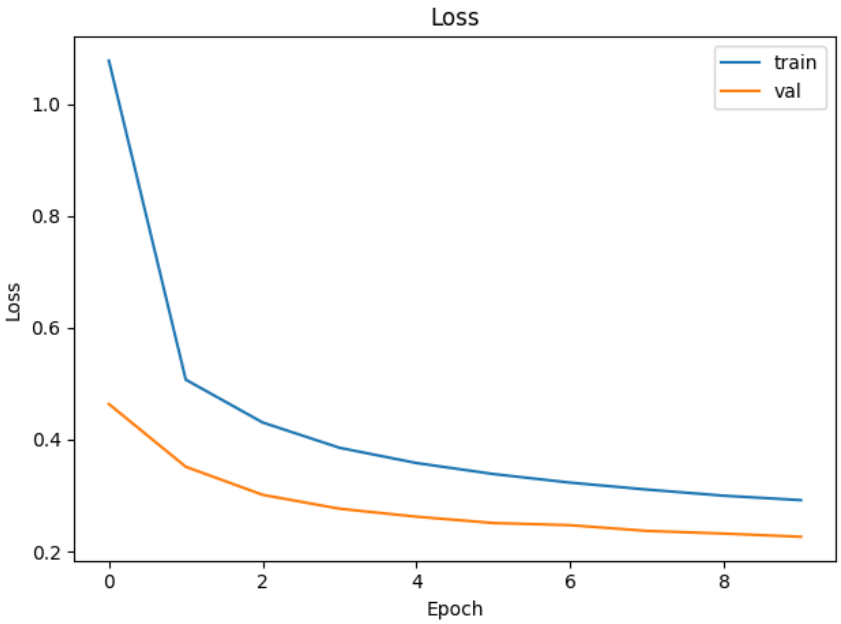

Figure 9: Loss (train/val) during epochs.

Confusion matrix for training set - In Figure 10, the confusion matrix is strongly diagonal, which means that most samples of each class are correctly classified. A small amount of off-diagonal elements (if present in a more granular view) indicates occasional errors among visually similar digits, e.g., *3 vs. 8*, *5 vs. 6*, *4 vs. 9*, which is typical for SVHN due to blurring, oblique angles, and partial occlusions. Such training results are expected after convergence and confirm that the model capacity matches the task difficulty.

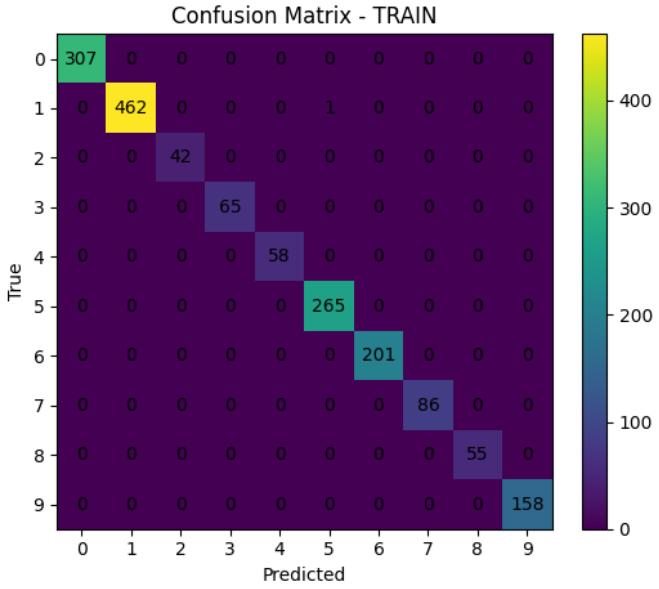

Figure 10: Confusion matrix on training set (10 classes: digits 0–9).

Confusion matrix for test set - In Figure 11, the diagonal still dominantly prevails, which confirms good generalization to unseen data. Compared to training, off-diagonal elements are more pronounced precisely for pairs of similar digits: *3–8* (closed loops and similar structure), *5–6* (similar shapes at different angles), *1–7* (slender vertical + horizontal line), *4–9* (rectangular segment versus closed loop). These errors correlate with examples in Figure 13 and often occur when digits are blurred, poorly lit, or partially occluded.

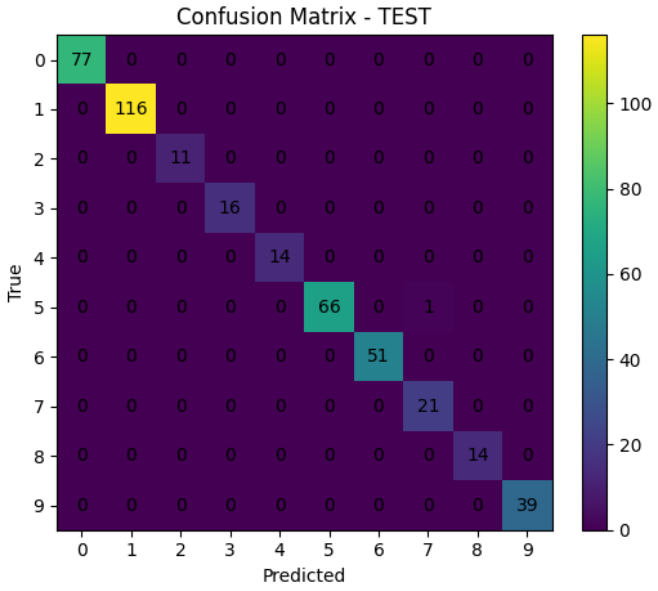

Figure 11: Confusion matrix on test set. Dominant diagonal and expected confusions among visually similar digits.

Examples of correct and incorrect classifications - Figure 12 shows samples that the model classifies correctly even under difficult conditions: slight rotations, background noise, variable lighting. This confirms that convolutional filters capture robust, translation-invariant features. In contrast, Figure 13 contains typical errors: blurring, sharp angles, partial occlusions, as well as "non-canonical" handwritten digit forms; in these cases, the model often changes the class to a visually similar prototype (e.g., 8→3, 6→5).

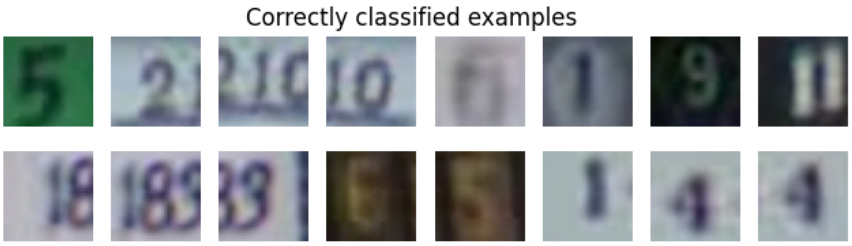

Figure 12: Examples of correctly classified images: the model is robust to slight rotations, noise, and lighting variations.

Figure 13: Examples of misclassified images: errors occur with blurred, partially occluded, and atypical digits.

Quantitative metrics - Based on curves from Figure 8, overall validation accuracy reaches ∼0.93–0.94, while precision, sensitivity, and F1-score per class are high and balanced (diagonally dominant confusion matrix). Due to moderate class imbalance, it is recommended to report *macro*-F1 in addition to overall accuracy; in our case, the difference between *macro* and *micro* F1 is small, which means that most classes achieve similar performance levels. The largest deviations are expected for classes that are either less represented or visually similar to their "confusion pairs".

Conclusion: the graphs show fast and stable learning without overfitting, while confusion matrices confirm that the model is reliable on most classes; remaining errors are primarily due to image quality and morphological similarity of digits.

2.5 Conclusion

The obtained results confirm that the convolutional architecture successfully learns translation-invariant filters and feature hierarchies relevant to the classification task. Stable validation curves and compact confusion matrix indicate good generalization. Remaining errors are mainly due to visual class similarity and signal quality variations (blurring, lighting).

3 Design of Fuzzy Controller

Our team opted for system design using an intuitive approach.

3.1 Part 1 – Definition of Fuzzy Variables and Rules

Figure 14 shows the block diagram of the implemented fuzzy controller. The diagram encompasses the process of fuzzification of input quantities, rule activation, defuzzification, and feedback to the system. Input variables pass through the fuzzy inference block, where a control signal is generated based on a set of defined rules. Then, through the dynamic system block, a response is obtained, which returns to the controller via feedback.

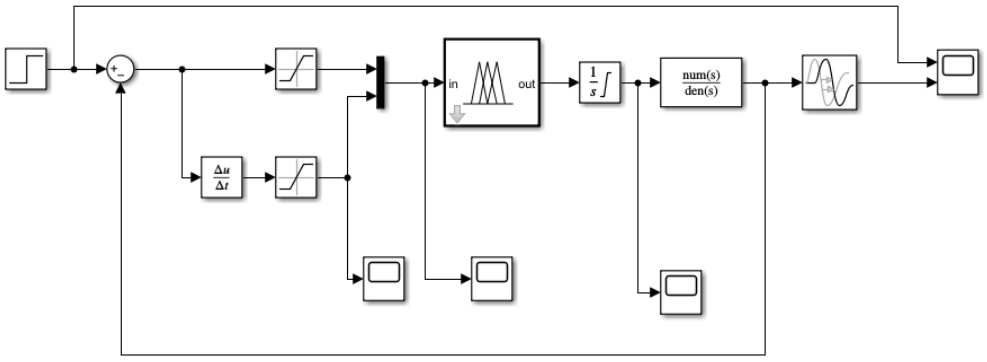

Figure 14: Block diagram of fuzzy controller

Fuzzy system – parameter overview. Figure 15 shows the basic configuration of a Mamdani-type fuzzy system: two input variables (*error* and *de*), one output (u), and 15 rules. This combination enables smooth output behavior and stable control.

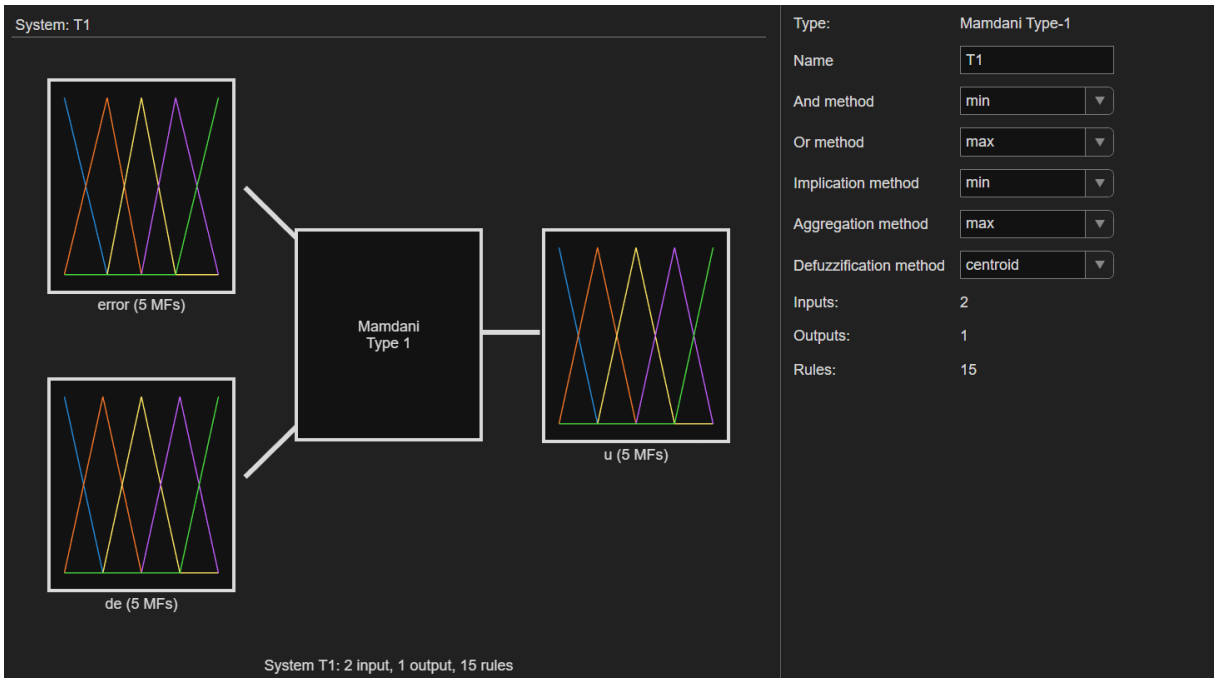

Figure 15: Mamdani-type fuzzy system configuration.

Membership function of input variable `error`. Figure 16 shows the membership functions for the `error` variable, defined on the interval $[-4.5, 4.5]$. Such division enables precise controller response to the deviation of the controlled variable from the desired value.

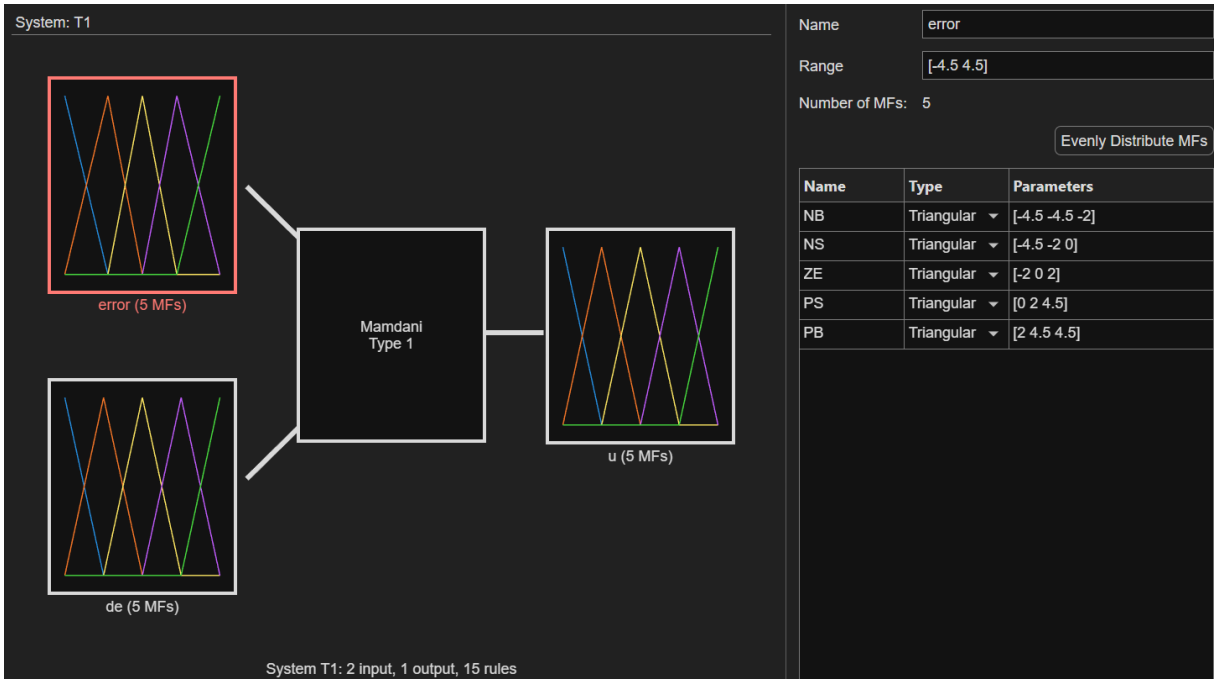

Figure 16: Membership functions for variable `error`.

Membership function of input variable `de`. Figure 17 shows the membership functions for the `de` variable (error rate of change), defined on the interval $[-3, 3]$. Including this variable enables the controller to have an anticipatory response and contributes to more stable system behavior.

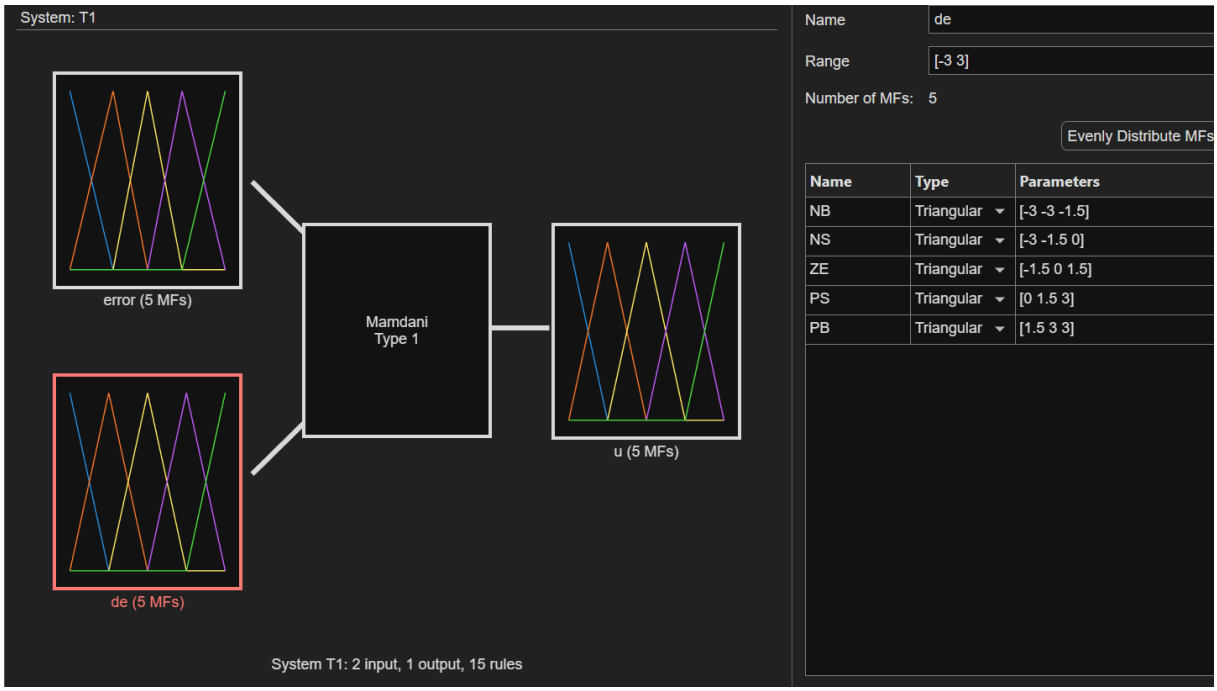

Figure 17: Membership functions for variable `de`.

Rule base. Figure 18 shows the complete set of 15 IF-THEN rules of the fuzzy controller. Such rules enable gradual and stable adjustment of the controller output according to the system state.

	Rule
1	If error is NB and de is NB then u is NB
2	If error is NB and de is ZE then u is NB
3	If error is NB and de is PB then u is NS
4	If error is NS and de is NB then u is NB
5	If error is NS and de is ZE then u is NS
6	If error is NS and de is PB then u is ZE
7	If error is ZE and de is NB then u is NS
8	If error is ZE and de is ZE then u is ZE
9	If error is ZE and de is PB then u is PS
10	If error is PS and de is NB then u is ZE
11	If error is PS and de is ZE then u is PS
12	If error is PS and de is PB then u is PB
13	If error is PB and de is NB then u is NS
14	If error is PB and de is ZE then u is PB
15	If error is PB and de is PB then u is PB

Figure 18: Set of 15 IF-THEN rules of the fuzzy controller.

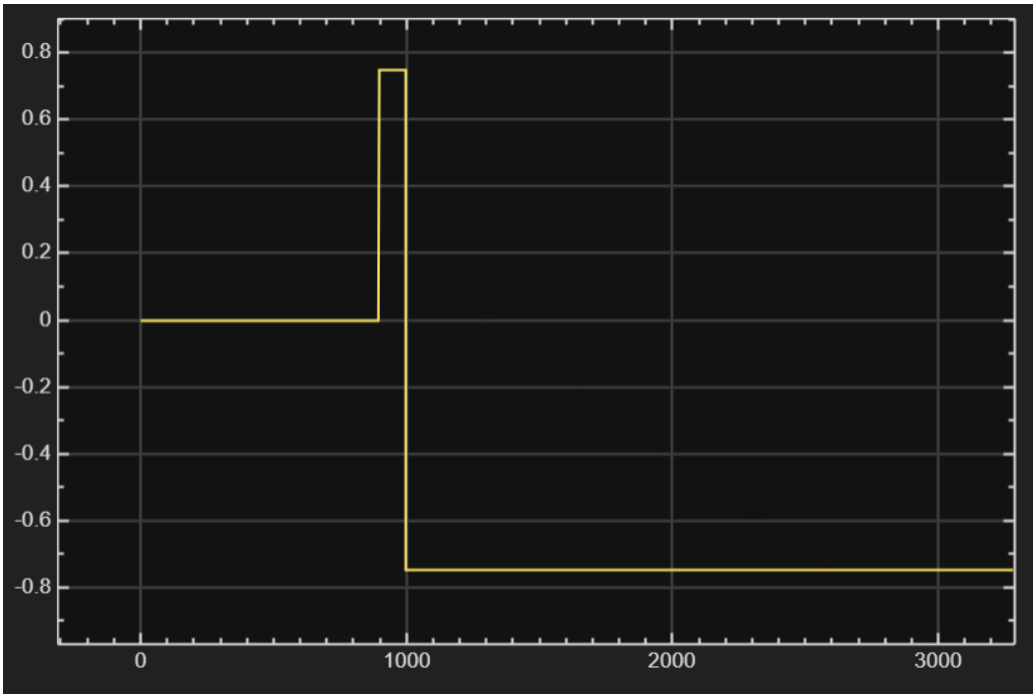

Figure 19: Step-type input signal – testing controller response to reference value change.

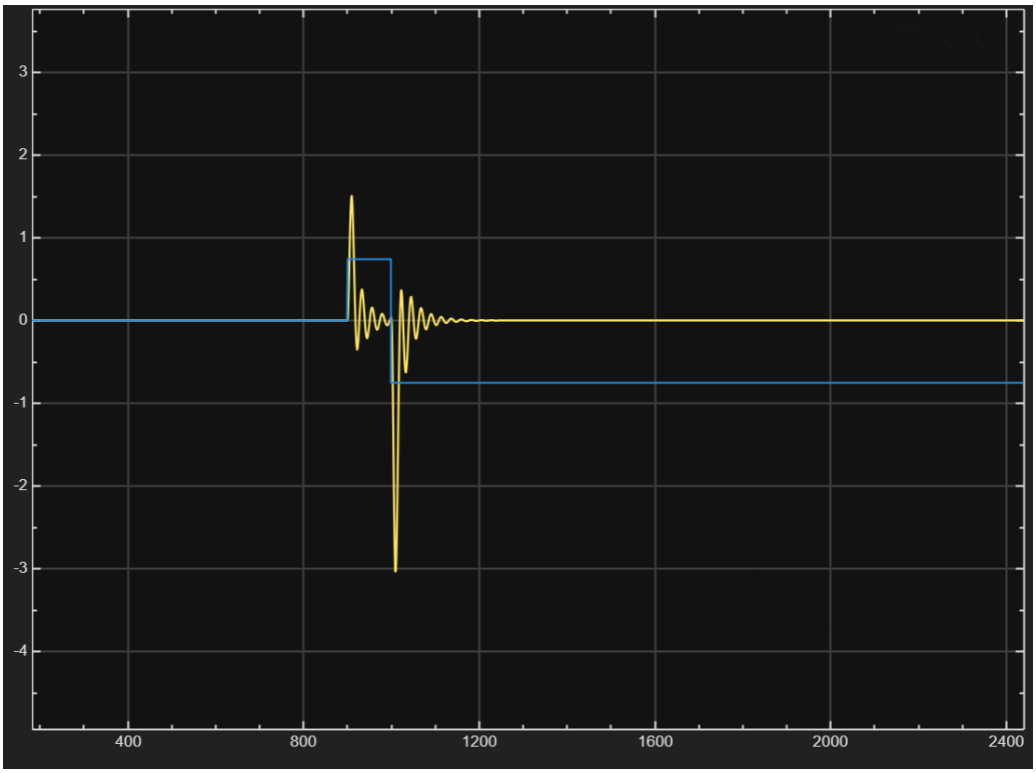

Figure 20: System response (yellow line) in relation to reference signal (blue line). It is visible that the controller successfully tracks the reference with present initial overshoot.

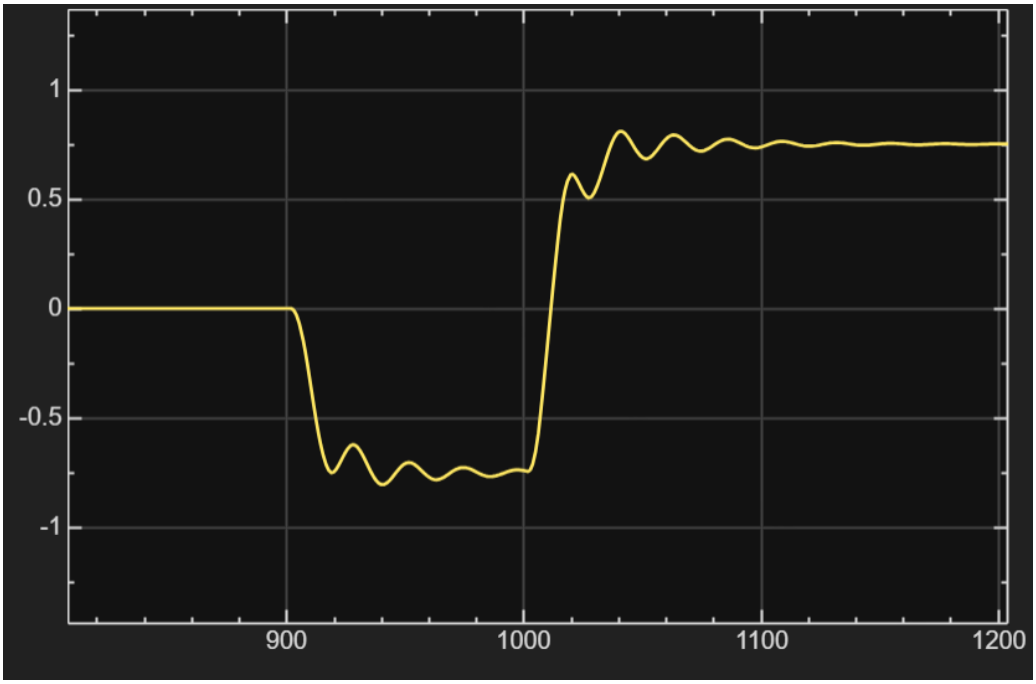

Figure 21: Fuzzy controller response to random input signal – the system remains stable and the response is smooth.

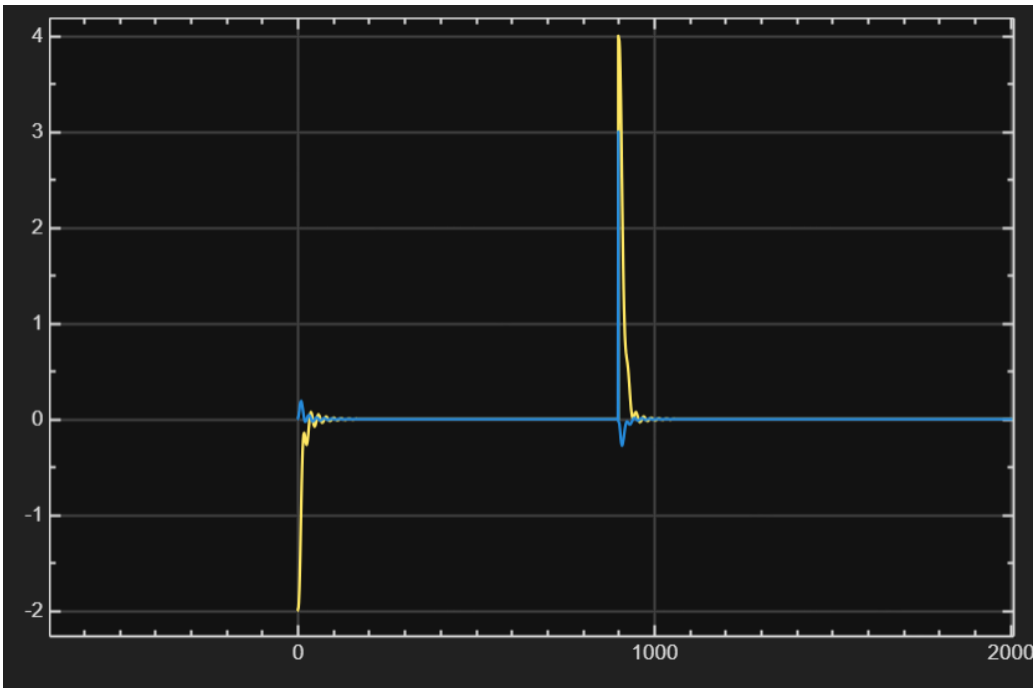

Figure 22: Additional test of input-output behavior of fuzzy system during reference signal change.

3.2 Part 2 – Extended Fuzzy Controller Block Diagram

Figure 23 shows the extended block diagram of the fuzzy controller. Compared to the previous version (Part 1), an integrator has been added to this structure that enables accumulation of the fuzzy controller output signal. The input signal (reference) is compared with the current system output, and the difference (error) and its change are passed to the fuzzy system which generates the control signal.

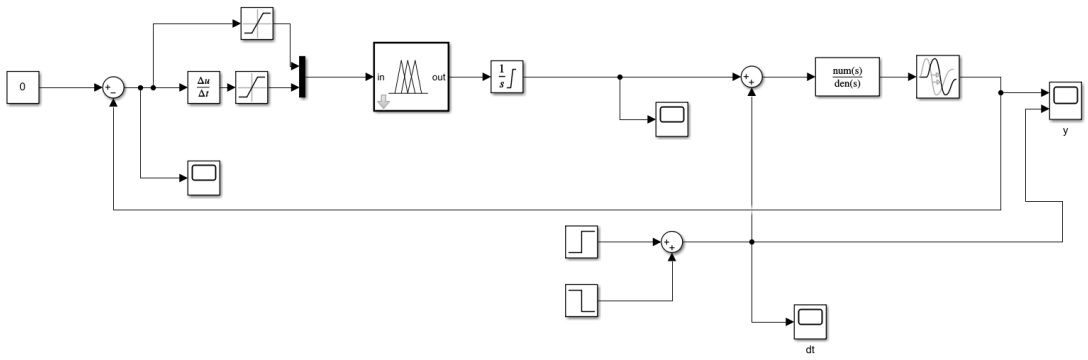

Figure 23: Extended fuzzy controller block diagram with integrator and additional input signals.

Brief block analysis.

- **Input signal (reference)** – defines the desired system state and is used to calculate error.
- **Fuzzy controller** – uses error and its change to generate appropriate control action.
- **Integrator** – enables retention of accumulated control action and improves reference tracking.
- **Transfer function block** – model of controlled process in Laplace domain.
- **Feedback** – returns output to controller and closes the system loop.
- **Test inputs and parameters (α, dt)** – enable simulation of different operating conditions.

Such fuzzy controller structure provides more robust response and better control compared to the simpler version from the previous chapter. Thanks to additional inputs, it is possible to analyze system dynamics and response to different excitations in detail.

Figure 24 shows the configuration of a Mamdani-type fuzzy system used in this part of the experiment. The system has two input variables (*error* and *de*) and one output variable (u). **Min** is used as the *AND* operator, **max** for *OR* and aggregation, while **centroid** method is applied for defuzzification. The total number of rules is 15, which is typical for a symmetric distribution of five membership functions per input.

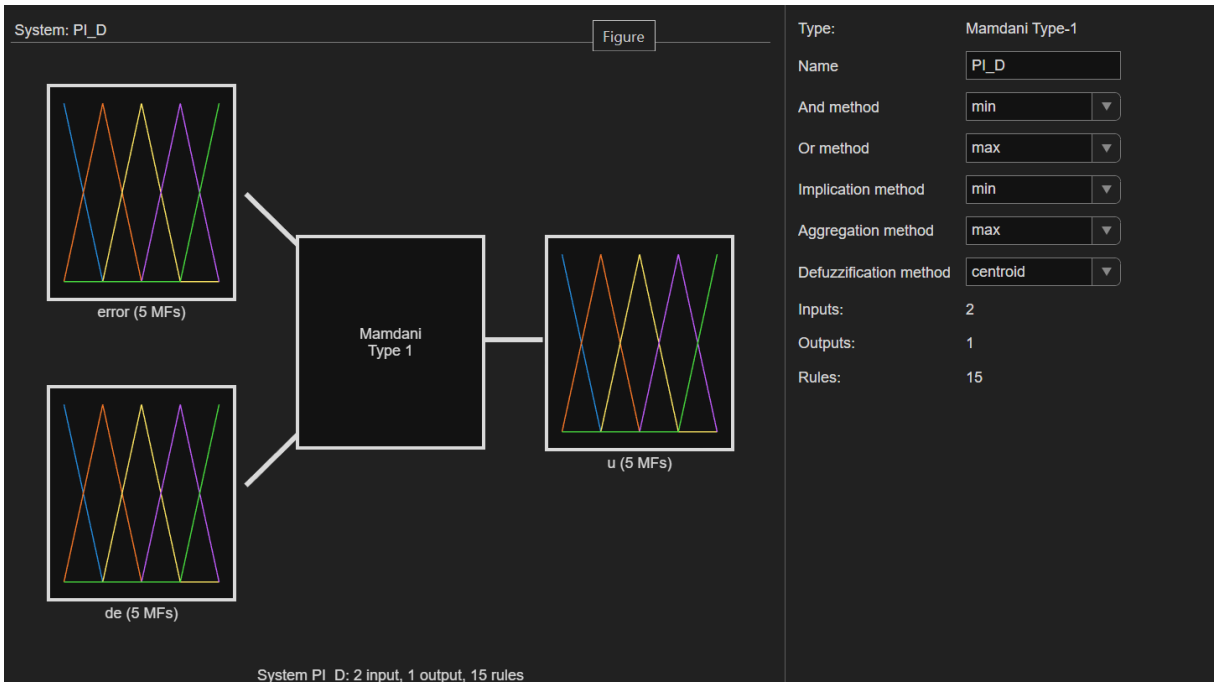

Figure 24: Mamdani fuzzy system with two input and one output variables.

Membership function of input variable `error`. Figure 25 shows the membership functions for the `error` variable, defined in the interval $[-4, 4]$. As in previous tasks, five triangular MFs are used (*NB, NS, ZE, PS, PB*). This distribution enables high resolution in the zone around zero (where controller sensitivity is greatest) and coarser response for extreme error values.

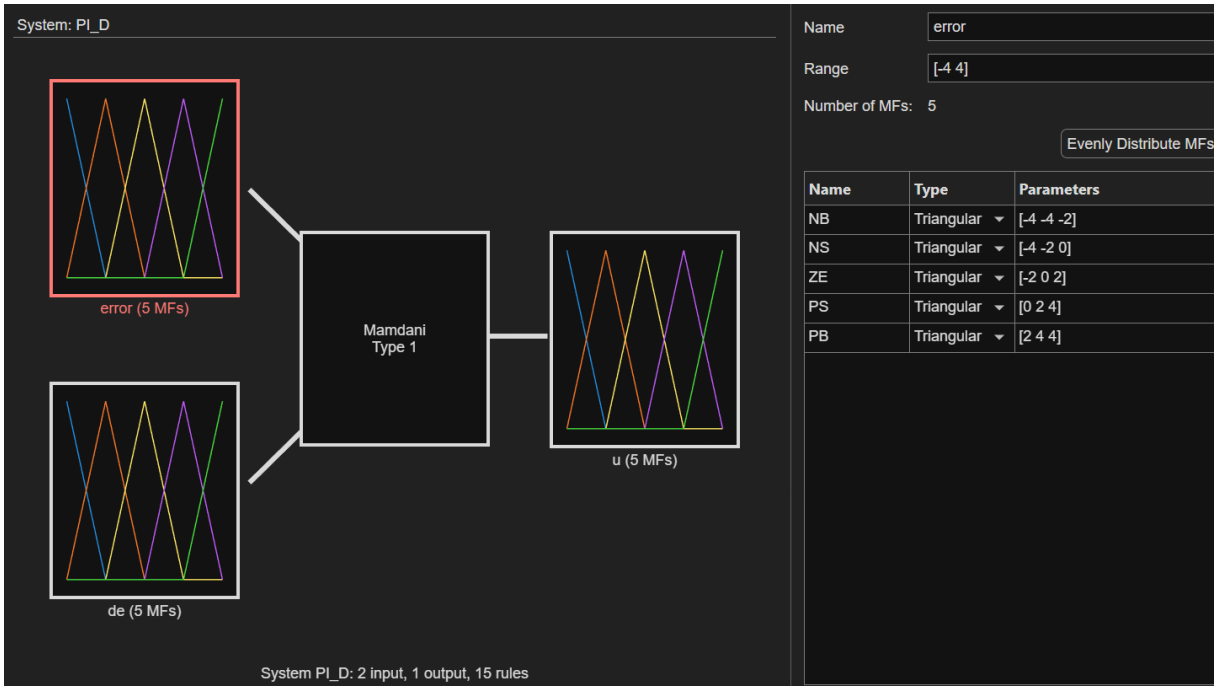

Figure 25: Membership functions for input variable `error`.

Membership function of input variable `de`. Figure 26 shows the membership functions for the `de` variable (error derivative), defined on the interval $[-3, 3]$ with the same five triangular MFs. This variable provides the **derivative component** of the fuzzy controller – the controller responds to the error rate of change, which significantly improves system dynamic characteristics.

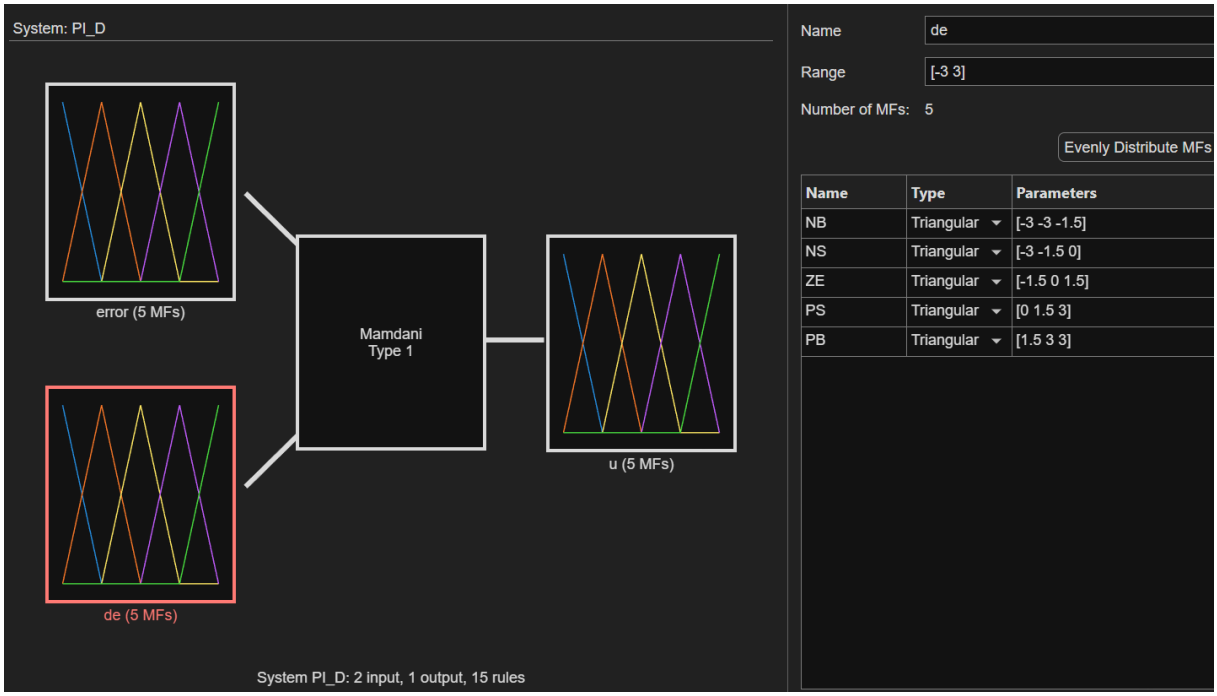

Figure 26: Membership functions for input variable `de`.

Fuzzy controller response analysis. The following figures show the system response implemented according to the block diagram from the previous chapter.

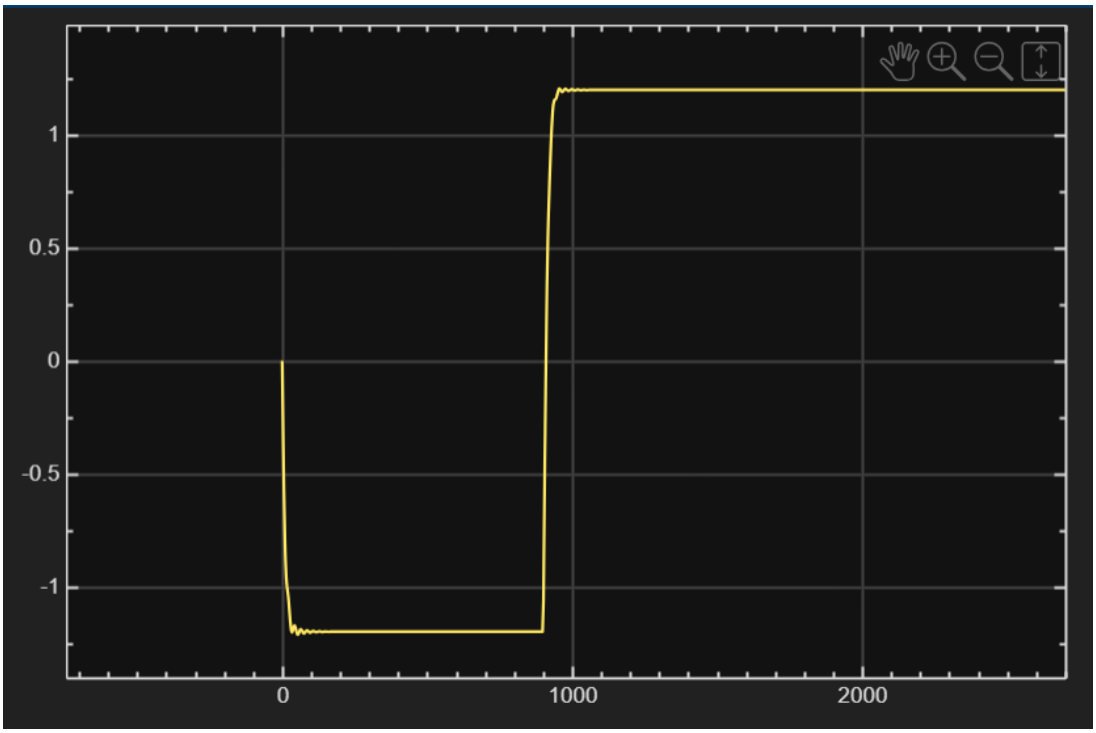

Figure 27: Integrator output $u(t)$ to the fuzzy system.

Integrator output. Figure 27 shows the change of control signal $u(t)$ generated by the fuzzy controller. It can be observed that the integrator smoothly accumulates the increment Δu coming from the fuzzy inference system. Initially, the control signal quickly responds to the negative step, and then, after the excitation sign change, adapts to the new reference without pronounced delay.

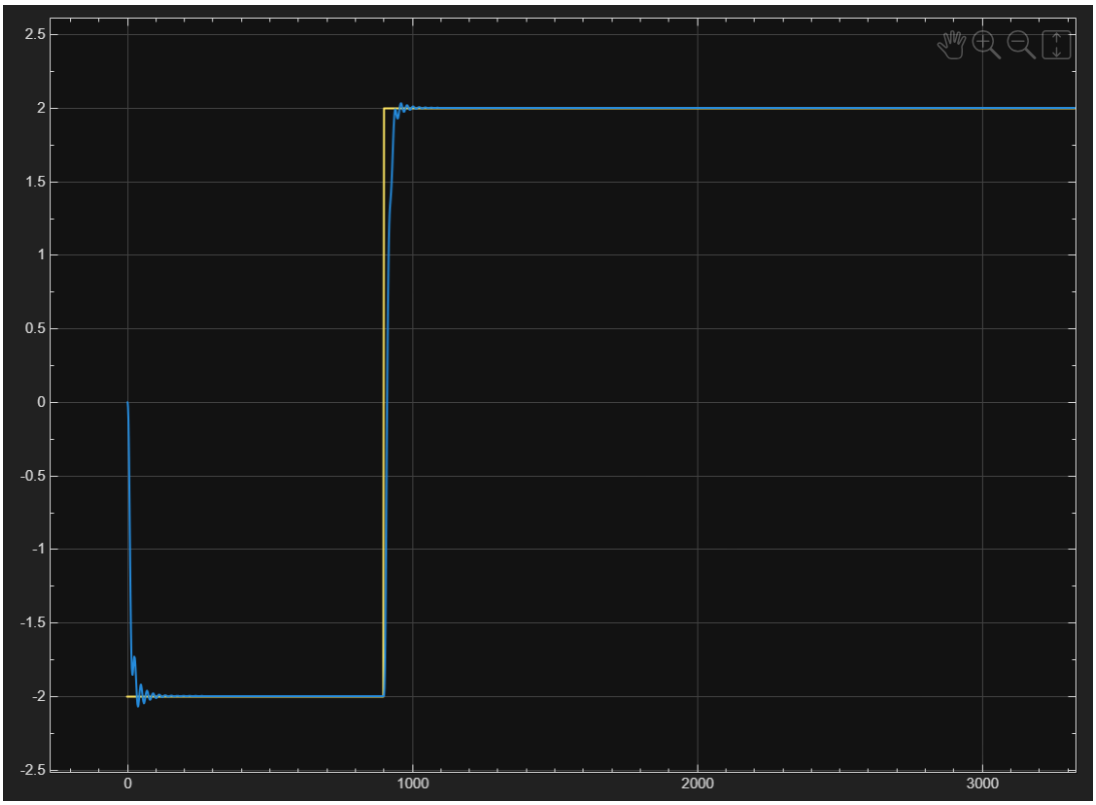

Figure 28: System response (yellow) to step input (blue).

System response to step excitation. Figure 28 shows the system output (yellow curve) compared to the reference input (blue curve). The controller shows fast response and system stabilization with minimal overshoot and short-term oscillations in transient regime.

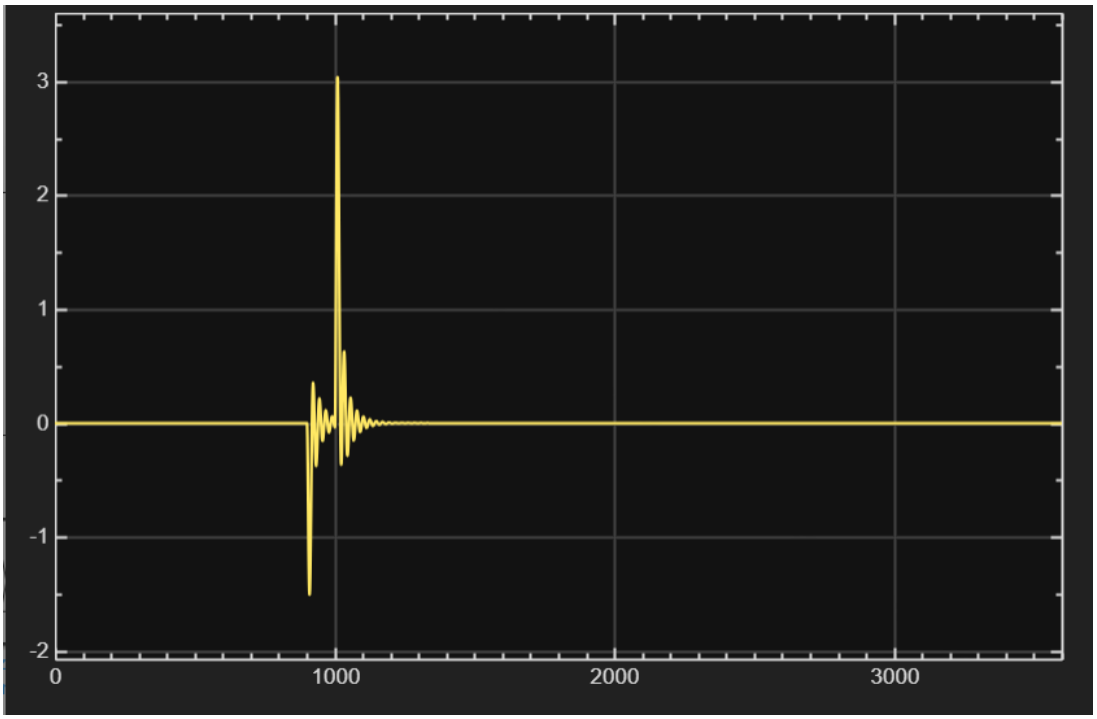

Figure 29: System response to random input signal.

Response to random excitations. Figure 29 shows the system response when a random signal is used as input. Despite sudden input changes, the system remains stable, and the output tracks the input with minimal delay and without significant overshoot. This shows that the fuzzy controller possesses sufficient robustness and flexibility.

Conclusion. Based on the displayed graphs, it can be concluded:

- provides fast and stable response without significant overshoot,
- eliminates steady-state error thanks to the integral component,
- remains stable even with random input changes.