|  |  |  |
| --- | --- | --- |
| **پرسش ۱** | **نام و نام خانوادگی** | سیدرضا مسلمی |
| **شماره دانشجویی** | 810103326 |
| **پرسش ۲** | **نام و نام خانوادگی** | بی‌بی رقیه |
| **شماره دانشجویی** | 810102053 |
|  | **مهلت ارسال پاسخ** | **15 آبان1403** |

|  |  |  |
| --- | --- | --- |
|  | **به نام خدا**  **دانشگاه تهران**  **دانشکده‌ مهندسی برق و کامپیوتر** |  |
| **درس شبکه‌های عصبی و یادگیری عمیق**  **تمرین اول** | | |

**فهرست**

[**قوانین** 1](#_Toc182004439)

[**پرسش 1**. **تحلیل و طراحی شبکه‌های عصبی چند لایه (MLP)** 1](#_Toc182004440)

[۱-۱. طراحی MLP 1](#_Toc182004441)

[2-۱. آموزش دو مدل متفاوت 2](#_Toc182004442)

[۳-۱. الگوریتم بازگشت به عقب 3](#_Toc182004443)

[۴-۱. بررسی هایپرپارامترهای متفاوت 5](#_Toc182004444)

[**پرسش ۲** **– آموزش و ارزیابی یک شبکه عصبی ساده** 15](#_Toc182004446)

[۱-۲. **آموزش یک شبکه عصبی ساده** 15](#_Toc182004447)

[۱-۲. **آموزش یک شبکه عصبی ساده** 16](#_Toc182004448)

[**پرسش ۳** **–** Madaline 21](#_Toc182004449)

[۱-۳. **MRI and MRII** 21](#_Toc182004450)

[2-۳.نمودار پراکندگی دادهها 24](#_Toc182004451)

[**MRI Implementation** 26](#_Toc182004452)

[Observations 37](#_Toc182004453)

[**پرسش ۴** **– MLP** 49](#_Toc182004454)

[۱-۴.نمایش تعداد ستون 49](#_Toc182004455)

[2-۴.ماتریس همبستگی 50](#_Toc182004456)

[Interpretation of Correlation Values 51](#_Toc182004457)

[Heatmap Features 52](#_Toc182004458)

[Practical Implications for Model Building 52](#_Toc182004459)

[3-۴.رسم نمودار 52](#_Toc182004460)

[Key Observations: 53](#_Toc182004461)

[Elements of the Plot 54](#_Toc182004462)

[Key Observations 54](#_Toc182004463)

[Insights 54](#_Toc182004464)

[Observations 55](#_Toc182004465)

[4-۴.پیش پردازش داده 56](#_Toc182004466)

[5-۴.پیاده سازی مدل (MLP) 60](#_Toc182004467)

[First Plot: Training and Validation Loss (1 Hidden Layer) 65](#_Toc182004468)

[Second Plot: Training and Validation Loss (2 Hidden Layers) 65](#_Toc182004469)

[Comparison and Interpretation 66](#_Toc182004470)

[6-۴.آموزش مدل 66](#_Toc182004471)

[7-۴.حلیل نتایج 68](#_Toc182004472)

# **قوانین**

قبل از پاسخ دادن به پرسش‌ها،‌ موارد زیر را با دقت مطالعه نمایید:

* از پاسخ‌های خود یک گزارش در قالبی که در صفحه‌ی درس در سامانه‌ی Elearn با نام ***REPORTS\_TEMPLATE.docx*** قرار داده شده تهیه نمایید.
* پیشنهاد می‌شود تمرین‌ها را در قالب گروه‌های دو نفره انجام دهید. (بیش از دو نفر مجاز نیست و تحویل تک نفره نیز نمره‌ی اضافی ندارد) توجه نمایید الزامی در یکسان ماندن اعضای گروه تا انتهای ترم وجود ندارد. (یعنی، می‌توانید تمرین اول را با شخص A و تمرین‌ دوم را با شخص B و ... انجام دهید)
* **کیفیت گزارش شما در فرآيند تصحيح از اهميت ويژه­اي برخوردار است**؛ بنابراین، لطفا تمامی نکات و فرض­هایی را كه در پیاده­سازی­ها و محاسبات خود در نظر مي­گيريد در گزارش ذکر کنید.
* در گزارش خود مطابق با آنچه در قالب نمونه قرار داده شده، برای شکل‌ها زیرنویس و برای جدول‌ها بالانویس در نظر بگیرید.
* الزامی به ارائه توضیح جزئیات کد در گزارش نیست، اما باید نتایج بدست آمده از آن را گزارش و تحلیل کنید.
* **تحلیل نتایج الزامی می‌باشد، حتی اگر در صورت پرسش اشاره‌ای به آن نشده باشد.**
* **دستیاران آموزشی ملزم به اجرا کردن کدهای شما نیستند**؛ بنابراین، هرگونه نتیجه و یا تحلیلی که در صورت پرسش از شما خواسته شده را به طور واضح و کامل در گزارش بیاورید. در صورت عدم رعایت این مورد، بدیهی است که از نمره تمرین کسر می­شود.
* **کدها حتما باید در قالب نوت‌بوک با پسوند .ipynb تهیه شوند، در پایان کار، تمامی کد اجرا شود و خروجی هر سلول حتما در این فایل ارسالی شما ذخیره شده باشد.** بنابراین برای مثال اگر خروجی سلولی یک نمودار است که در گزارش آورده‌اید، این نمودار باید هم در گزارش هم در نوت‌بوک کد‌ها وجود داشته باشد.
* **در صورت مشاهده‌ی تقلب امتیاز تمامی افراد شرکت­کننده در آن، 100- لحاظ می­شود.**
* تنها زبان برنامه نویسی مجاز **Python** است.
* **استفاده از کدهای آماده برای تمرین­ها به­ هیچ ­وجه مجاز نیست. در صورتی که دو گروه از یک منبع مشترک استفاده کنند و کدهای مشابه تحویل دهند، تقلب محسوب می‌شود.**
* نحوه محاسبه­ تاخیر به این شکل است: پس از پایان رسیدن مهلت ارسال گزارش، حداکثر تا یک هفته امکان ارسال با تاخیر وجود دارد، پس از این یک هفته نمره آن تکلیف برای شما صفر خواهد شد.
  + سه روز اول: بدون جریمه
  + روز چهارم: ۵ درصد
  + روز پنجم: ۱۰ درصد
  + روز ششم: ۱۵ درصد
  + روز هفتم: ۲۰ درصد
* حداکثر نمره‌ای که برای هر سوال می‌توان اخد کرد ۱۰۰ بوده و اگر مجموع بارم یک **سوال** بیشتر از ۱۰۰ باشد، در صورت اخد نمره بیشتر از ۱۰۰، اعمال نخواهد شد.
  + برای مثال: اگر نمره اخذ شده از سوال ۱ برابر ۱۰۵ و نمره سوال ۲ برابر ۹۵ باشد، نمره نهایی تمرین ۹۷.۵ خواهد بود و نه ۱۰۰.
* لطفا گزارش، کدها و سایر ضمایم را به در یک پوشه با نام زیر قرار داده و آن را فشرده سازید، سپس در سامانه‌ی Elearn بارگذاری نمایید:

HW[Number] \_[Lastname]\_[StudentNumber]\_[Lastname]\_[StudentNumber].zip

(مثال: HW1\_Ahmadi\_810199101\_Bagheri\_810199102.zip)

* برای گروه‌های دو نفره، بارگذاری تمرین از جانب یکی از اعضا کافی است ولی پیشنهاد می‌شود هر دو نفر بارگذاری نمایند.

# **پرسش 1**. **تحلیل و طراحی شبکه‌های عصبی چند لایه (MLP)**

Codes and more details for this question has been provided in HW01\_NNDL\_Moslemi Notbook.

۱-۱. طراحی MLP

Base on the results of confusion matrix, the most getting mistaken classes are T-shirt and Shirt classes:

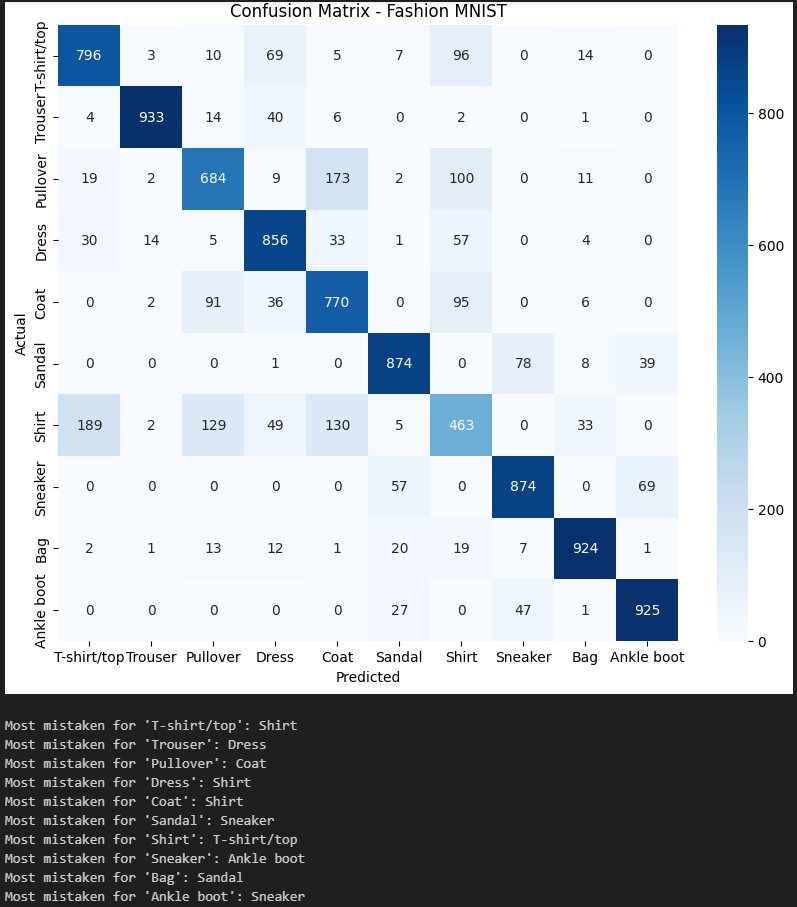
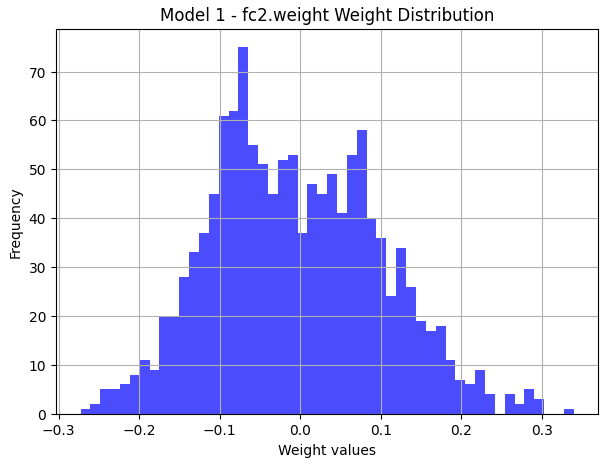
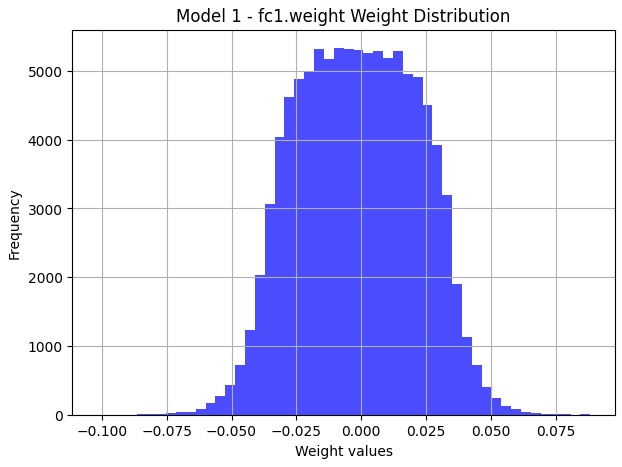


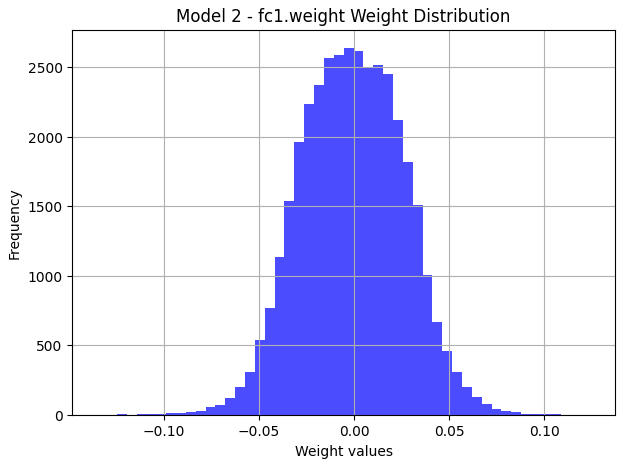
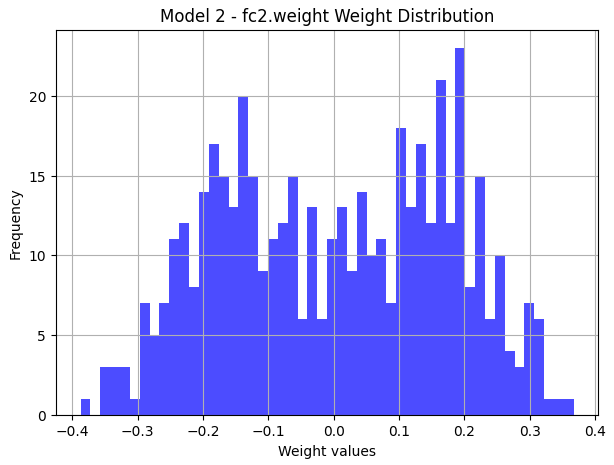
Figure 1: Heatmap of confusion matrix

c) Increasing the complexity of a model with more neurons and layers typically enhances its ability to learn complex, non-linear relationships in the data, which often leads to improved performance on challenging tasks. However, it’s essential to balance this complexity to avoid overfitting and ensure efficient training.

d) There are many measures to evaluate and choose the best model; I have mentiond some of them. To choose the best model configuration, use cross-validation to evaluate performance across multiple data splits, reducing variance. Metrics like accuracy, precision, recall, and F1-score are crucial for classification, while mean squared error (MSE) is standard for regression. Validation loss helps monitor overfitting; a low validation loss indicates good generalization. Use techniques like grid search or random search to explore configurations.

2-۱. آموزش دو مدل متفاوت

Histograms for the first models:

Histograms for the first models:

1. Base on the provided histograms, models have differences in the variance and frequencies in the first layer. The first model has less variance and more frequency. For the second layer, the first model recalls normal distribution more the second moedel. Also contains more frequency in less variance. Base on results, the second model provides us with more accurate model, since consists of more number of hidden neurons.
2. As has been mentiond in notebook, training process converges and findes the local optimum better and more efficient using Adam backpropagation.

جدول 1. Accuaracies of two models

|  |  |
| --- | --- |
| **Accuracy** |  |
|  | The First model |
| **شبکه‌ی دوم** | The Second model |

۳-۱. الگوریتم بازگشت به عقب

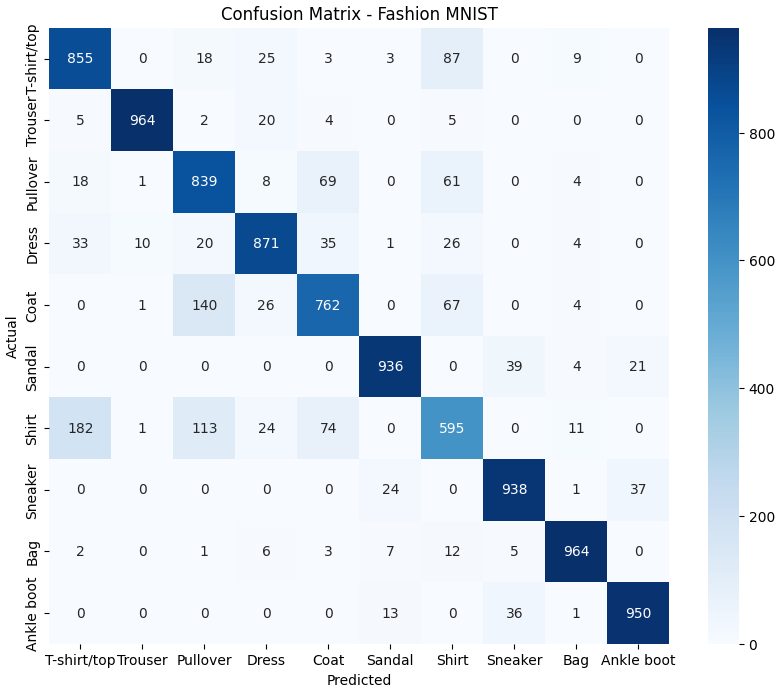
1. Adam (Adaptive Moment Estimation): Adam combines the concepts of momentum and adaptive learning rates. It keeps track of an exponentially decaying average of past gradients (momentum) and the squared gradients (adaptive learning rate). Adam is popular for its fast convergence and ability to handle sparse gradients well, making it suitable for a wide range of tasks.

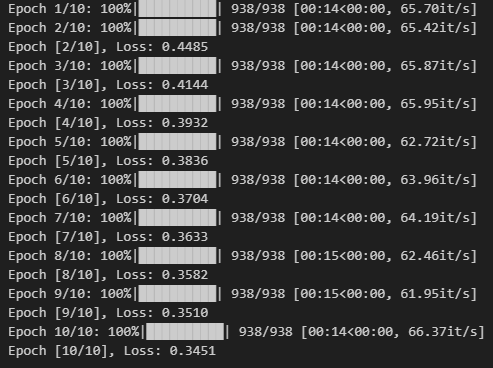
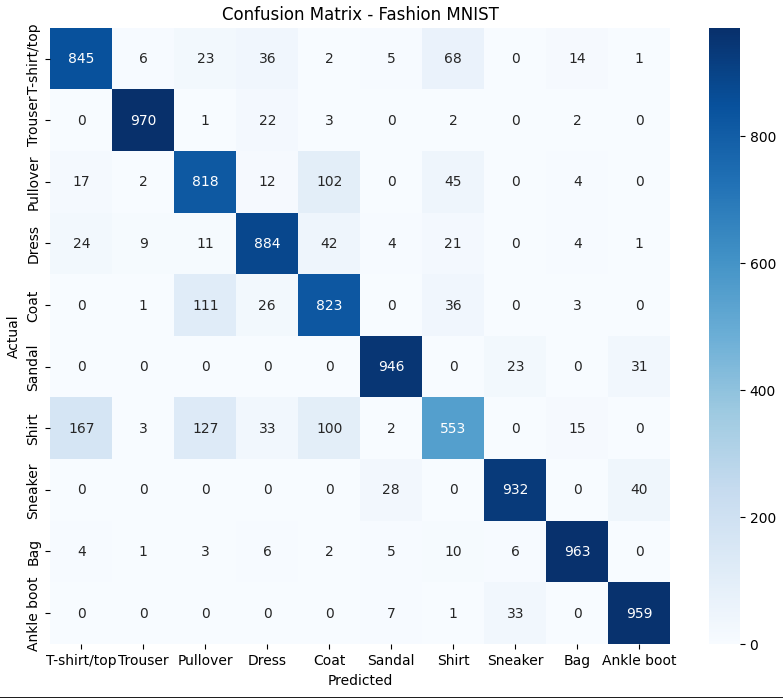
NAdam (Nesterov-accelerated Adaptive Moment Estimation): NAdam is a variant of Adam that incorporates Nesterov momentum. Unlike Adam, which updates the parameters based on the gradient at the current position, NAdam applies momentum to make a “look-ahead” adjustment before calculating the gradient. This typically results in faster convergence and can slightly improve performance.

RMSprop (Root Mean Square Propagation): RMSprop adjusts the learning rate for each parameter based on an exponentially decaying average of the squared gradients. It prevents oscillations and stabilizes training by reducing the step size for parameters with large gradients, making it well-suited for non-stationary or noisy objectives and widely used in deep learning.

1. An comaritoon has been provided base on taining on the same structur with different optimization methods:
2. Adam: Shows better convergance rate and accuracy:

A screenshot of a computer

Description automatically generated

1. Nadam: Demonstrates medium convergace rate and accuracy:
2. RMSprop: Demonstrates the worst results between these three methods:

A chart with numbers and symbols

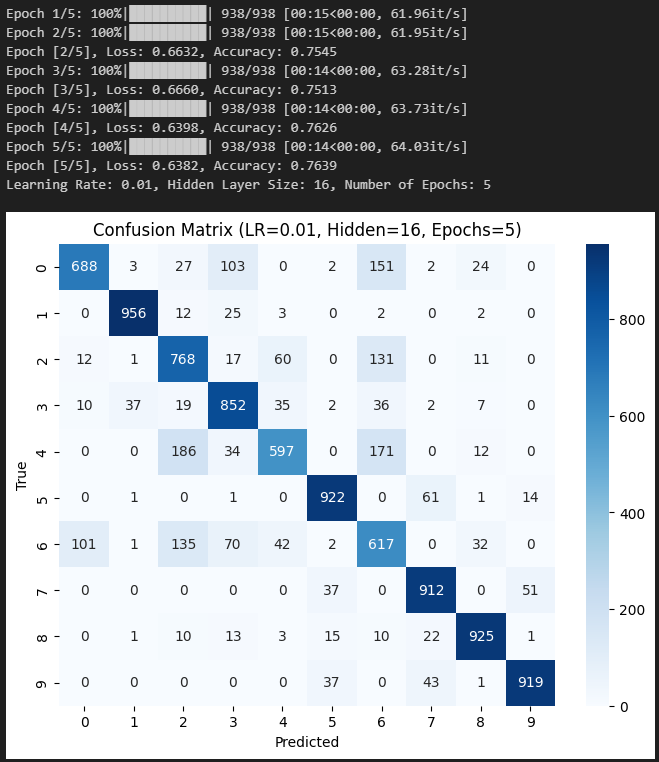
Description automatically generated with medium confidence

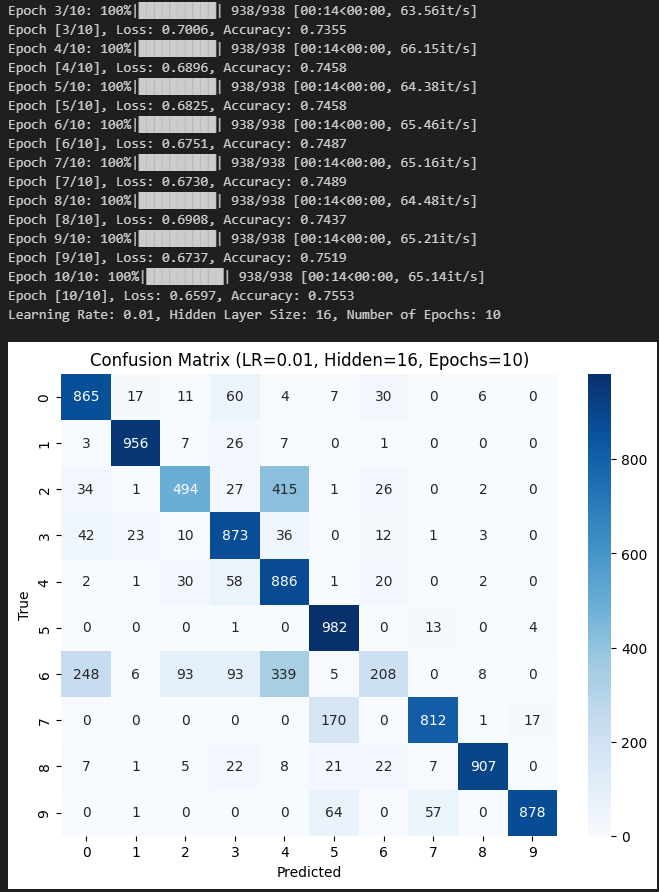
Bayesian and interpolation methods can improve the convergance rate, if they find some conditions to avoid them from infinite loops (like set bounds for the minimum loss).

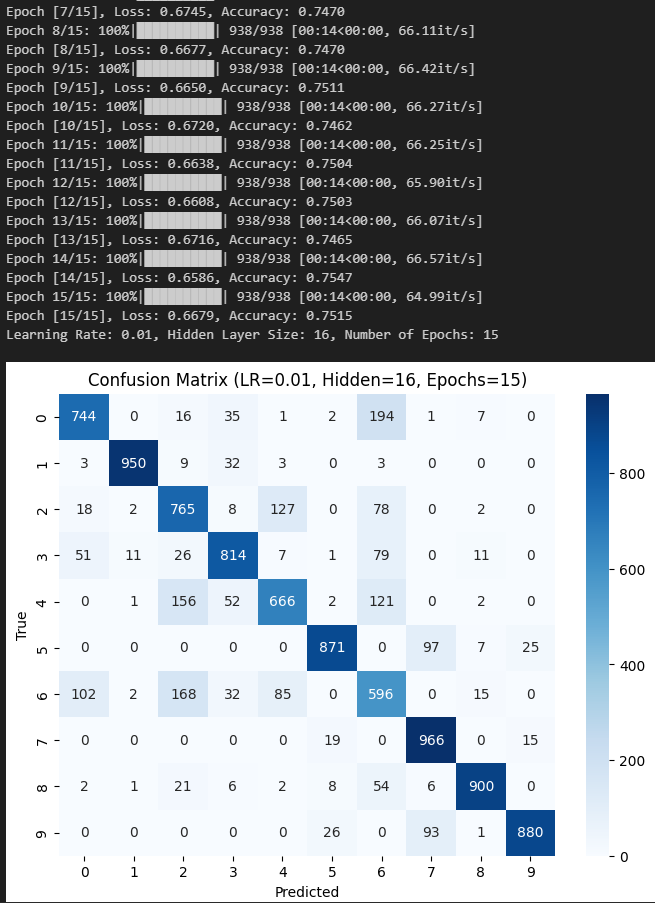
۴-۱. بررسی هایپرپارامترهای متفاوت

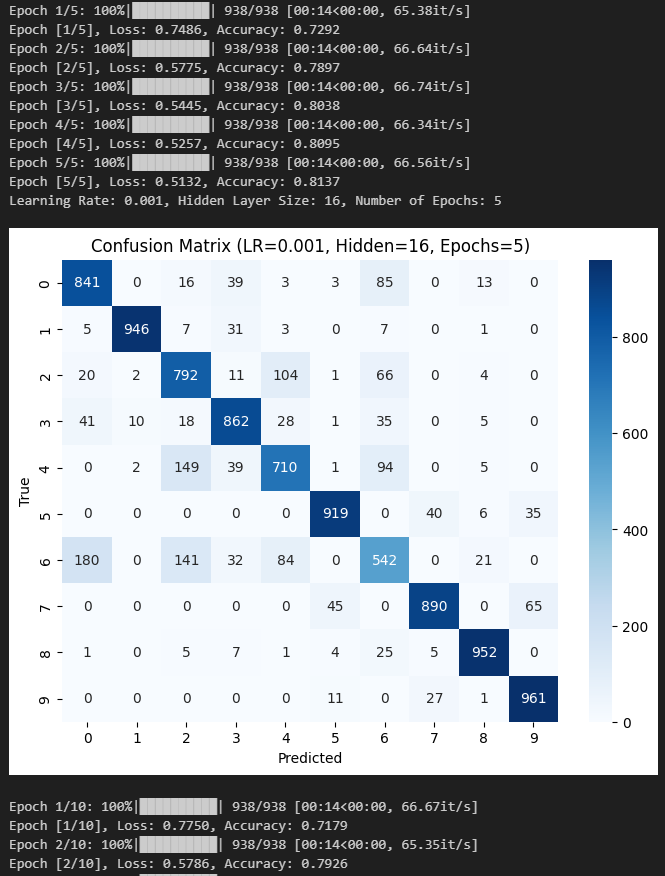
Hyperparameter optimization methods like **random search** help find the best model configuration by systematically testing different combinations of hyperparameters. Unlike grid search, which tests every possible combination, random search samples combinations randomly within specified ranges. This allows for broader coverage of the hyperparameter space with fewer trials, often leading to comparable or better results in less time. By focusing on diverse areas of the search space, random search increases the likelihood of finding effective hyperparameters for complex models.

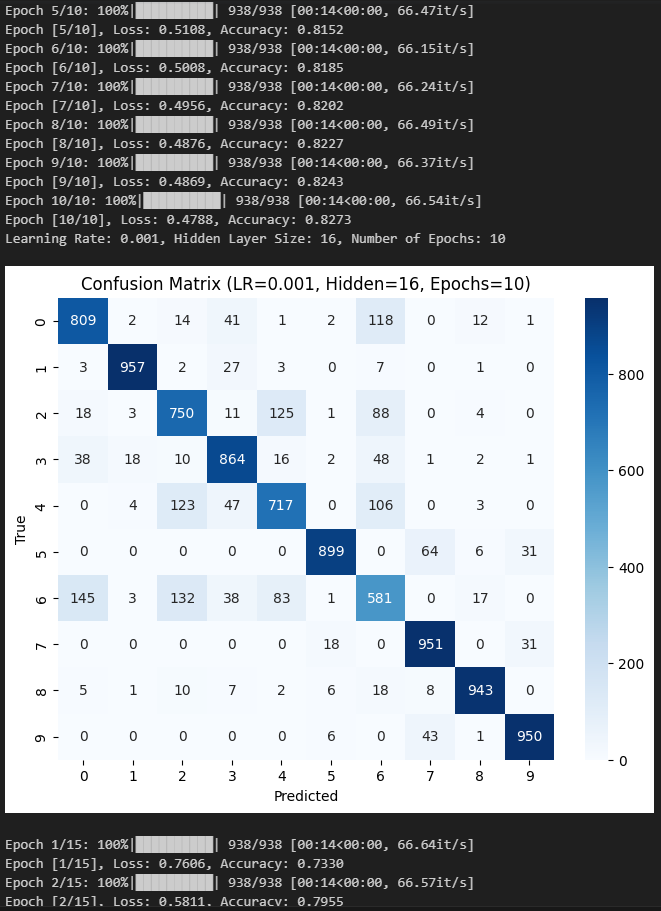
# These are the results for different hyperparameters. I could not provide them all, there were 27 combinations. Other results are in notebook. (I put it as a header to be convenient to close):





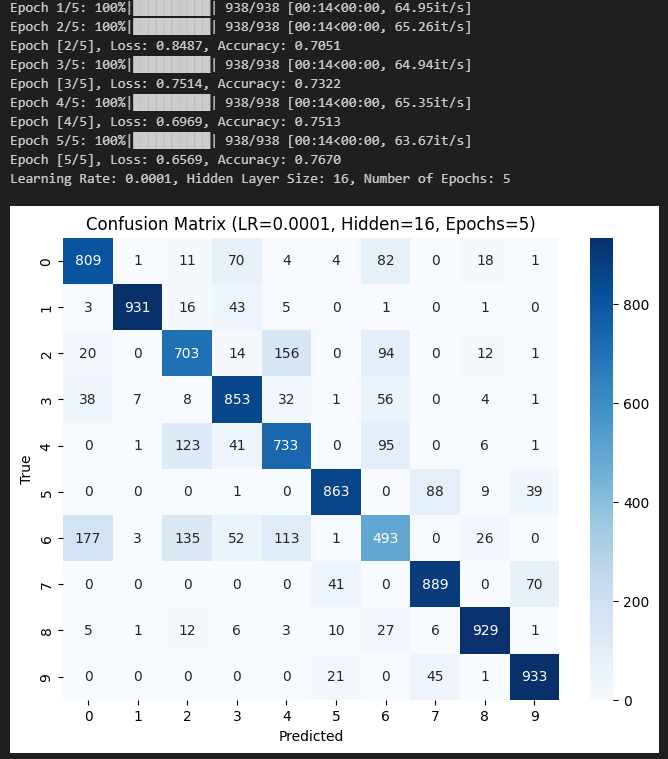






A screenshot of a computer

Description automatically generated



A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

We can not say a special hyperparameter belongs to a preidentified feature, theire combinations gives many different results. But in general the combination of LR=0.01, Epochs=10 and hidden neurons = 100 gives the best results.

.

# **پرسش ۲** **– آموزش و ارزیابی یک شبکه عصبی ساده**

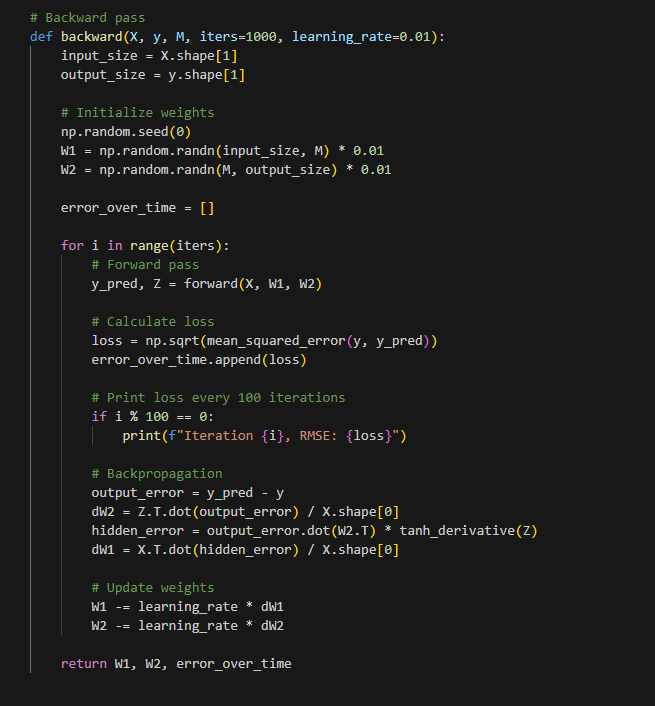
## ۱-۲. **آموزش یک شبکه عصبی ساده**



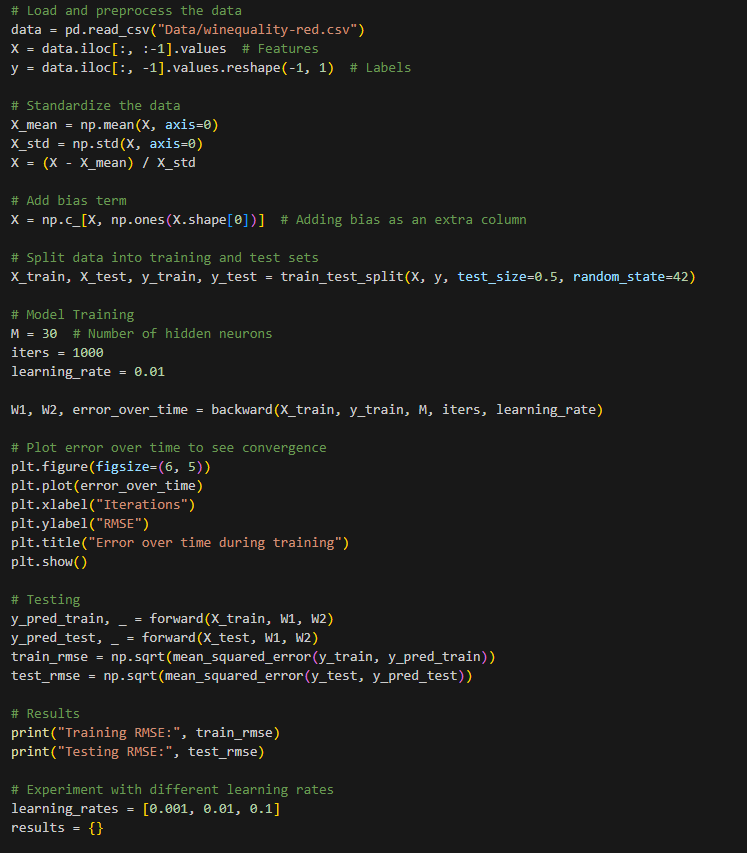
A screen shot of a computer program

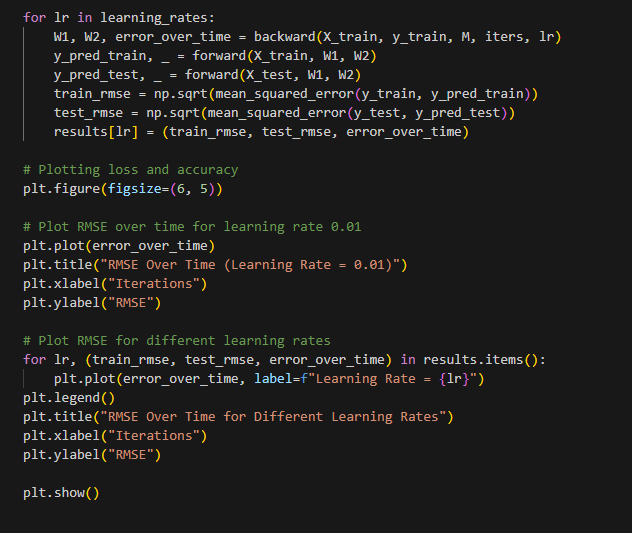
Description automatically generated



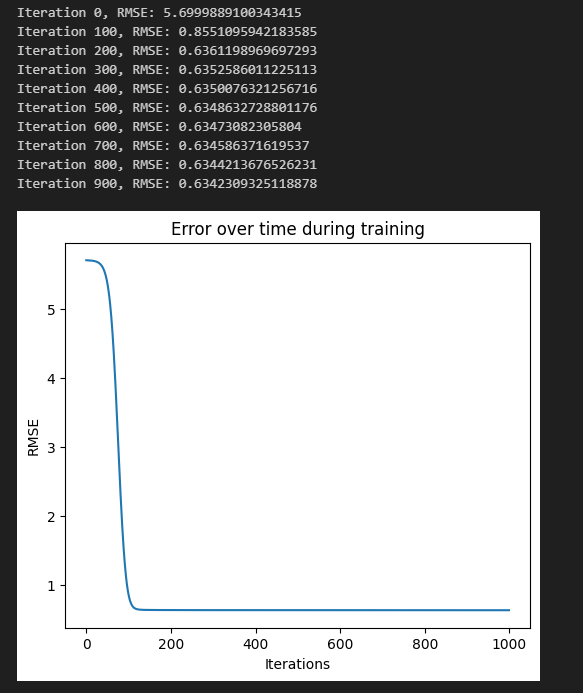


## 2-۲. **آموزش یک شبکه عصبی ساده**



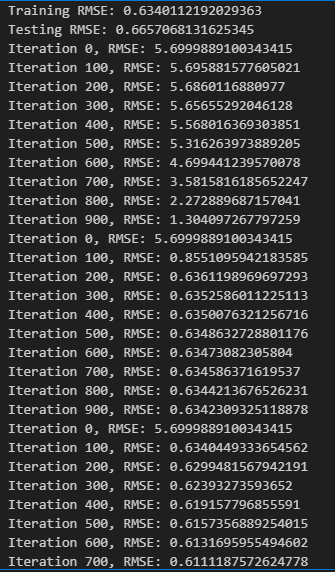


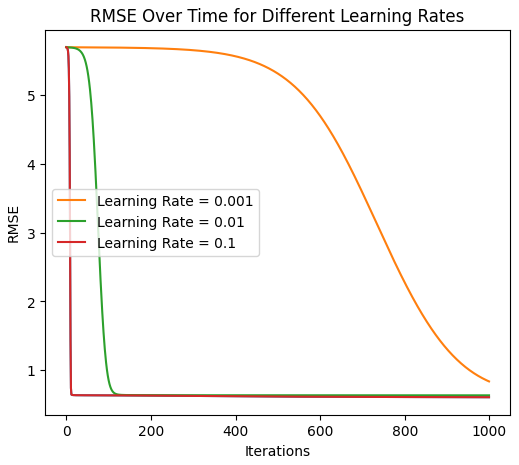
Here are the results:



For different values:

A black screen with white text

Description automatically generated



The best output belongs to LR with the value of 0.01, since it converges smoothly and gives the best output. As this is a simple costum neural network, and tasks and architectures are vary in different problems, it can not be any garaunty to work in other architectures.

# **پرسش ۳** **–** Madaline

## ۱-۳. **MRI and MRII**

Madaline Rule I

The MRI rule allows the adaptation of a first layer of hard limited (signum) Adaline elements whose outputs provide inputs to a second layer, consisting of a single fixed-threshold-logic element which may be, for example, the OR gate, AND gate, or majority-vote-taker discussed previously. The weights of the Adalines are initially set to small random values.

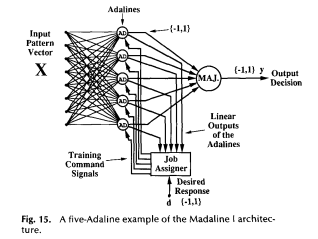
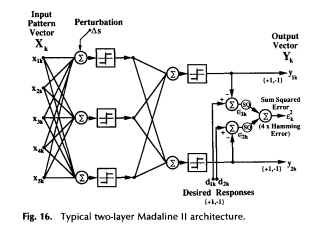


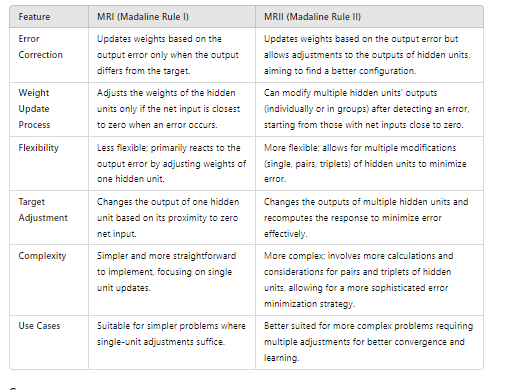
Figure 15 shows a Madaline I architecture with five fully connected first-layer Adalines. The second layer is a majority element (MAJ). Because the second-layer logic element is fixed and known, it is possible to determine which first layer Adalines can be adapted to correct an output error. The Adalines in the first layer assist each other in solving problems by automatic load-sharing. One procedure for training the network in Fig. 15follows. A pattern is presented, and if the output response of the majority element matches the desired response, no adaptation takes place. However, if, for instance, the desired response is +I and three of the five Adalines read -1 for a given input pattern, one of the latter three must be adapted to the +I state. The element that is adapted by MRI is the one whose linear outputs k is closest to zero-the one whose analog response is closest to the desired response. If more of the Adalines were originally in the -1 state, enough of them are adapted to the +I state to make the majority decision equal +I. The elements adapted are those whose linear outputs are closest to zero. A similar procedure is followed when the desired response is -1. When adapting a given element, the weight vector can be moved in the LMS direction far enough to reverse the Adaline's output (absolute correction, or "fast" learning), or it can be adapted by the small increment determined by the a-LMS algorithm (statistical, or "slow" learning). The one desired response dk is used for all Adalines that are adapted. The procedure can also be modified to allow one of Mays's rules to be used. In that event, for the case we have considered (majority output element), adaptations take place if at least half of the Adalines either have outputs differing from the desired response or have analog outputs which are in the dead zone. By setting the dead zone of Mays's increment adaptation rule to zero, the weights can also be adapted by Rosenblatt's Perceptron rule. Differences in initial conditions and the results of subsequent adaptation cause the various elements to take "responsibility" for certain parts of the training problem. The basic principle of load sharing is summarized thus: Assign responsibility to the Adaline or Adalines that can most easily assume it. In Fig. 15, the “job assigner,” a purely mechanized process, assigns responsibility during training by transferring the appropriate adapt commands and desired response signals to the selected Adalines. The job assigner utilizes linear-output information. Load sharing is important, since it results in the various adaptive elements developing individual weight vectors. If all the weights vectors were the same, there would be no point in having more than one element in the first layer. When training the Madaline, the pattern presentation sequence should be random. Experimenting with this, Ridgway found that cyclic presentation of the patterns could lead to cycles of adaptation. These cycles would cause the weights of the entire Madaline to cycle, preventing convergence. The adaptive system of Fig. 15 was suggested by common sense, and was found to work well in simulations. Ridgway found that the probability that a given Adaline will be adapted in response to an input pattern is greatest if that element had taken such responsibility during the previous adapt cycle when the pattern was most recently presented. The division of responsibility stabilizes at the same time that the responses of individual elements stabilize to their share of the load. When the training problem is not perfectly separable by this system, the adaptation process tends to minimize error probability, although it is possible for the algorithm to “hang up” on local optima. The Madaline structure of Fig. 15 has 2 layers-the first layer consists of adaptive logic elements, the second of fixed logic. A variety of fixed-logic devices could be used for the second layer. A variety of MRI adaptation rules were devised by Hoff that can be used with all possible fixed-logic output elements. An easily described training procedure results when the output element is an OR gate. During training, if the desired output for a given input pattern is +I, only the one Adaline whose linear output is closest to zero would be adapted if any adaptation is needed-in other words, if all Adalines give -1 outputs. If the desired output is -1, all elements must give -1 outputs, and any giving +I outputs must be adapted. The MRI rule obeys the “minimal disturbance principle” in the following sense. No more Adaline elements are adapted than necessary to correct the output decision and any dead-zone constraint. The elements whose linear outputs are nearest to zero are adapted because they require the smallest weight changes to reverse their output responses. Furthermore, whenever an Adaline is adapted, the weights are changed in the direction of its input vector, providing the requisite error correction with minimal weight change.

Madaline Rule II

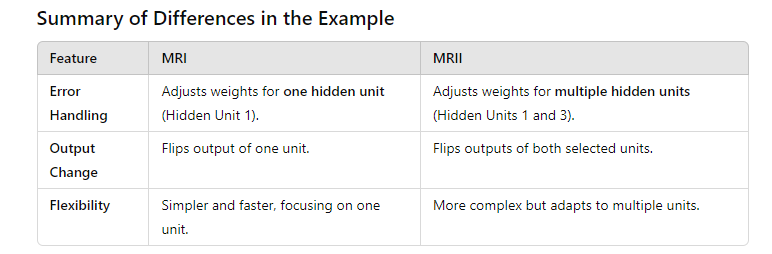
The MRI rule was recently extended to allow the adaptation of multilayer binary networks by Winter and Widrow with the introduction of Madaline Rule II (MRII). A typical two-layer MRll network is shown in Fig. 16. The weights in both layers are adaptive. Training with the MRll rule is similar to training with the MRI algorithm. The weights are initially set to small random values. Training patterns are presented in a random sequence. If the network produces an error during a training presentation, we begin by adapting first-layer Adalines.



By the minimal disturbance principle, we select the first layer Adaline with the smallest linear output magnitude and perform a “trial adaptation” by inverting its binary output. This can be done without adaptation by adding a perturbation ∆s of suitable amplitude and polarity to the Adaline’s sum (refer to Fig. 16). If the output Hamming error is reduced by this bit inversion, that is, if the number of output errors is reduced, the perturbation ∆s is removed and the weights of the selected Adaline element are changed by α-LMS in a direction collinear with the corresponding input vector the direction that reinforces the bit reversal with minimal disturbance to the weights. Conversely, if the trial adaptation does not improve the network response, no weight adaptation is performed. After finishing with the first element, we perturb and update other Adalines in the first layer which have “sufficiently small” linear-output magnitudes. Further error reductions can be achieved, if desired, by reversing pairs, triples, and so on, up to some predetermined limit. After exhausting possibilities with the first layer, we move on to the next layer and proceed in a like manner. When the final layer is reached, each of the output elements is adapted by α-LMS. At this point, a new training pattern is selected at random and the procedure is repeated. The goa1 is to reduce Hamming error with each presentation, thereby hopefully minimizing the average Hamming error over the training set. Like MRI, the procedure can be modified so that adaptations follow an absolute correction rule or one of Mays‘s rules rather than α-LMS. Like MRI, MRll can “hang up” on local optima.



Differences



References:

[1] Widrow, B., & Lehr, M. A. (1990). 30 years of adaptive neural networks: perceptron, madaline, and backpropagation. *Proceedings of the IEEE*, *78*(9), 1415-1442.

## 2-۳.نمودار پراکندگی دادهها

Importing several libraries commonly used for data manipulation, visualization, and machine learning.

Reading a CSV file named MadaLine.csv into a Pandas DataFrame

df = pd.read\_csv('/content/MadaLine.csv',names=['x', 'y','label'],header=None)

df

It is used to shuffle the rows of the DataFrame

df = df.sample(frac = 1)

df

It extracts specific columns from the DataFrame df and converts them into a NumPy array.

data = df[['x','y']]

inputs = data.to\_numpy()

inputs.shape[0]

It converts the labels in the target array to a bipolar format.

# Convert to Bipolar

label = df[['label']]

target = label.to\_numpy()

target[np.isclose(target, 0)] = -1

It creates two new DataFrames, df0 and df1, that filter the original DataFrame df based on the values in the label column.

df0 = df.loc[df['label']  == 0 ]

df1 = df.loc[df['label']  == 1]

A scatter plot using Matplotlib to visualize the data points from two different classes.

# scatter plot

plt.scatter(df0['x'], df0['y'], c="red", linewidths=2)

plt.scatter(df1['x'], df1['y'], c="blue", linewidths=2)

# add axis labels and legend

plt.xlabel("x")

plt.ylabel("y")

plt.legend(["Class 1", "Class 2"])

# add grid

plt.grid(True)

# add title

plt.title("Scatter Plot")

# adjust subplot spacing

plt.tight\_layout()

# show the plot

plt.show()

 **Scatter Plot**:

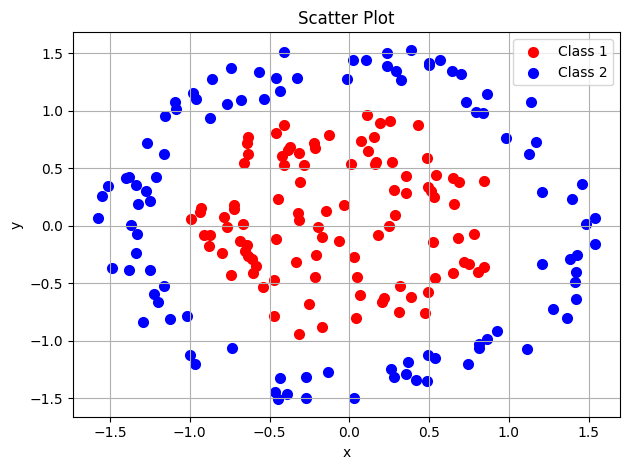
* **plt.scatter(df0['x'], df0['y'], c="red", linewidths=2)**: This line creates a scatter plot of the points in df0, with the x-coordinates from the x column and y-coordinates from the y column, colored in red.
* **plt.scatter(df1['x'], df1['y'], c="blue", linewidths=2)**: This line adds another scatter plot for the points in df1, colored in blue.

 **Axis Labels and Legend**:

* **plt.xlabel("x")**: This sets the label for the x-axis to "x".
* **plt.ylabel("y")**: This sets the label for the y-axis to "y".
* **plt.legend(["Class 1", "Class 2"])**: This adds a legend to the plot, labeling the red points as "Class 1" and the blue points as "Class 2".

 **Grid and Title**:

* **plt.grid(True)**: This adds a grid to the plot for better visualization.
* **plt.title("Scatter Plot")**: This sets the title of the plot to "Scatter Plot".



3-۳.آموزش مدل

## **MRI Implementation**

The MRI function incorporates training, forward propagation, error calculation, and optional plotting of the results.

def find\_decision\_boundary(start\_x, end\_x, weights, biases):

    """

    Calculates the decision boundary given the start and end x values, weights, and biases.

    Args:

    start\_x (float): The start x value.

    end\_x (float): The end x value.

    weights (numpy.ndarray): An array of weights.

    biases (float): The bias value.

    Returns:

    inputs (numpy.ndarray): An array of x values.

    output (numpy.ndarray): An array of y values representing the decision boundary.

    """

    # Create an array of x values

    inputs = np.linspace(start\_x, end\_x)

    # Calculate the corresponding y values using the given weights and biases

    output = -(weights[0] \* inputs + biases)

    output = output / weights[1]

    return inputs, output

def initialize\_weights(sm, num\_neurons\_layer1, num\_neurons\_layer2):

    # Set a random seed for reproducibility

    np.random.seed(10)

    # Generate random weights and biases for the first layer

    weights = np.random.rand(num\_neurons\_layer1, num\_neurons\_layer2) \* sm

    biases = np.zeros((num\_neurons\_layer1, 1))

    # Initialize weights and biases for the second layer

    # The weights for the second layer are initialized to 1

    weights\_layer2 = np.array([[1]\*num\_neurons\_layer1])

    biases\_layer2 = num\_neurons\_layer1 - 1

    # Return all the initialized weights and biases

    return weights, biases, weights\_layer2, biases\_layer2

def apply\_activation\_function(net):

    # np.where(condition, x, y) returns an array with the same shape as condition,

    # where the elements are taken from x where condition is True,

    # and from y elsewhere. In this case, the condition is whether each element of

    # the input net is greater than or equal to 0. If it is, the corresponding element

    # in the output h is set to 1; otherwise, it is set to -1.

    h = np.where(net >= 0, 1, -1)

    return h

def forward\_propagation(weights, inputs, biases, should\_reshape):

    # Check if inputs should be reshaped

    if should\_reshape:

        inputs = inputs.reshape((2, 1))

    # Calculate the net input

    net\_input = np.dot(weights, inputs) + biases

    # Apply activation function to net input to obtain outputs

    outputs = apply\_activation\_function(net\_input)

    # Return both net input and outputs

    return net\_input, outputs

def update\_weights(weights, biases, inputs, target, net\_input, output, learning\_rate, num\_neurons\_layer1):

    # Reshape inputs and net\_input

    inputs = inputs.reshape((1, 2))

    net\_input = net\_input.reshape((num\_neurons\_layer1, 1))

    # If target is equal to output, no weight or bias update is necessary

    if target == output:

        return weights, biases

    # If target is 1 and output is -1, update weights and biases for the neuron with the highest net input

    elif target == 1 and target != output:    #output=-1 but target=1

        argmax\_neuron = np.argmax(net\_input)

        diff\_bias = learning\_rate \* (1 - net\_input[argmax\_neuron])

        diff\_weight = learning\_rate \* np.dot((1 - net\_input[argmax\_neuron]), inputs)

        biases[argmax\_neuron] = biases[argmax\_neuron] + diff\_bias

        weights[argmax\_neuron] = weights[argmax\_neuron] + diff\_weight

    # If target is -1 and output is 1, update weights and biases for all neurons with positive net input

    elif target == -1 and target != output:   # output=1 but target=-1

        positive\_indices = np.argwhere(net\_input > 0)

        diff\_bias = learning\_rate \* (-1 - net\_input)

        diff\_weight = learning\_rate \* np.dot((-1 - net\_input), inputs)

        new\_biases = biases + diff\_bias

        new\_weights = weights + diff\_weight

        for i in positive\_indices[:, 0]:

            weights[i] = new\_weights[i]

            biases[i] = new\_biases[i]

    # Return updated weights and biases

    return weights, biases

# Define a function named calculate\_error

def calculate\_error(target, output):

    # Calculate the error using the mean squared error formula

    error = 0.5 \* np.power((target - output), 2)

    # Return the calculated error

    return error

def predict(inputs, target, weights, biases, num\_neurons\_layer1):

    # initialize an empty list to store predicted outputs

    predicted\_output = []

    # initialize biases\_layer2 as a numpy array of zeros

    biases\_layer2 = np.zeros((num\_neurons\_layer1, 1))

    # initialize weights\_layer2 as a numpy array of shape (1, num\_neurons\_layer1) with all elements as 1

    weights\_layer2 = np.array([[1]\*num\_neurons\_layer1])

    # update biases\_layer2 to have a value of num\_neurons\_layer1 - 1

    biases\_layer2 = num\_neurons\_layer1 - 1

    # loop through each input and calculate the predicted output using forward propagation

    for i in range(inputs.shape[0]):

        # call the forward\_propagation function with inputs, weights, biases, and should\_reshape=True

        net\_input, outputs = forward\_propagation(weights, inputs[i], biases, should\_reshape=True)

        # call the forward\_propagation function with weights\_layer2, outputs, biases\_layer2, and should\_reshape=False

        net\_input2, output = forward\_propagation(weights\_layer2, outputs, biases\_layer2, should\_reshape=False)

        # append the predicted output to the predicted\_output list

        predicted\_output.append(output[0])

    # return the list of predicted outputs

    return predicted\_output

def MRI(df0, df1, inputs, target, num\_neurons\_layer1=3, num\_neurons\_layer2=2, learning\_rate=0.0001, max\_iter=200, samples=None, plot=True):

    # If samples is not provided, set it to the number of rows in the input data.

    if samples is None:

        samples = inputs.shape[0]

    # Print the number of samples.

    print('sample:', samples)

    # Initialize variables

    sm = 0.001

    error\_list = []

    errors = []

    mean\_error = 10\*\*3

    weights, biases, weights\_layer2, biases\_layer2 = initialize\_weights(sm, num\_neurons\_layer1, num\_neurons\_layer2)  # Step 0

    # Iterate through the training process

    for i in range(max\_iter):

        # Perform forward propagation

        net\_input, outputs = forward\_propagation(weights, inputs[i % samples], biases, should\_reshape=True)  # Step 4 and 5

        net\_input2, output = forward\_propagation(weights\_layer2, outputs, biases\_layer2, should\_reshape=False)  # Step 6

        # Calculate the error of the output

        error = calculate\_error(target[i % samples], output)

        errors.append(error)

        # If an epoch has ended, calculate the mean error of the epoch and append it to the error list

        if i % samples == 0 and i != 0:

            mean\_error = np.mean(errors)

            error\_list.append(mean\_error)

            errors = []

            # Print the epoch number and the mean error of the epoch

            print('Epoch %d / %d' % (len(error\_list), int(max\_iter / samples)))

            print('loss:', mean\_error)

            for j in range(len(weights)):

                # Print the weights and biases for each layer

                print('W%d:'%(j+1),weights[j])

                print('b%d:'%(j+1),biases[j])

        # If the mean error is 0 or the difference between the mean error of the last two epochs is 0, an early stop occurred

        if mean\_error == 0 or (i > 50 and len(error\_list) >= 2 and error\_list[-1] - error\_list[-2] == 0):

            print('An early stop occurred!')

            # If plot is True, plot the error, decision boundary, confusion matrix, and classification report

            if plot:

                plt.plot(error\_list)

                plt.xlabel('Epochs')

                plt.ylabel('Mean Squared Error')

                plt.title('Error Plot')

                plt.grid(True)

                plt.show()

                df0 = df0

                df1 = df1

                plt.scatter(df0['x'], df0['y'], c ="red", linewidths = 0.1)

                plt.scatter(df1['x'], df1['y'], c ="blue", linewidths = .1)

                # Plot the decision boundary

                for i in range(num\_neurons\_layer1):

                    px1, px2 = find\_decision\_boundary(-2, 2, weights[i], biases[i])

                    plt.plot(px1, px2)

                plt.xlabel("x")

                plt.ylabel("y")

                plt.legend(["Class 1" , "Class 2"])

                plt.xlim([-2, 2])

                plt.ylim([-2, 2])

                plt.show()

                # Generate predictions

                predicted\_output = predict(inputs, target, weights, biases, num\_neurons\_layer1)

                # Create confusion matrix

                cm = confusion\_matrix(target, predicted\_output)

                # Plot heatmap of confusion matrix

                plt.figure(figsize=(6, 4))

                sns.heatmap(cm, annot=True, cmap="Blues", fmt="g")

                plt.xlabel("Predicted labels")

                plt.ylabel("True labels")

                plt.title("Confusion Matrix")

                plt.tight\_layout()

                plt.show()

                predicted\_output = predict(inputs, target,weights,biases,num\_neurons\_layer1)

                print(classification\_report(target, predicted\_output))

            return weights, biases, error\_list

        weights, b = update\_weights(weights, biases, inputs[i % samples], target[i % samples], net\_input, output, learning\_rate, num\_neurons\_layer1)  # Step 7

    if plot:

        plt.plot(error\_list)

        plt.xlabel('Epochs')

        plt.ylabel('Mean Squared Error')

        plt.title('Error Plot')

        plt.grid(True)

        plt.show()

        df0 = df0

        df1 = df1

        plt.scatter(df0['x'], df0['y'], c ="red", linewidths = 0.1)

        plt.scatter(df1['x'], df1['y'], c ="blue", linewidths = .1)

        for i in range(num\_neurons\_layer1):

            px1, px2 = find\_decision\_boundary(-2, 2, weights[i], biases[i])

            plt.plot(px1, px2)

        plt.xlabel("x")

        plt.ylabel("y")

        plt.legend(["Class 1" , "Class 2"])

        plt.xlim([-2, 2])

        plt.ylim([-2, 2])

        plt.show()

        # Generate predictions

        predicted\_output = predict(inputs, target, weights, biases, num\_neurons\_layer1)

        # Create confusion matrix

        cm = confusion\_matrix(target, predicted\_output)

        # Plot heatmap of confusion matrix

        plt.figure(figsize=(6, 4))

        sns.heatmap(cm, annot=True, cmap="Blues", fmt="g")

        plt.xlabel("Predicted labels")

        plt.ylabel("True labels")

        plt.title("Confusion Matrix")

        plt.tight\_layout()

        plt.show()

        predicted\_output = predict(inputs, target,weights,biases,num\_neurons\_layer1)

        print(classification\_report(target, predicted\_output))

    return weights, biases, error\_list

import random

import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.metrics import confusion\_matrix

from sklearn.metrics import classification\_report

def find\_decision\_boundary(start\_x, end\_x, weights, biases):

    inputs = np.linspace(start\_x, end\_x)

    output = -(weights[0] \* inputs + biases)

    output = output / weights[1]

    return inputs, output

def initialize\_weights(sm, num\_neurons\_layer1, num\_neurons\_layer2):

    np.random.seed(10)

    weights = np.random.rand(num\_neurons\_layer1, num\_neurons\_layer2) \* sm

    biases = np.zeros((num\_neurons\_layer1, 1))

    weights\_layer2 = np.array([[1]\*num\_neurons\_layer1])

    biases\_layer2 = num\_neurons\_layer1 - 1

    return weights, biases, weights\_layer2, biases\_layer2

def apply\_activation\_function(net):

    h = np.where(net >= 0, 1, -1)

    return h

def forward\_propagation(weights, inputs, biases, should\_reshape):

    if should\_reshape:

        inputs = inputs.reshape((2, 1))

    net\_input = np.dot(weights, inputs) + biases

    outputs = apply\_activation\_function(net\_input)

    return net\_input, outputs

def update\_weights(weights, biases, inputs, target, net\_input, output, learning\_rate, num\_neurons\_layer1):

    inputs = inputs.reshape((1, 2))

    net\_input = net\_input.reshape((num\_neurons\_layer1, 1))

    if target == output:

        return weights, biases

    elif target == 1 and target != output:    #output=-1 but target=1

        argmax\_neuron = np.argmax(net\_input)

        diff\_bias = learning\_rate \* (1 - net\_input[argmax\_neuron])

        diff\_weight = learning\_rate \* np.dot((1 - net\_input[argmax\_neuron]), inputs)

        biases[argmax\_neuron] = biases[argmax\_neuron] + diff\_bias

        weights[argmax\_neuron] = weights[argmax\_neuron] + diff\_weight

    elif target == -1 and target != output:   # output=1 but target=-1

        positive\_indices = np.argwhere(net\_input > 0)

        diff\_bias = learning\_rate \* (-1 - net\_input)

        diff\_weight = learning\_rate \* np.dot((-1 - net\_input), inputs)

        new\_biases = biases + diff\_bias

        new\_weights = weights + diff\_weight

        for i in positive\_indices[:, 0]:

            weights[i] = new\_weights[i]

            biases[i] = new\_biases[i]

    return weights, biases

def calculate\_error(target, output):

    error = 0.5 \* np.power((target - output), 2)

    return error

def predict(inputs, target, weights, biases, num\_neurons\_layer1):

    predicted\_output = []

    biases\_layer2 = np.zeros((num\_neurons\_layer1, 1))

    weights\_layer2 = np.array([[1]\*num\_neurons\_layer1])

    biases\_layer2 = num\_neurons\_layer1 - 1

    for i in range(inputs.shape[0]):

        net\_input, outputs = forward\_propagation(weights, inputs[i], biases, should\_reshape=True)

        net\_input2, output = forward\_propagation(weights\_layer2, outputs, biases\_layer2, should\_reshape=False)

        predicted\_output.append(output[0])

    return predicted\_output

def MRI(df0, df1, inputs, target, num\_neurons\_layer1=3, num\_neurons\_layer2=2, learning\_rate=0.0001, max\_iter=200, samples=None, plot=True):

    if samples is None:

        samples = inputs.shape[0]

    print('sample:', samples)

    sm = 0.001

    error\_list = []

    errors = []

    mean\_error = 10\*\*3

    weights, biases, weights\_layer2, biases\_layer2 = initialize\_weights(sm, num\_neurons\_layer1, num\_neurons\_layer2)  # Step 0

    for i in range(max\_iter):

        net\_input, outputs = forward\_propagation(weights, inputs[i % samples], biases, should\_reshape=True)  # Step 4 and 5

        net\_input2, output = forward\_propagation(weights\_layer2, outputs, biases\_layer2, should\_reshape=False)  # Step 6

        error = calculate\_error(target[i % samples], output)

        errors.append(error)

        if i % samples == 0 and i != 0:

            mean\_error = np.mean(errors)

            error\_list.append(mean\_error)

            errors = []

            print('Epoch %d / %d' % (len(error\_list), int(max\_iter / samples)))

            print('loss:', mean\_error)

            for j in range(len(weights)):

                print('W%d:'%(j+1),weights[j])

                print('b%d:'%(j+1),biases[j])

        if mean\_error == 0 or (i > 50 and len(error\_list) >= 2 and error\_list[-1] - error\_list[-2] == 0):

            print('An early stop occurred!')

            if plot:

                plt.plot(error\_list)

                plt.xlabel('Epochs')

                plt.ylabel('Mean Squared Error')

                plt.title('Error Plot')

                plt.grid(True)

                plt.show()

                df0 = df0

                df1 = df1

                plt.scatter(df0['x'], df0['y'], c ="red", linewidths = 0.1)

                plt.scatter(df1['x'], df1['y'], c ="blue", linewidths = .1)

                for i in range(num\_neurons\_layer1):

                    px1, px2 = find\_decision\_boundary(-2, 2, weights[i], biases[i])

                    plt.plot(px1, px2)

                plt.xlabel("x")

                plt.ylabel("y")

                plt.legend(["Class 1" , "Class 2"])

                plt.xlim([-2, 2])

                plt.ylim([-2, 2])

                plt.show()

                # Generate predictions

                predicted\_output = predict(inputs, target, weights, biases, num\_neurons\_layer1)

                # Create confusion matrix

                cm = confusion\_matrix(target, predicted\_output)

                # Plot heatmap of confusion matrix

                plt.figure(figsize=(6, 4))

                sns.heatmap(cm, annot=True, cmap="Blues", fmt="g")

                plt.xlabel("Predicted labels")

                plt.ylabel("True labels")

                plt.title("Confusion Matrix")

                plt.tight\_layout()

                plt.show()

                predicted\_output = predict(inputs, target,weights,biases,num\_neurons\_layer1)

                print(classification\_report(target, predicted\_output))

            return weights, biases, error\_list

        weights, b = update\_weights(weights, biases, inputs[i % samples], target[i % samples], net\_input, output, learning\_rate, num\_neurons\_layer1)  # Step 7

    if plot:

        plt.plot(error\_list)

        plt.xlabel('Epochs')

        plt.ylabel('Mean Squared Error')

        plt.title('Error Plot')

        plt.grid(True)

        plt.show()

        df0 = df0

        df1 = df1

        plt.scatter(df0['x'], df0['y'], c ="red", linewidths = 0.1)

        plt.scatter(df1['x'], df1['y'], c ="blue", linewidths = .1)

        for i in range(num\_neurons\_layer1):

            px1, px2 = find\_decision\_boundary(-2, 2, weights[i], biases[i])

            plt.plot(px1, px2)

        plt.xlabel("x")

        plt.ylabel("y")

        plt.legend(["Class 1" , "Class 2"])

        plt.xlim([-2, 2])

        plt.ylim([-2, 2])

        plt.show()

        # Generate predictions

        predicted\_output = predict(inputs, target, weights, biases, num\_neurons\_layer1)

        # Create confusion matrix

        cm = confusion\_matrix(target, predicted\_output)

        # Plot heatmap of confusion matrix

        plt.figure(figsize=(6, 4))

        sns.heatmap(cm, annot=True, cmap="Blues", fmt="g")

        plt.xlabel("Predicted labels")

        plt.ylabel("True labels")

        plt.title("Confusion Matrix")

        plt.tight\_layout()

        plt.show()

        predicted\_output = predict(inputs, target,weights,biases,num\_neurons\_layer1)

        print(classification\_report(target, predicted\_output))

    return weights, biases, error\_list

It defines a multi-layer neural network using Madaline architecture concepts.

**Function Definitions**:

* + find\_decision\_boundary: Calculates the decision boundary based on the weights and biases.
  + initialize\_weights: Initializes weights and biases for each layer.
  + apply\_activation\_function: Applies a threshold activation function to the net inputs.
  + forward\_propagation: Propagates inputs through the layers to get outputs.
  + update\_weights: Adjusts weights and biases based on the difference between the target and predicted output.
  + calculate\_error: Computes mean squared error for given targets and outputs.
  + predict: Generates predictions based on current model weights for the input data.

1. **Main Function (MRI)**:
   * This function is the core of the Madaline implementation and handles:
     + Initialization of weights and variables.
     + Iterative training with forward propagation and weight updating.
     + Early stopping if the error plateaus.
     + Visualization of the error over epochs, decision boundaries, and a confusion matrix.
2. **Plotting and Evaluation**:
   * Error plot, decision boundary plot, and confusion matrix are shown when the model converges or reaches the maximum iteration limit. It also outputs a classification report.

### Observations

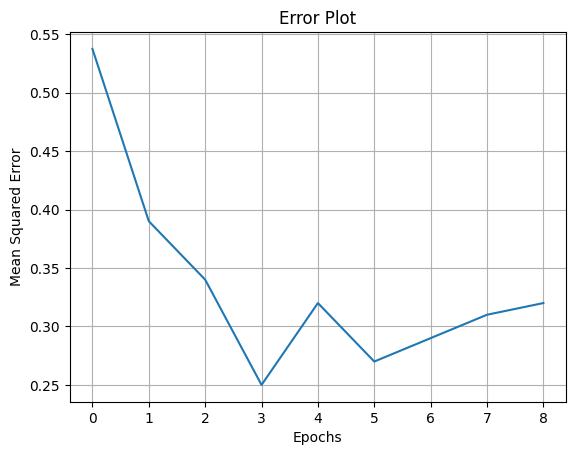
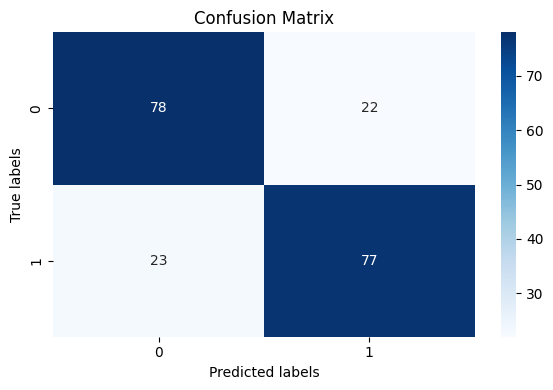
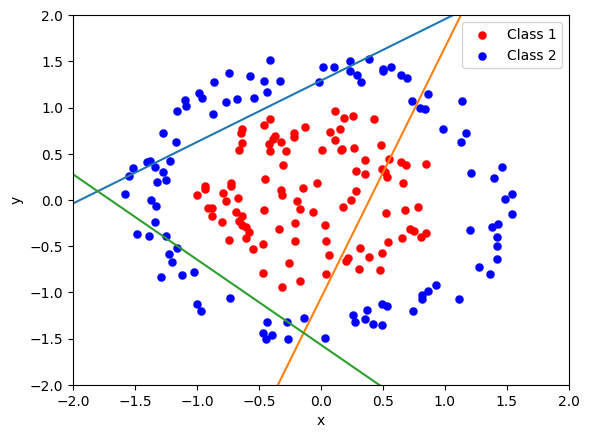
* **Learning Rate**: A small learning rate (0.0001), which can result in slow convergence. Consider increasing it slightly if training takes too long.
* **Early Stopping Condition**: The function halts if error differences are zero for two consecutive epochs. This is helpful in cases where a plateau is reached.
* **Decision Boundary Visualization**: The function plots decision boundaries and scatter points for each class to visualize how well the network is separating them.

4-۳.حلیل نتایج

**3 Neurons**

The MRI function with the specified parameters, you should ensure that df0, df1, inputs, and target are properly defined and structured.

weights, biases, error\_list = MRI(df0, df1, inputs,target,num\_neurons\_layer1 = 3 ,num\_neurons\_layer2 = 2,max\_iter = 2000,learning\_rate = 0.0001,samples = inputs.shape[0])

****

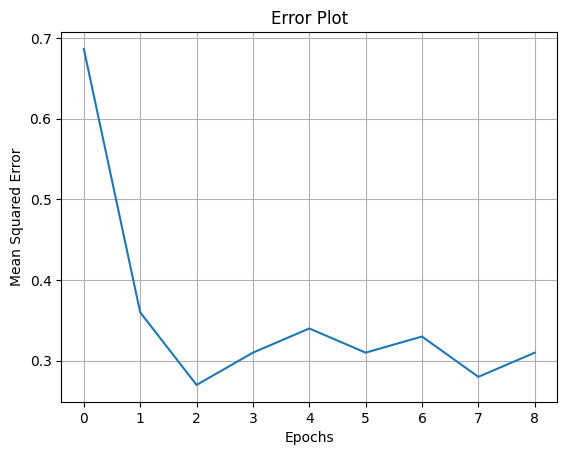
The configuration to keep only 3 neurons in the first layer while lowering the learning rate back to 0.0001.

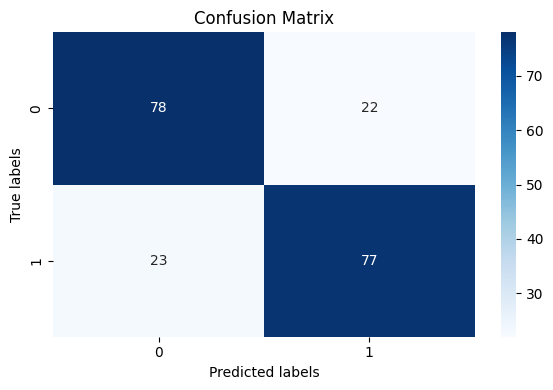
**Minimal First Layer Neurons**: With 3 neurons, the model’s representational power is very limited, which can be helpful for simpler, linearly separable data but might struggle with more complex data structures.

1. **Very Low Learning Rate (0.0001)**: The learning rate is quite low, which encourages stability and minimizes the risk of overshooting but will also slow down convergence. This rate can help ensure that the model moves gradually towards an optimal solution without large fluctuations in the error\_list.
2. **Controlled Training**: With a lower learning rate and fewer neurons, this setup is cautious, reducing the chance of overfitting or instability. If this configuration underperforms due to slow convergence, consider increasing max\_iter or switching to a slightly higher learning rate (e.g., 0.0005).

This configuration is suitable if you’re aiming for stability and generalization over speed.

weights, biases, error\_list = MRI(df0, df1, inputs,target,num\_neurons\_layer1 = 3 ,num\_neurons\_layer2 = 2,max\_iter = 2000,learning\_rate = 0.01,samples = inputs.shape[0])

****

****

With this configuration, the number of neurons in the first layer to 3 and the learning rate back to 0.01.

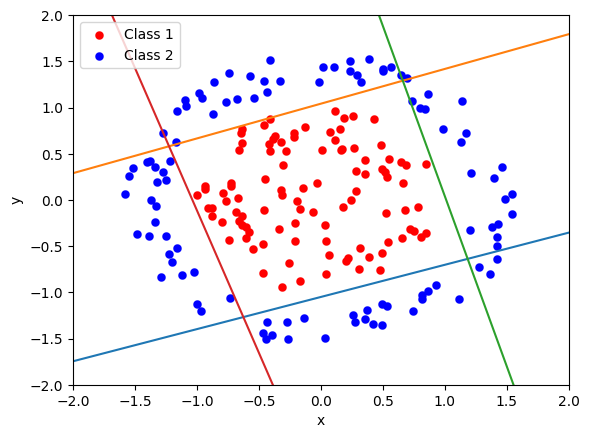
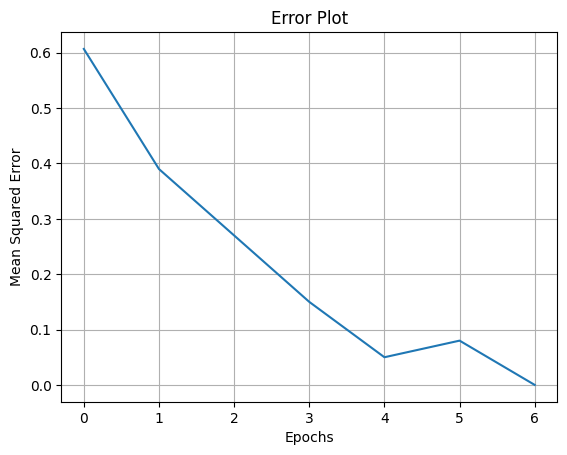
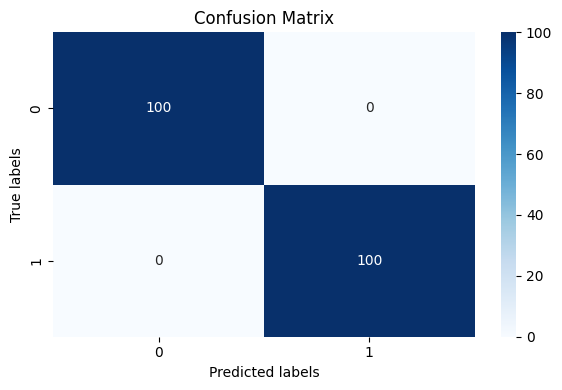
**Three Neurons in Layer 1**: With only 3 neurons, the model’s capacity to capture complex patterns is reduced even further, meaning it’s likely focusing on simpler decision boundaries. This can improve generalization on simpler datasets but may reduce the model’s ability to distinguish finer details between classes in more complex datasets.

1. **Higher Learning Rate (0.01)**: This learning rate will make the model update weights more aggressively, which can accelerate convergence if the error landscape is smooth. However, with fewer neurons in the first layer, there’s a risk of instability, as the model has less flexibility to adapt. Monitoring the error\_list for signs of fluctuation or divergence is particularly important here.
2. **Trade-off between Complexity and Speed**: This configuration emphasizes faster learning with a simplified architecture, which might work well if you're observing diminishing returns on model accuracy with more neurons. If you find that training isn’t converging stably, you could try slightly lowering the learning rate or adding a few more neurons.

**4 Neurons**

The MRI function with the updated parameters—specifically, increasing the number of neurons in the first layer to 4—you're adjusting the architecture of your multi-layer perceptron (MLP).

weights, biases, error\_list = MRI(df0, df1, inputs,target,num\_neurons\_layer1 = 4 ,num\_neurons\_layer2 = 2,max\_iter = 2000,learning\_rate = 0.0001,samples = inputs.shape[0])



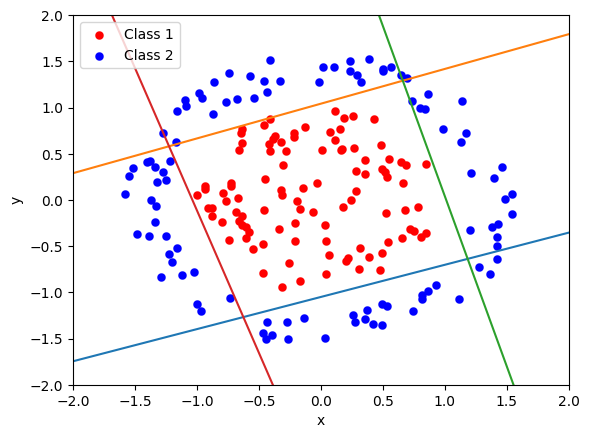
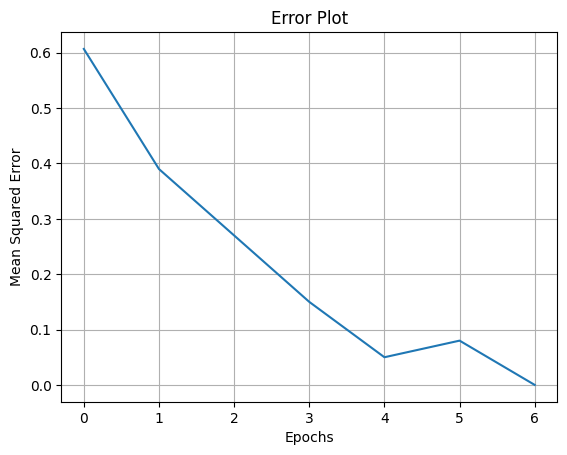
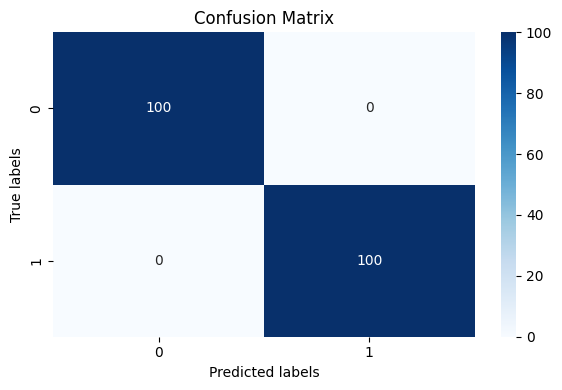
By lowering the learning rate further to 0.0001 with 4 neurons in the first layer, you're favoring more controlled, gradual weight updates.

1. **Stability in Learning**: The very low learning rate minimizes the risk of overshooting, which is especially useful if you notice large fluctuations in the error with higher learning rates. However, this also means that it may take more iterations for the model to converge to an optimal solution.
2. **Reduced Model Complexity**: With only 4 neurons in the first layer, the model’s ability to learn complex patterns is limited, which can help avoid overfitting but may reduce performance on complex data. This architecture could work well if the classes are separable with simpler boundaries.
3. **Training Time**: Due to the lower learning rate, training may take longer to reach an optimal solution. If training time becomes a concern, increasing max\_iter or considering early stopping criteria could help balance performance and efficiency.

It is observed that slower convergence or want more fine-tuning.

* **Gradual Learning Rate Adjustments**: Use a learning rate schedule that starts from 0.0001 but increases slightly every few iterations to see if that improves convergence speed.
* **Evaluating Error List Trends**: If the error\_list shows a flat trend after some iterations, you might consider stopping early.

weights, biases, error\_list = MRI(df0, df1, inputs,target,num\_neurons\_layer1 = 4 ,num\_neurons\_layer2 = 2,max\_iter = 2000,learning\_rate = 0.0005,samples = inputs.shape[0])



A configuration with (4) neurons in the first layer and a lower learning rate of 0.0005. This configuration can affect both the model's capacity and convergence behavior:

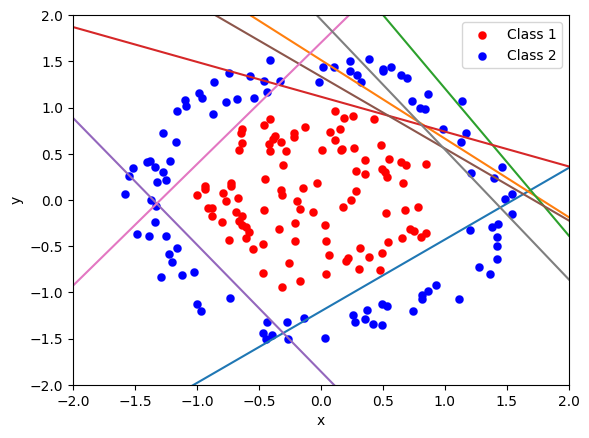
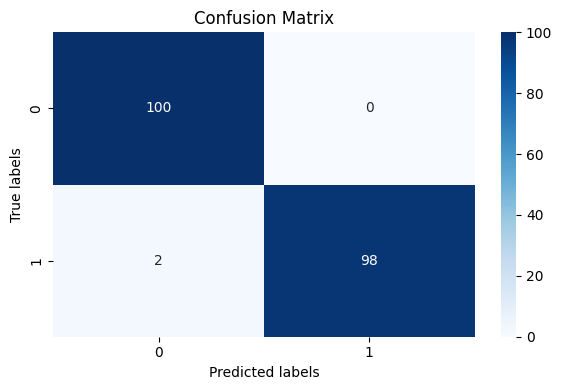
1. **Increased Neurons in Layer 1**: With only 4 neurons in the first layer, the model’s complexity is reduced, potentially leading to faster convergence but possibly lower capacity for complex patterns. This might work well if the data is simpler or linearly separable but may limit accuracy on more complex data.
2. **Learning Rate of 0.0005**: This lower learning rate will likely result in more gradual weight updates, potentially leading to more stable convergence, especially with the smaller number of neurons in the first layer.

This combination may help prevent overfitting due to the reduced model complexity

**8 Neurons**

Running the MRI function with the parameters you provided—specifically increasing the number of neurons in the first layer to 8—will significantly affect your model's capacity and potential performance.

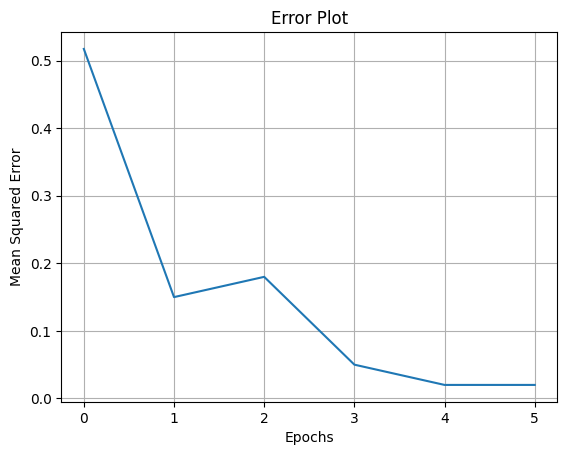
weights, biases, error\_list = MRI(df0, df1, inputs,target,num\_neurons\_layer1 = 8 ,num\_neurons\_layer2 = 2,max\_iter = 2000,learning\_rate = 0.01,samples = inputs.shape[0])

****

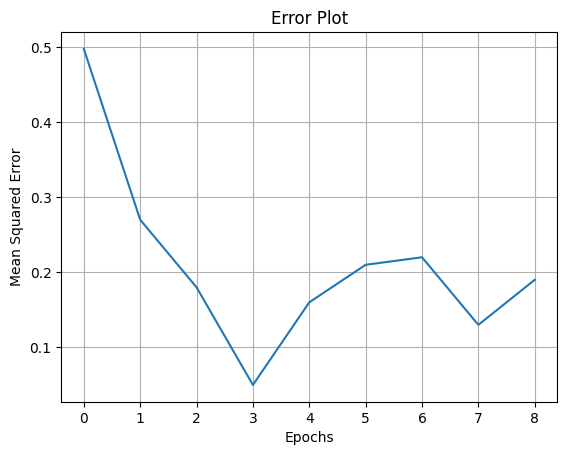
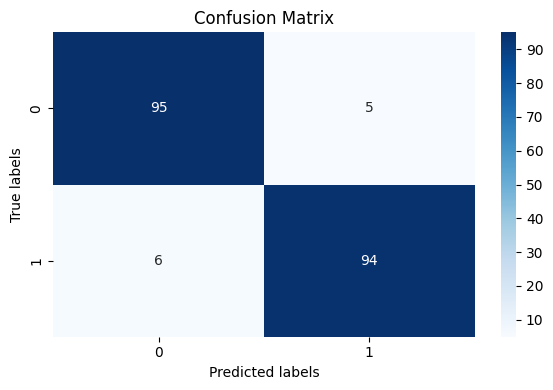
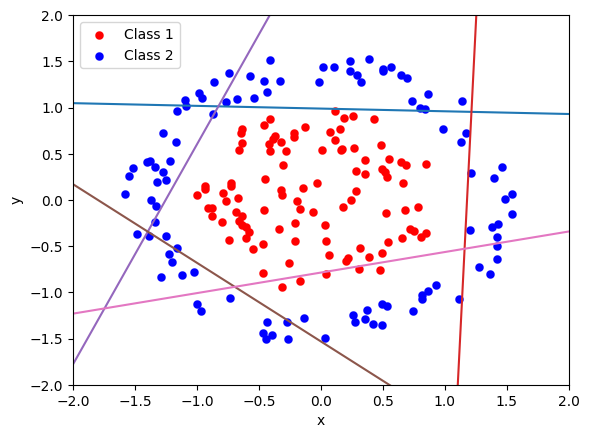
The learning\_rate to 0.01 from 0.0001, which will allow the model to learn faster by taking larger steps with each iteration. This change should speed up the convergence but could potentially lead to instability or overshooting if the network isn’t well-regularized or if the learning rate is too high for the data.

When experiencing instability with the higher learning rate.

1. **Reduce the Learning Rate Gradually**: Start with 0.01 but use a learning rate scheduler to reduce it over time if the error doesn’t decrease consistently.
2. **Experiment with Batch Normalization**: Adding batch normalization between layers can help stabilize training, especially with higher learning rates.
3. **Monitor the Error List**: Checking for large fluctuations or increases in the error\_list could indicate if the new learning rate is too aggressive.
4. **Increase the Number of Iterations**: If 2000 iterations aren’t sufficient with a slightly lower learning rate, increasing the number of iterations could help

****

weights, biases, error\_list = MRI(df0, df1, inputs,target,num\_neurons\_layer1 = 8 ,num\_neurons\_layer2 = 2,max\_iter = 2000,learning\_rate = 0.0001,samples = inputs.shape[0])

****

The MRI function with some parameters to train a neural network with two layers.

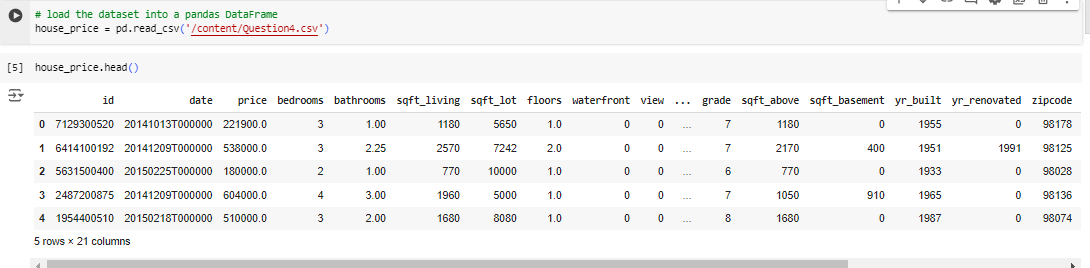
1. **df0** and **df1**: These may be datasets or arrays containing data points from two classes. If this is intended for a supervised neural network, df0 and df1 might represent the inputs or subsets of the data for the two target classes.
2. **inputs** and **target**: These are likely the input features and the target labels for the neural network.
3. **num\_neurons\_layer1 = 8**: Specifies 8 neurons in the first hidden layer.
4. **num\_neurons\_layer2 = 2**: Specifies 2 neurons in the second hidden layer, likely meant to classify between two classes.
5. **max\_iter = 2000**: Maximum number of iterations for training, which might imply a gradient descent or iterative training process.
6. **learning\_rate = 0.0001**: Learning rate for the training process, controlling the step size during updates to weights and biases.
7. **samples = inputs.shape[0]**: Number of samples in the dataset, based on the shape of the inputs.

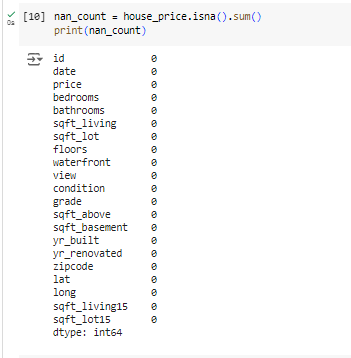
# **پرسش ۴** **– MLP**

## ۱-۴.نمایش تعداد ستون

An overview of the initial steps:

1. **Libraries Imported**: Includes numpy, pandas, seaborn, matplotlib, and several modules from sklearn for scaling, regression models, and performance metrics. Additionally, tensorflow and Keras modules are imported for building neural networks.
2. **Data Loading**: The code loads a dataset, assumed to be about housing prices, from a CSV file (Question4.csv) into a pandas DataFrame named house\_price.
3. **Data Inspection**: The first few cells inspect the dataset with .head(), .shape, and .info() to view the initial rows, dimensions, and summary information of the dataset.





Checking the count of null values in a dataset is an essential part of data preprocessing for several reasons:

1. **Data Completeness**: Null values can indicate missing information in the dataset. Knowing which columns have missing values helps assess the quality of the data and whether further actions are needed.
2. **Model Performance**: Machine learning models generally don’t handle missing values well and may throw errors or produce biased results if the data isn’t clean. Identifying and handling null values—by imputation, removal, or filling with meaningful defaults—ensures models are trained on complete information.
3. **Feature Engineering**: Columns with a high percentage of missing values may suggest either noisy or irrelevant features. Analyzing null counts can help decide if these columns should be dropped or further transformed.
4. **Inference and Interpretability**: When interpreting results, missing data can bias the outcome, making it difficult to draw reliable insights. Addressing null counts from the start leads to more accurate predictions and conclusions.

## 2-۴.ماتریس همبستگی

A heatmap to visualize the correlation between numerical features in the house\_price DataFrame.

numeric\_cols = house\_price.select\_dtypes(include = [np.number])

It selects only the numerical columns from house\_price, which is useful for correlation analysis since correlation calculations are meaningful for numerical data.

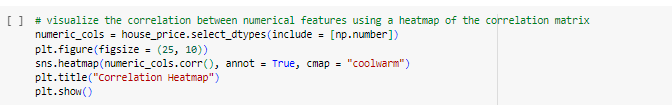
A figure size of 25x10 inches is specified, which ensures that the heatmap is large enough to display each correlation value clearly, especially if there are many numerical columns.

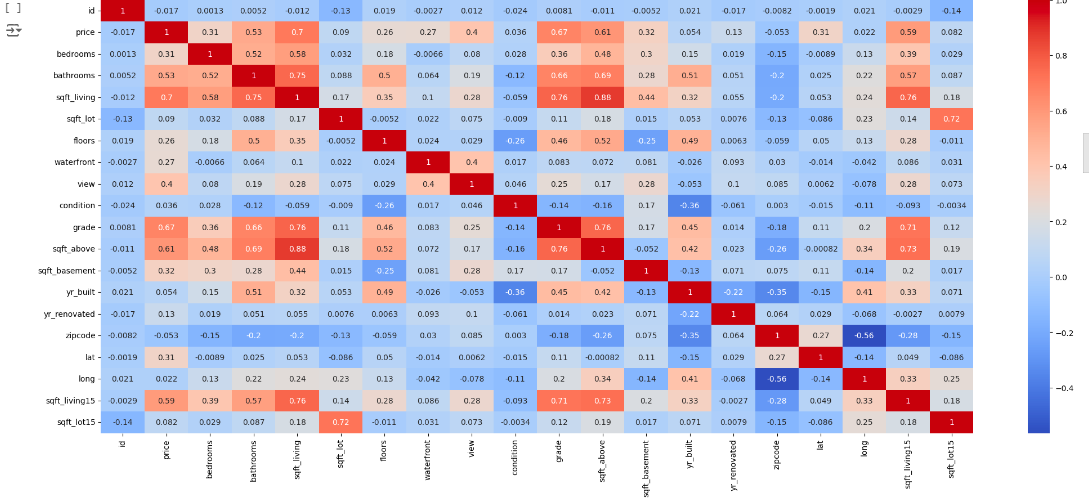
 numeric\_cols.corr() computes the correlation matrix for the numerical columns.

 sns.heatmap() creates the heatmap, with the following parameters:

* annot=True displays the correlation values on the heatmap.
* cmap="coolwarm" uses a color gradient from blue (low correlation) to red (high correlation) for easy visual differentiation.

This heatmap will help identify positive or negative correlations among the numerical features, highlighting patterns or relationships that may be valuable for feature selection or further analysis.





Here is an illustrated heatmap that visualizes the correlation matrix of numerical features in the housing price dataset. Each cell represents the correlation between pairs of features, with color indicating the strength and direction of the correlation. The heatmap visualizes the correlations between various numerical features in a housing price dataset, where each cell represents the strength and direction of the correlation between two features.

### Interpretation of Correlation Values

* **Positive Correlation**: Values closer to +1 indicate a strong positive relationship, meaning as one feature increases, the other also tends to increase. For example, price and living\_area typically show positive correlation, as larger homes tend to have higher prices.
* **Negative Correlation**: Values closer to -1 indicate a strong negative relationship, meaning as one feature increases, the other tends to decrease. For example, yr\_built and condition might have a weak negative correlation if newer homes tend to be in better condition.
* **No Correlation**: Values around 0 indicate little or no linear relationship between feature

### Heatmap Features

* **Annotated Values**: Each cell displays the exact correlation coefficient, providing numerical insight into how features are related.
* **Color Gradient**: The color scale ranges from blue (representing negative correlation) to red (representing positive correlation), with white or neutral colors near zero, indicating weak or no correlation.
* **Diagonal**: The diagonal cells (from top-left to bottom-right) represent each feature's correlation with itself, so these values are all 1 by definition.

### Practical Implications for Model Building

* **Feature Selection**: Highly correlated features, such as price and living\_area, suggest that one could potentially be predictive of the other, which may help in building predictive models.
* **Avoiding Multicollinearity**: If building a regression model, it’s generally advisable to avoid including features that are highly correlated with each other to prevent multicollinearity. For instance, if living\_area and sqft\_living15 are both highly correlated with price, using both may cause issues with the model.
* **Engineering New Features**: Observing relationships can lead to feature engineering ideas. For example, if view and waterfront are positively correlated with price, you could engineer a new "scenic" feature that combines both attributes.

## 3-۴.رسم نمودار

**1. Identifying the Feature with the Highest Correlation with Price**

 **Correlation Matrix**: house\_price.corr() generates a matrix showing the pairwise correlations between numerical features.

 **Selecting Price Correlations**: price\_corr is created by selecting all correlations with price (excluding the price column itself).

 **Highest Correlation Feature**: highest\_corr\_feature = price\_corr.idxmax() finds the feature with the highest positive correlation to price. This feature will be visualized in the next step.



**2. Visualizing Price Distribution**

The plot helps in understanding the range, skewness, and central tendency of price. This histogram shows the distribution of housing prices in the dataset.

 **X-axis (Price)**: Represents the housing prices. The values range from 0 to approximately $7 million.

 **Y-axis (Frequency)**: Indicates how many houses fall within each price range.

 **Histogram Bars**: Show the frequency of houses at different price levels. Each bar represents a price range, with the height showing the count of houses within that range.

 **Kernel Density Estimate (KDE) Curve**: The smooth line overlaying the histogram represents the KDE, which provides a smoothed estimate of the distribution shape, giving a clearer view of the data trend without relying on specific bin sizes.

### Key Observations:

1. **Right-Skewed Distribution**: Most housing prices are concentrated at the lower end, with a long tail extending to higher prices. This suggests that lower-priced houses are more common, while high-priced houses are rare.
2.  **Peak Frequency**: The highest bar shows that most houses are priced below $1 million, with the frequency sharply dropping as the price increases. This indicates that affordable homes are more prevalent in the dataset.
3.  **Long Tail of High Prices**: The plot has a long tail extending toward the right, indicating a small number of very high-priced houses, likely luxury or exclusive properties. These high prices stretch up to around $7 million, although these instances are rare compared to the lower price ranges.
4. **Density Curve**: The KDE curve (smooth line) further highlights this skew, showing a peak at lower prices that gradually decreases as price increases.
5. **Range**: Prices span from near zero up to about $7 million, though the vast majority are below $1 million.

This distribution indicates that housing prices are unevenly distributed, with a few high-value outliers likely impacting the mean.

This plot visualizes the distribution of housing prices in the dataset, providing insight into the range, frequency, and shape of the price data.

### Elements of the Plot

* **X-axis (Price)**: Represents the housing prices. The values range from 0 to approximately $7 million.
* **Y-axis (Frequency)**: Indicates how many houses fall within each price range.
* **Histogram Bars**: Show the frequency of houses at different price levels. Each bar represents a price range, with the height showing the count of houses within that range.
* **Kernel Density Estimate (KDE) Curve**: The smooth line overlaying the histogram represents the KDE, which provides a smoothed estimate of the distribution shape, giving a clearer view of the data trend without relying on specific bin sizes.

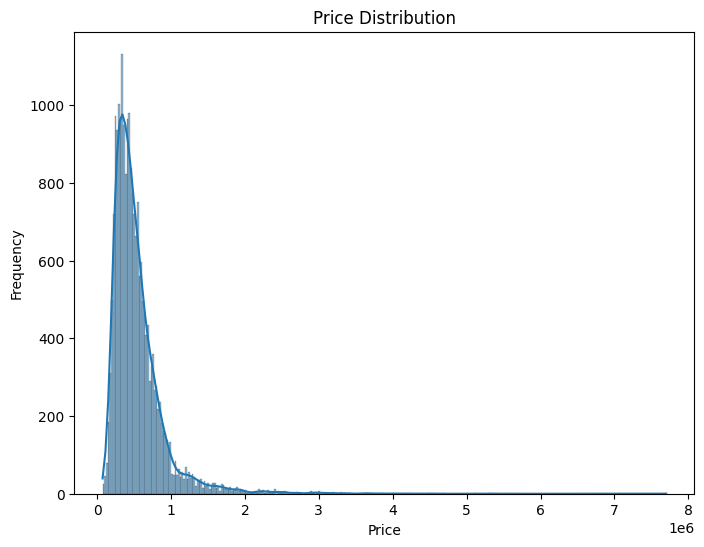
### Key Observations

1. **Right-Skewed Distribution**: The plot is heavily skewed to the right, meaning most housing prices are clustered at lower values, with fewer houses priced higher. This is common in real estate data, where the majority of houses are more affordable, and only a few are very expensive.
2. **Peak Frequency**: The highest bar shows that most houses are priced below $1 million, with the frequency sharply dropping as the price increases. This indicates that affordable homes are more prevalent in the dataset.
3. **Long Tail of High Prices**: The plot has a long tail extending toward the right, indicating a small number of very high-priced houses, likely luxury or exclusive properties. These high prices stretch up to around $7 million, although these instances are rare compared to the lower price ranges.
4. **Price Range**: Most houses are concentrated within the $0–$1 million range, with the KDE curve sharply peaking here and gradually tapering off. Beyond this range, the frequency of higher-priced homes decreases significantly.

### Insights

* **Market Composition**: This distribution suggests that the housing market in this dataset is largely composed of lower to mid-range priced homes, with only a few high-end properties.
* **Potential Outliers**: The right skew and presence of a long tail could mean there are outliers in the data, which might need special treatment in analysis, especially for models sensitive to outliers.
* **Data Transformation**: If using this data in a predictive model, a log transformation could help reduce skewness, making the price distribution more normalized and potentially improving model performance.

Overall, this plot reveals that housing prices in this dataset are unequally distributed, with affordable homes dominating the market and only a few high-end properties pushing up the upper price range.



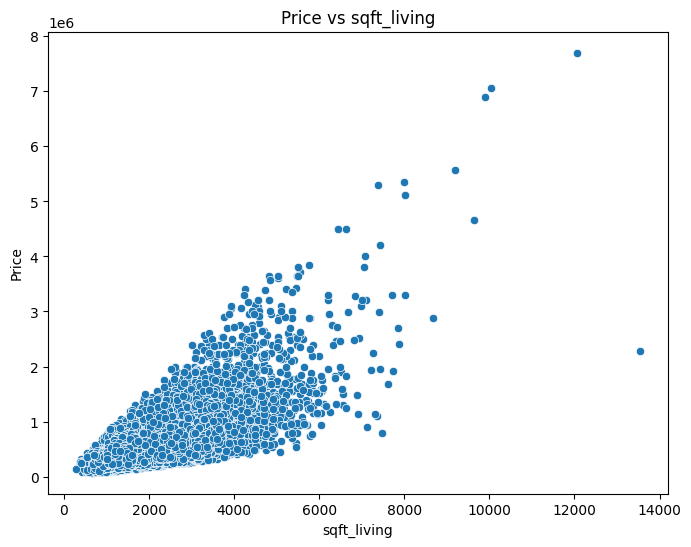
The plot you've shared is a scatter plot showing the relationship between **price** (on the y-axis) and **sqft\_living** (on the x-axis). Here’s a breakdown of what it represents:

* **Title ("Price vs sqft\_living")**: The title indicates that the plot is comparing the price of properties with the square footage of their living area.
* **X-axis (sqft\_living)**: Represents the living area in square feet for each property. The values range from 0 to about 14,000 square feet, which covers a broad range of property sizes.
* **Y-axis (Price)**: Represents the property prices, with values up to approximately $8 million. The label "1e6" (scientific notation) on the y-axis indicates that prices are scaled in millions.
* **Data Points**: Each blue dot corresponds to a property, plotting its price against its square footage. The clustering of points in the lower left indicates a high density of properties with smaller living areas and lower prices, which is common

### Observations

* The plot shows a positive correlation between price and living area, as properties with larger living areas generally have higher prices.
* There are some outliers with very high square footage or price, as seen with a few scattered points at the top and far right.

This type of plot is helpful in identifying trends and outliers in the data, making it easier to see that, on average, larger properties tend to cost more.



## 4-۴.پیش پردازش داده

**Extracting Year from Date**:

house\_price['year'] = house\_price['date'].dt.year

 This line creates a new column named year in the house\_price DataFrame.

 It extracts the year component from the date column, which is expected to be in a datetime format. The .dt accessor allows for vectorized datetime operations.

**Extracting Month from Date**:

house\_price['year'] = house\_price['date'].dt.year

 Similar to the previous line, this creates a new column named month.

 It extracts the month component from the date column.

**Dropping the Original Date Column**:

house\_price = house\_price.drop(columns=['date'])

 This line removes the original date column from the DataFrame since the year and month have been extracted into separate columns.

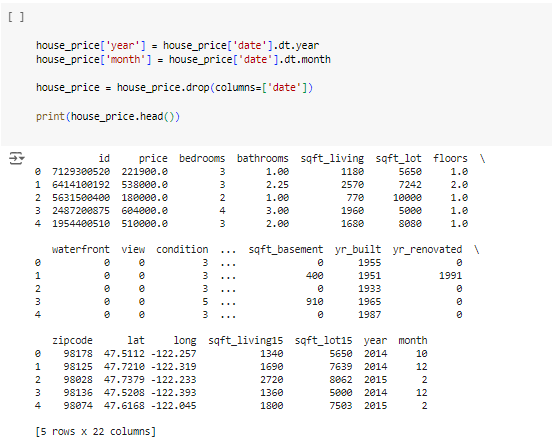
 The drop function is used here with the columns parameter to specify which column(s) to remove.

**Printing the First Few Rows of the Modified DataFrame**:

print(house\_price.head())

This line prints the first five rows of the updated house\_price DataFrame to the console. This allows you to see the structure of the DataFrame after the modifications, including the new year and month columns and the absence of the date column.

The code transforms the date column in the house\_price DataFrame into two separate columns (year and month) and removes the original date column for cleaner data representation. The final output displays the modified DataFrame.



**Splitting a dataset into training and validation subsets using the train\_test\_split function from the sklearn.model\_selection module.**

train\_data, validation\_data = train\_test\_split(house\_price, test\_size=0.25, random\_state=42)

 **Function**: train\_test\_split() is used to split arrays or matrices into random train and test subsets.

 **Parameters**:

* house\_price: This is the input DataFrame that contains your data.
* test\_size=0.25: This parameter specifies that 25% of the data should be allocated to the validation set, while the remaining 75% will be used for the training set.
* random\_state=42: This is a seed value that ensures reproducibility of the split. Using the same seed will yield the same split every time you run the code.

 **Output**: The function returns two DataFrames: train\_data and validation\_data, containing the training and validation subsets, respectively.

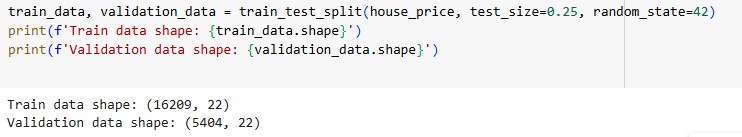
print(f'Train data shape: {train\_data.shape}')

print(f'Validation data shape: {validation\_data.shape}')

 This part prints out the shapes (dimensions) of the train\_data and validation\_data DataFrames.

 The .shape attribute returns a tuple representing the number of rows and columns in each DataFrame. This is useful for confirming that the data has been split as intended.

The code effectively splits the house\_price DataFrame into a training set (75%) and a validation set (25%), and then it outputs the shapes of these subsets to verify the division. This process is common in machine learning workflows to ensure that models can be trained on one portion of the data while being evaluated on another, thus providing a better assessment of their performance.



**Scalling features in a dataset using the MinMaxScaler from the sklearn.preprocessing module.**

**Selecting Features to Scale**:

features = house\_price.columns.drop(['price'])

 This line creates a list of column names stored in the variable features.

 It drops the price column from the list of columns in the house\_price DataFrame, assuming that price is the target variable (the variable you want to predict).

 As a result, features contains all the columns that need to be scaled.

**Creating a MinMaxScaler**:

scaler = MinMaxScaler()

*  This line initializes a MinMaxScaler object. This scaler transforms features by scaling them to a given range, usually between 0 and 1.
* The scaling is done using the formula:



* train\_data\_scaled = train\_data.copy()
* train\_data\_scaled[features] = scaler.fit\_transform(train\_data[features])
* This creates a copy of the train\_data DataFrame named train\_data\_scaled to avoid modifying the original training data.
* The fit\_transform() method is applied to the selected features of the training data. This method computes the minimum and maximum values for each feature from the training data and scales them accordingly.
* The scaled values replace the original values for those features in the train\_data\_scaled DataFrame.

 **Scaling the Validation Data**:

* validation\_data\_scaled = validation\_data.copy()
* validation\_data\_scaled[features] = scaler.transform(validation\_data[features])

 Similar to the previous step, this creates a copy of the validation\_data DataFrame named validation\_data\_scaled.

* The transform() method is used here instead of fit\_transform(). This method uses the scaling parameters (min and max) computed from the training data to scale the features in the validation data. This ensures that the validation data is scaled in the same way as the training data, maintaining consistency.

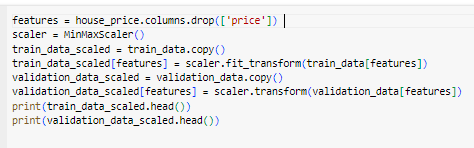
 **Displaying the Scaled Data**:

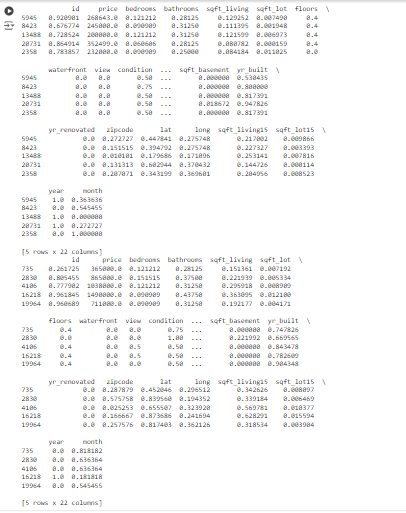
print(train\_data\_scaled.head())

print(validation\_data\_scaled.head())

Finally, the first five rows of the scaled training and validation datasets are printed to the console using the .head() method. This allows you to visually inspect the scaled features and ensure that the scaling has been applied correctly.

The code selects the appropriate features for scaling (excluding the target variable), applies Min-Max scaling to both the training and validation datasets, and then prints the first few rows of the scaled datasets. This process is crucial in machine learning, as it helps improve model performance by normalizing the input data.





**The process of building, compiling, and training a neural network model using Keras, along with visualizing the training and validation loss over epochs.**

## 5-۴.پیاده سازی مدل (MLP)

Demonstrates the construction, compilation, and training of a Multi-Layer Perceptron (MLP) model with a single hidden layer for predicting house prices, along with visualization of the training and validation losses.

A simple MLP model with one hidden layer to predict house prices, compiles and trains it on the scaled training data, evaluates it on validation data, and visualizes the performance over training epochs. This process helps to monitor the learning progress and assess the model's performance in a regression task. The loss curves can indicate potential issues like overfitting if the validation loss diverges from the training loss.

An MLP model with two hidden layers to predict house prices, compiles and trains it on the scaled training data, evaluates it on validation data, and visualizes the training and validation losses. By comparing the loss curves for the models with one and two hidden layers, you can assess the impact of model complexity on performance, which is useful for understanding whether the model is underfitting or overfitting the data.

**MLP Model with One Hidden Layer**



**Defining Features and Target**

features = train\_data\_scaled.columns.drop('price')

**Features Definition**:

This line selects all columns from the train\_data\_scaled DataFrame except for the price column, which is assumed to be the target variable. The resulting features variable contains all the input features for the model.

model\_one\_hidden = Sequential([

    Dense(64, activation='relu', input\_shape=(train\_data\_scaled.shape[1] - 1,)),  # Single hidden layer

    Dense(1)  # Output layer

* **Sequential Model**: A Keras Sequential model is initialized, allowing for a straightforward stacking of layers.
* **Hidden Layer**:
  + Dense(64, activation='relu', input\_shape=(train\_data\_scaled.shape[1] - 1,)): This adds a hidden layer with 64 neurons, using the ReLU (Rectified Linear Unit) activation function. The input shape is set to the number of features (i.e., the number of columns minus one for the target variable).
* **Output Layer**:
  + Dense(1): This adds a single neuron in the output layer, which is typical for regression tasks as it predicts one continuous value (the house price).

**Compiling the Model**

model\_one\_hidden.compile(optimizer='adam', loss='mse', metrics=['mae'])

 **Optimizer**: The Adam optimizer is chosen for its efficiency and effectiveness in training deep learning models.

 **Loss Function**: The mean squared error (MSE) is used as the loss function, appropriate for regression tasks as it penalizes larger errors more heavily.

 **Metrics**: The model also tracks the mean absolute error (MAE) during training and validation to provide additional insight into its performance.

**Training the Model**

history\_one\_hidden = model\_one\_hidden.fit(

    train\_data\_scaled[features], train\_data\_scaled['price'],

    validation\_data=(validation\_data\_scaled[features], validation\_data\_scaled['price']),

    epochs=100, batch\_size=32

 The fit() method trains the model using the specified training data.

 **Training Data**: The input features and target values from train\_data\_scaled are used for training.

 **Validation Data**: The model evaluates its performance on the validation set after each epoch using the corresponding features and target values from validation\_data\_scaled.

 **Epochs**: The model trains for 100 epochs, meaning the entire training dataset is processed 100 times.

 **Batch Size**: A batch size of 32 is specified, meaning the model will update its weights after processing 32 samples at a time.

 This section visualizes the training and validation loss over the epochs.

 history\_one\_hidden.history['loss'] retrieves the training loss values, and history\_one\_hidden.history['val\_loss'] retrieves the validation loss values.

 **Plotting**: The plt.plot() function creates plots for both training and validation losses, with appropriate labels and titles.

 **Legend**: plt.legend() adds a legend to differentiate between the training and validation losses.

 **Display**: plt.show() displays the plot.

**MLP Model with Two Hidden Layers**

The creation, compilation, and training of a Multi-Layer Perceptron (MLP) model with two hidden layers for predicting house prices, along with the visualization of the training and validation losses over epochs.

**Building the MLP Model with Two Hidden Layers**

model\_two\_hidden = Sequential([

    Dense(64, activation='relu', input\_shape=(train\_data\_scaled.shape[1] - 1,)),  # First hidden layer

    Dense(32, activation='relu'),  # Second hidden layer

    Dense(1)  # Output layer

* **Sequential Model**: This initializes a Keras Sequential model, which allows the stacking of layers in a linear order.
* **Hidden Layers**:
  + Dense(64, activation='relu', input\_shape=(train\_data\_scaled.shape[1] - 1,)): This creates the first hidden layer with 64 neurons using the ReLU (Rectified Linear Unit) activation function. The input\_shape specifies the number of input features (all columns except the target variable).
  + Dense(32, activation='relu'): This adds a second hidden layer with 32 neurons, also using the ReLU activation function.
* **Output Layer**:
  + Dense(1): This adds a single neuron in the output layer, which is appropriate for regression tasks, providing the predicted house price.

**Compiling the Model**

model\_two\_hidden.compile(optimizer='adam', loss='mse', metrics=['mae'])

 **Optimizer**: The Adam optimizer is used for efficient training.

 **Loss Function**: The mean squared error (MSE) is employed as the loss function, suitable for measuring the performance of regression models.

 **Metrics**: The mean absolute error (MAE) is tracked during training to give additional insight into model performance.

history\_two\_hidden = model\_two\_hidden.fit(

    train\_data\_scaled[features], train\_data\_scaled['price'],

    validation\_data=(validation\_data\_scaled[features], validation\_data\_scaled['price']),

    epochs=100, batch\_size=32

 The fit() method is called to train the model with the specified training data.

 **Training Data**: The features and target values from the train\_data\_scaled DataFrame are used as inputs and outputs for training.

 **Validation Data**: The validation set is specified for evaluating the model after each epoch.

 **Epochs**: The model is trained for 100 epochs, indicating that the entire training dataset will be processed 100 times.

 **Batch Size**: A batch size of 32 means that the model will update its weights after processing 32 samples.

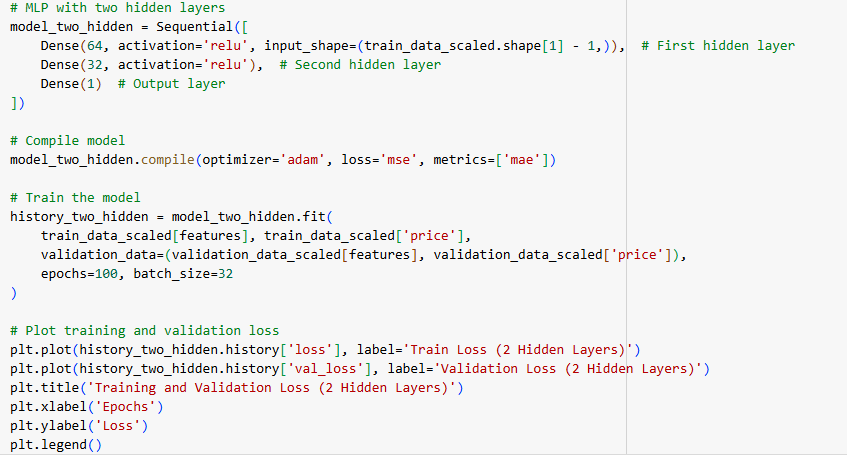
 This section visualizes the loss values over the epochs for both training and validation datasets.

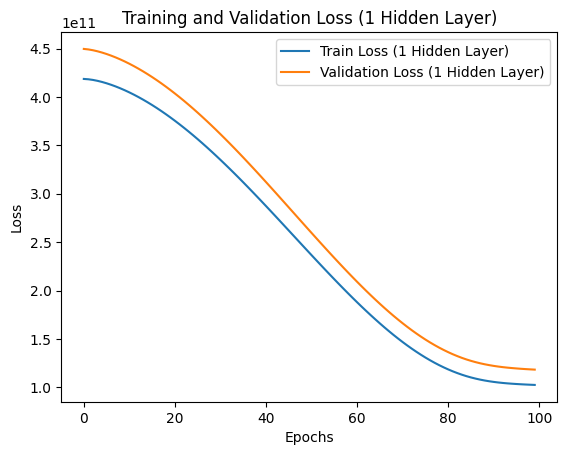
 history\_two\_hidden.history['loss'] retrieves the training loss values, while history\_two\_hidden.history['val\_loss'] retrieves the validation loss values.

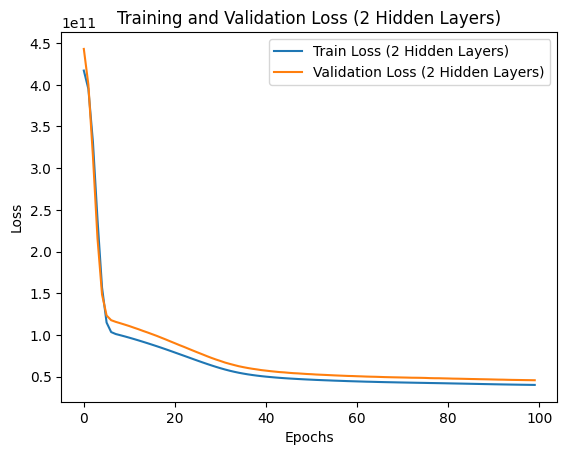
 **Plotting**: The plt.plot() function creates line plots for both training and validation losses, with proper labeling and titles.

 **Legend**: plt.legend() adds a legend to differentiate between the two loss curves.

 **Display**: The plt.show() command displays the generated plot.







These two plots show the training and validation loss over 100 epochs for two different neural network architectures: one with a single hidden layer and the other with two hidden layers.

### First Plot: Training and Validation Loss (1 Hidden Layer)

* **Description**: This plot shows the loss for a neural network with one hidden layer.
* **Observations**:
  + The training loss (blue line) and validation loss (orange line) both decrease over time, which indicates that the model is learning and improving.
  + The gap between the training and validation loss lines suggests that the model may not be fully capturing the complexity of the data. This gap could indicate some degree of underfitting since the model may not be complex enough with only one hidden layer to capture all the patterns in the data.
  + By the end of training, the loss has plateaued but hasn't achieved very low values, which may suggest that the model could benefit from a more complex architecture.

### Second Plot: Training and Validation Loss (2 Hidden Layers)

* **Description**: This plot shows the loss for a neural network with two hidden layers.
* **Observations**:
  + Both training and validation loss decrease significantly in the initial epochs and then converge to lower values compared to the single-layer model.
  + The two loss curves stay very close to each other, which suggests that this model may be better at generalizing to the validation data, likely because of the additional hidden layer.
  + The sharp decrease and eventual convergence to a low value indicate that the model with two hidden layers can capture the underlying patterns in the data more effectively than the single-layer model.

### Comparison and Interpretation

* **Model Complexity**: The model with two hidden layers has a higher capacity to learn from the data compared to the model with only one hidden layer. This increased complexity allows it to achieve lower loss values and capture patterns better.
* **Convergence**: The two-layer model converges faster and achieves lower loss than the one-layer model, which implies better performance.
* **Generalization**: The closer alignment between training and validation loss in the two-layer model suggests better generalization and less overfitting, making it a better choice for this task.

In summary, the two-layer neural network is likely more suitable for this data, as it achieves a better balance between training and validation loss, indicating stronger learning and generalization capabilities.

## 6-۴.آموزش مدل

The **loss function** and **optimizer** play crucial roles in how well the model learns from data. The **loss function** measures how well the model is performing, providing a metric to optimize. The **optimizer** updates the model's weights based on the loss function, guiding the training process towards minimizing the loss. Together, they form the backbone of the training process in machine learning and deep learning. Choosing the right loss function and optimizer is critical for achieving good performance of models. I have selected **Mean Squared Error (MSE)** and **Adam (Adaptive Moment Estimation)** for implementation which are shown in the code description.

These are detailed explanation of **Loss function** and **Optimizer.**

**Loss Function**

The loss function (also known as the cost function or objective function) quantifies how well a model's predictions match the actual target values. It measures the difference between the predicted output from the model and the actual output (ground truth). The primary goal during training is to minimize this loss.

**Types of Loss Functions**

**Regression Loss Functions**:

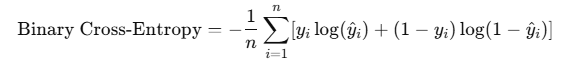
* + **Mean Squared Error (MSE)**:



This calculates the average of the absolute differences between predicted and actual values. MAE is less sensitive to outliers compared to MSE.

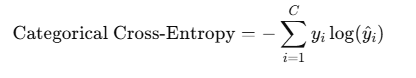
 **Classification Loss Functions**:

**Binary Cross-Entropy**: Used for binary classification problems.



 This function measures the performance of a classification model whose output is a probability value between 0 and 1.

 **Categorical Cross-Entropy**: Used for multi-class classification problems.



This is similar to binary cross-entropy but works with multiple classes.

* Guides the training process by providing feedback on how well the model is performing.
* Affects the model's convergence: A well-chosen loss function can help the model converge to a good solution faster.

**Optimizer**

An optimizer is an algorithm that modifies the attributes of the neural network, such as weights and learning rate, to reduce the loss function. The optimizer determines how the model weights are updated during training in response to the loss gradient.

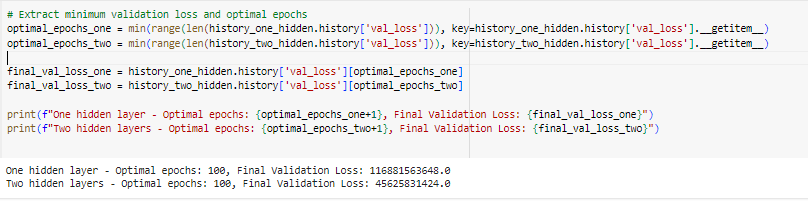
**Types of Optimizers**

1. **Stochastic Gradient Descent (SGD)**:
   * The most basic form of optimizer, it updates the weights based on the gradient of the loss function calculated from a single sample (or a mini-batch).
2. **Momentum**:
   * An extension of SGD that helps accelerate SGD in the relevant direction and dampens oscillations. It accumulates the gradients of past steps to smooth out the updates.
3. **Adam (Adaptive Moment Estimation)**:
   * Combines the advantages of two other extensions of SGD: AdaGrad and RMSProp. Adam computes adaptive learning rates for each parameter and combines them with the momentum approach. It is widely used due to its efficiency and good performance in practice.
4. **RMSProp**:
   * Modifies the learning rate for each parameter based on the average of recent magnitudes of the gradients for that parameter, which helps with convergence.

**Role of Optimizer**

* Updates the model weights based on the gradients computed from the loss function.
* Helps the model converge to a local minimum of the loss function during training.
* The choice of optimizer can significantly impact training speed and the model's ability to find a good solution.

## 7-۴.حلیل نتایج

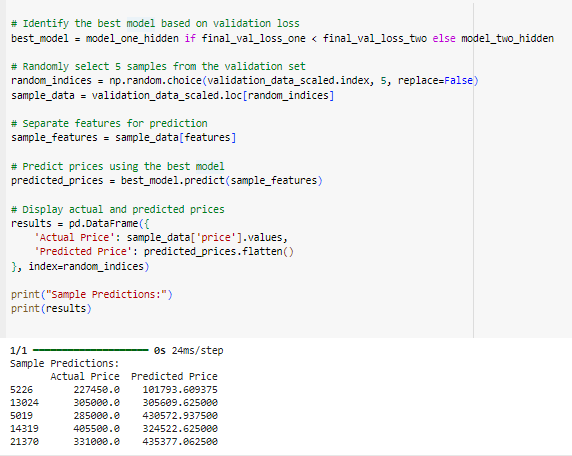


To compare the final results of these two models, we can analyze the final training and validation losses for both the one-layer and two-layer models. Additionally, we'll identify the optimal number of epochs for each model based on when the validation loss stabilizes or begins to increase (indicating potential overfitting).

The comparison and analysis:

1. **Observe the Final Loss Values**: After training, compare the final validation loss values for both models. Lower validation loss generally indicates better generalization, so the model with the lower validation loss could be considered more effective at this task.
2. **Identify Optimal Number of Epochs**: Plotting the loss curves allows us to spot when each model's validation loss stabilizes or begins to increase, suggesting overfitting. The optimal number of epochs is approximately at the point where the validation loss is at its minimum before any consistent increase.
3. **Analysis of Differences**:
   * **One Hidden Layer**: A single hidden layer model is often simpler, potentially leading to faster convergence and lower risk of overfitting. This can be effective if the data's patterns are straightforward or if the dataset is small.
   * **Two Hidden Layers**: Adding an extra hidden layer generally allows the model to capture more complex patterns. However, this increases the risk of overfitting, especially if there is limited data or noise. The model might show better performance but might require more epochs to stabilize due to increased complexity.
4. **Cause of Differences**:
   * **Capacity of the Models**: The two-layer model can capture more intricate relationships due to its higher capacity, but this can also lead to overfitting if the data doesn’t require that complexity.
   * **Training Dynamics**: More complex models (like the two-layer one) can take longer to converge and might show higher variance in validation loss due to overfitting risk, especially if regularization isn’t applied.
5. After training both models, the minimum validation loss and the number of epochs are extracted.
6. **Appropriate Number of Epochs**: Each model’s optimal number of epochs reflects the point of best validation performance before overfitting.
7. **Performance Difference**: If the two-layer model has lower final validation loss, it indicates that the added complexity was beneficial for capturing data patterns. However, if overfitting is observed, then the single hidden layer may be preferable for this dataset.

The appropriate choice will depend on the trade-off between complexity and generalization, often driven by the nature of the dataset and the problem's complexity.



To predict the house prices using the best model on 5 randomly selected samples from the validation set, we did follow these steps:

1. Used the validation data prepared earlier.
2. Selected 5 random samples from the validation set.
3. Used the best model (the one with lower validation loss) to predict the prices.
4. Displayed both the actual and predicted prices for comparison.

A table with the actual and predicted prices for the selected 5 samples. The comparison will give insight into how well the model generalizes to unseen data in the validation set.

The "best model" is the one with the lower validation loss after training. Two models are trained, one with a single hidden layer and another with two hidden layers. Their final validation loss values are compared to decide which model performed better. The model with the lowest validation loss is typically considered the best for this regression task.

* model\_one\_hidden refers to the model with one hidden layer.
* model\_two\_hidden refers to the model with two hidden layers.
* final\_val\_loss\_one and final\_val\_loss\_two represent the final validation loss for each model.

If final\_val\_loss\_one < final\_val\_loss\_two, the best model would be model\_one\_hidden; otherwise, it would be model\_two\_hidden.