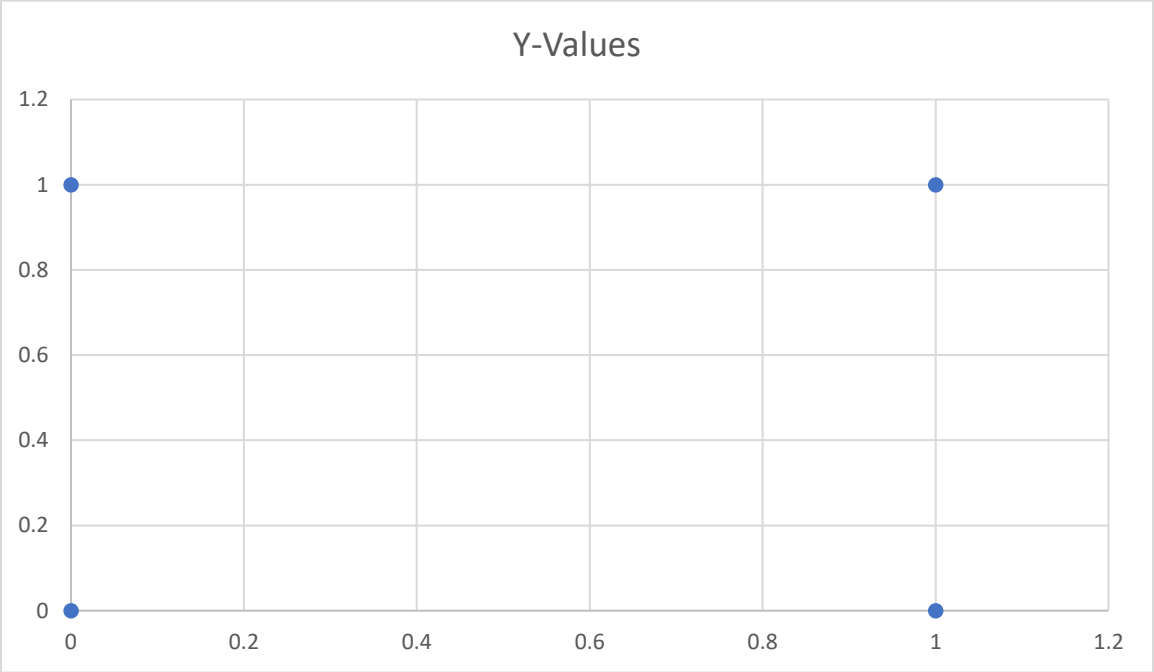
## سوال اول

در فرض سوال، این طراحی امکان پذیر نیست زیرا یک نورون مصنوعی تنها قادر به حل مسائلی است که می‌توان با استفاده از یک خط، داده‌ها را از هم جدا کرد درحالیکه درصورت رسم نمودار داده ها، ، امکان جداسازی داده‌ها با استفاده از یک خط وجود ندارد. برای حل ان مشکل، از یک Madaline با دو نورون داخلی و یک نورون OR استفاده می‌شود.




در نمودار بالا، داده‌های (۰،۰) و (۱،۱) متعلق به یک کلاس و داده‌های دیگر متعلق به کلاس دیگری است که واضح است امکان جداسازی با یک خط وجود ندارد. راه حل، استفاده از دو خط (به عبارتی، دو نورون) می‌باشد.

**سوال دوم**

خروجی گیت AND به صورت مقابل است:

| $x_1$ | $x_2$ | o |
|---|---|---|
| 1 | 1 | 1 |
| 1 | -1 | -1 |
| -1 | 1 | -1 |
| -1 | -1 | -1 |

ورودی‌های شبکه به صورت زیر خواهد بود:

| $x_1$ | $x_2$ | b | o |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 1 | -1 | 1 | -1 |
| -1 | 1 | 1 | -1 |
| -1 | -1 | 1 | -1 |

در ابتدا روش بروزرسانی وزن‌ها بصورت GD و با تابع فعالسازی پله خواهد بود. در مدل Adaline با تابع فعالسازی پله، آموزش بدون احتساب تابع فعالسازی خواهد بود.

حالت کلی فرمول‌ها (برای GD) به صورت زیر است:

$$y = x_1 w_1 + x_2 w_2 + b$$

$$b = b + lr * \sum(o - y) = b + lr * \left(\sum o - \sum y\right)$$

$$w_1 = w_1 + lr * \sum\left((o - y) * x_i\right)$$

$$w_2 = w_2 + lr * \sum\left((o - y) * x_i\right)$$

حالت کلی فرمول‌ها (برای SGD) به صورت زیر است:

$$y = x_1 w_1 + x_2 w_2 + b$$

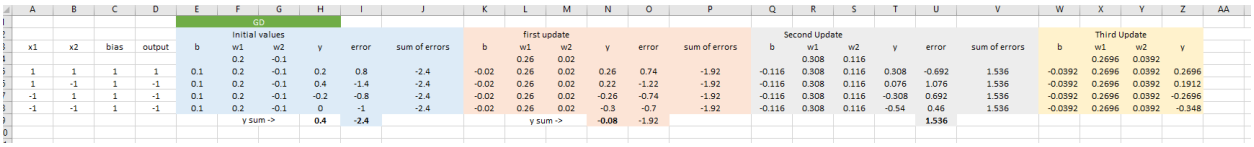
$$b = b + lr * (o_i - y_i)$$

$$w_1 = w_1 + lr * (o - y) * x_1$$

$$w_2 = w_2 + lr * (o - y) * x_2$$

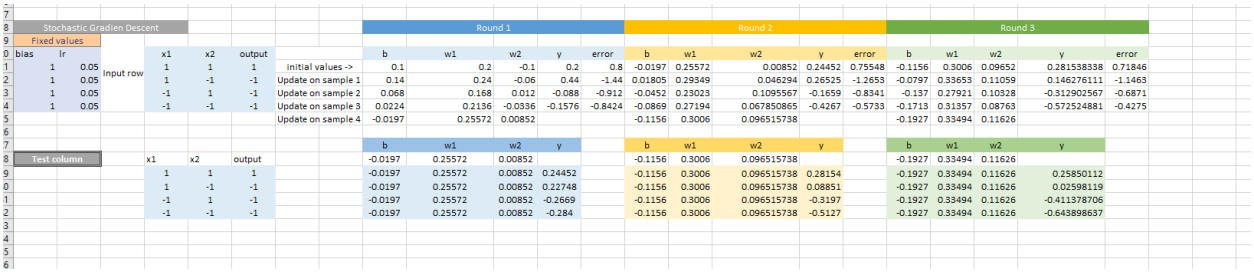
بدلیل زیاد بودن و پیچیدگی محاسبات و برای جلوگیری از اشتباهات محاسباتی، محاسبات در فایل اکسلی انجام شده است که در پوشه زیپ قرار دارد.

بخش زیر در فایل اکسل، مربوط به آپدیت وزن‌ها با استفاده از روش GD می‌باشد.



| | | | | GD | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Initial values | | | | | | first update | | | | | | Second Update | | | | | | Third Update | | | | | | | | |
| x1 | x2 | bias | output | b | w1 | w2 | y | error | sum of errors | b | w1 | w2 | y | error | sum of errors | b | w1 | w2 | y | error | sum of errors | b | w1 | w2 | y | | | | | |
| | | | | | 0.2 | -0.1 | | | | | 0.26 | 0.02 | | | | | 0.308 | 0.116 | | | | | 0.2696 | 0.0392 | | | | | | |
| 1 | 1 | 1 | 1 | 0.1 | 0.2 | -0.1 | 0.2 | 0.8 | -2.4 | -0.02 | 0.26 | 0.02 | 0.26 | 0.74 | -1.92 | -0.116 | 0.308 | 0.116 | 0.308 | -0.692 | 1.536 | -0.0392 | 0.2696 | 0.0392 | 0.2696 | | | | | |
| 1 | -1 | 1 | -1 | 0.1 | 0.2 | -0.1 | 0.4 | -1.4 | -2.4 | -0.02 | 0.26 | 0.02 | 0.22 | -1.22 | -1.92 | -0.116 | 0.308 | 0.116 | 0.076 | 1.076 | 1.536 | -0.0392 | 0.2696 | 0.0392 | 0.1912 | | | | | |
| -1 | 1 | 1 | -1 | 0.1 | 0.2 | -0.1 | -0.2 | -0.8 | -2.4 | -0.02 | 0.26 | 0.02 | -0.26 | -0.74 | -1.92 | -0.116 | 0.308 | 0.116 | -0.308 | 0.692 | 1.536 | -0.0392 | 0.2696 | 0.0392 | -0.2696 | | | | | |
| -1 | -1 | 1 | -1 | 0.1 | 0.2 | -0.1 | 0 | -1 | -2.4 | -0.02 | 0.26 | 0.02 | -0.3 | -0.7 | -1.92 | -0.116 | 0.308 | 0.116 | -0.54 | 0.46 | 1.536 | -0.0392 | 0.2696 | 0.0392 | -0.348 | | | | | |
| | | | | y sum -> | | | 0.4 | -2.4 | | y sum -> | | | -0.08 | -1.92 | | | | | | 1.536 | | | | | | | | | | |

بخش زیر نیز به روش SGD اختصاص دارد.



| Stochastic Gradien Descent | | | | | | | | Round 1 | | | | | Round 2 | | | | | Round 3 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Fixed values | | | | | | | | | | | | | | | | | | | | | | |
| bias | lr | | x1 | x2 | output | | | b | w1 | w2 | y | error | b | w1 | w2 | y | error | b | w1 | w2 | y | error |
| 1 | 0.05 | | 1 | 1 | 1 | | initial values -> | 0.1 | 0.2 | -0.1 | 0.2 | 0.8 | -0.0197 | 0.25572 | 0.00852 | 0.24452 | 0.75548 | -0.1156 | 0.3006 | 0.09652 | 0.281538338 | 0.71846 |
| 1 | 0.05 | input row | 1 | -1 | -1 | | Update on sample 1 | 0.14 | 0.24 | -0.06 | 0.44 | -1.44 | 0.01805 | 0.29349 | 0.046294 | 0.26525 | -1.2653 | -0.0797 | 0.33653 | 0.11059 | 0.146276111 | -1.1463 |
| 1 | 0.05 | | -1 | 1 | -1 | | Update on sample 2 | 0.068 | 0.168 | 0.012 | -0.088 | -0.912 | -0.0452 | 0.23023 | 0.1095567 | -0.1659 | -0.8341 | -0.137 | 0.27921 | 0.10328 | -0.312902567 | -0.6871 |
| 1 | 0.05 | | -1 | -1 | -1 | | Update on sample 3 | 0.0224 | 0.2136 | -0.0336 | -0.1576 | -0.8424 | -0.0869 | 0.27194 | 0.067850865 | -0.4267 | -0.5733 | -0.1713 | 0.31357 | 0.08763 | -0.572524881 | -0.4275 |
| | | | | | | | Update on sample 4 | -0.0197 | 0.25572 | 0.00852 | | | -0.1156 | 0.3006 | 0.096515738 | | | -0.1927 | 0.33494 | 0.11626 | | |
| | | | | | | | | b | w1 | w2 | y | | b | w1 | w2 | y | | b | w1 | w2 | y | |
| Test column | | | x1 | x2 | output | | | -0.0197 | 0.25572 | 0.00852 | | | -0.1156 | 0.3006 | 0.096515738 | | | -0.1927 | 0.33494 | 0.11626 | | |
| | | | 1 | 1 | 1 | | | -0.0197 | 0.25572 | 0.00852 | 0.24452 | | -0.1156 | 0.3006 | 0.096515738 | 0.28154 | | -0.1927 | 0.33494 | 0.11626 | 0.25850112 | |
| | | | 1 | -1 | -1 | | | -0.0197 | 0.25572 | 0.00852 | 0.22748 | | -0.1156 | 0.3006 | 0.096515738 | 0.08851 | | -0.1927 | 0.33494 | 0.11626 | 0.02598119 | |
| | | | -1 | 1 | -1 | | | -0.0197 | 0.25572 | 0.00852 | -0.2669 | | -0.1156 | 0.3006 | 0.096515738 | -0.3197 | | -0.1927 | 0.33494 | 0.11626 | -0.411378706 | |
| | | | -1 | -1 | -1 | | | -0.0197 | 0.25572 | 0.00852 | -0.284 | | -0.1156 | 0.3006 | 0.096515738 | -0.5127 | | -0.1927 | 0.33494 | 0.11626 | -0.643898637 | |

توضیح قسمت GD:

این قسمت متشکل از ۴ بخش به نام‌های initial values, first update, second update, third update می‌باشد. ستون‌های b, w1, w2, y در هر قسمت به وزن‌ها و خروجی تابع (بدون تابع فعالسازی) اشاره دارد. در قسمت initial values، مقادیر b, w1, w2 همان مقادیر اولیه مساله بوده و y در هر سطر به خروجی هر یک از ورودی‌ها اشاره دارد (یعنی y در هر سطر، به خروجی نورون با توجه به ورودی x1, x2 موجود در ستون A,B همان سطر اشاره دارد و مجموع y ها نیز در پایین محاسبه شده است (که برای محاسبه خطا استفاده می‌شود).

همچنین ستون error نیز به مقدار خطا برای هر ورودی اشاره دارد. ستون sum of errors نیز به مجموع خطاها اشاره دارد. دلیل استفاده از یک ستون بجای یک سلول برای sum of errors، راحت‌تر شدن قراردهی فرمول‌ها در سلول‌های بعدی می‌باشد که به این مقدار وابسته می‌باشد و صورت سوال ربطی ندارد.

مقادیر b, w1, w2 در بخش first update نیز به مقادیر وزن‌ها پس از اولین آپدیت اشاره دارد و مشابه بخش قبلی، y, error, sum of errors نیز با استفاده از وزن‌های آپدیت شده محاسبه شده است. برای بخش‌های بعدی نیز به همین ترتیب داریم.

در بخش SGD، جدول ما از دو بخش کلی تشکیل شده است :

۱- ستون ها : که با Round 1, Round 2, Round 3 نام گذاری شده است و به هر epoch اشاره دارد.

۲-سطرها : Update on sample # که به هر آپدیت با یک ورودی اشاره دارد.

در این بخش، روند به این صورت است:

در ابتدا، وزن‌های اولیه داده شده از مساله را در ستون Round 1 و سطر initial values داریم. y, error در این سطر و ستون نیز با توجه به همین مقادیر محاسبه می‌شوند. سپس به سطر Update on sample 1 و ستون Round 1 می‌رویم. در این بخش، وزن‌ها با توجه به ارور محاسبه شده در بخش قبلی آپدیت شده اند. سپس y, error در سطر Update on sample 1 و ستون Round 1 نیز با توجه به اولین آپدیت وزن‌ها محاسبه شده اند. سپس به سطر های بعدی رفته و به همین ترتیب محاسبه می‌کنیم تا در نهایت به سطر Update on sample 4 می‌رویم. در این سطر، وزن‌ها با توجه به آخرین ورودی آپدیت شده اند و یکبار وزن‌ها با توجه به تمام ورودی‌ها آپدیت شده اند.

در مرحله بعدی، به سطر initial values و ستون Round 2 می‌رویم. وزن‌های موجود در این سطر و ستون، همان وزن‌های محاسبه شده در بخش قبلی است. y, error نیز با توجه به این مقادیر محاسبه می‌شوند. سپس به سطر Update on sample 1 و ستون Round 2 می‌رویم که در این بخش، وزن‌ها براساس اولین ورودی آپدیت شده و خروجی و ارور نیز محاسبه شده اند و به همین ترتیب پیش می‌رویم تا به آخرین آپدیت برسیم.

برای SGD نیز یک بخش تست قرار داده شده است که خروجی را با توجه به آخرین آپدیت وزن‌ها در آن راند حساب می‌کند.

نتیجه:

با توجه به مقادیر y، روش SGD همگرایی سریع‌تری دارد و روش بهتری است. همچنین چون در مدل آدالاین، ما آموزش را برروی مدل بدون تابع فعالسازی اعمال می‌کنیم، به همین دلیل تابع فعالسازی اثری در آموزش ندارد هرچند در نتیجه نهایی، تابع پله به دلیل ۱ کردن برای مقادیر مثبت، به آموزش کمتری برای رسیدن به جواب دارد و از تابع relu که مقادیر مثبت را خودش خروجی می‌دهد بهتر است. وزن اولیه نیز اگر نزدیک‌تر به مقادیر بهینه باشد سرعت همگرایی را سریعتر و زمان لازم را کمتر می‌کند. نرخ یادگیری نیز اگر خیلی کم تنظیم شود، همگرایی آهسته و اگر زیاد تنظیم شود پرش در اطراف نقطه بهینه را شاهد خواهیم بود.

**سوال ۳:**

```
# Initialize parameters (bias, w1, w2) as zeros
#######################################
## code here
parameters = torch.tensor([[0],[0],[0]], dtype=torch.float64)
#######################################
```

در این بخش، وزن‌ها با توجه به راهنمایی تعریف و مقداردهی شده‌اند. همچنین برای راحتی کار با دیتاست، تایپ مقادیر برابر با float64 قرار داده‌شده است.

```
[ ]  # Convert data to PyTorch tensors
     ######################################
     ## code here
     X = torch.tensor(X)
     y = torch.tensor(y)
     ######################################
```

در این بخش نیز دیتاهای X,Y به تایپ تنسور تبدیل شده اند.

```
[ ] ###################################
    ## code here
    X = torch.cat((torch.ones(size=(X.shape[0],1)),X), dim=1)
    ###################################
```

در این بخش نیز بایاس با مقدار ۱ اضافه شده است. در ابتدا با تابع torch.ones، یک تنسور با ابعاد داده شده با تعداد سطرهای ورودی مساله و ۱ ستون ساخته شده است و سپس با تابع torch.cat، با X ترکیب شده است و با توجه به جدا سازی بایاس در بخش آخر (تصویر زیر)

```
# Extract bias and weights
b = parameters[0]
w1 = parameters[1]
w2 = parameters[2]
X = X[:, 1:]  # Remove the bias term from input matrix
y = y.view(-1)  # Reshape y for plotting
```

این بایاس با کمک پارامتر dim=1 در اولین ستون قرار داده شده است.

```python
def calculate_output(X, parameters):
    '''
    Calculate the output of perceptron with unit step activation function
    for all samples
    intput(s):
    X (ndarray): a 2-D array with the shape of (number of samples, 3)
    parameters (ndarray): a 2-D array with the shape of (3, 1)
    output(s):
    y_out (ndarray): a 2-D array with the shape of (number of samples, 1)
    '''
    ################################################
    ## code here
    ## y_out is the output of your neuron.
    ## First, you should implement matrix multiplicatoin of paramertrs and inputs
    ## Second, you should implement a Unit Step function as an activation function for each element of the y_out
    ## the y_out should be an array with the shape of (500,1).
    ## please note that you are not allowed to use any loop!!!
    mat_mul = torch.matmul(X,parameters)
    sum = torch.sum(mat_mul, dim = 1, keepdim=True)
    y_out = torch.heaviside(sum, torch.tensor([0], dtype=sum.dtype))
    ################################################

    return y_out
```

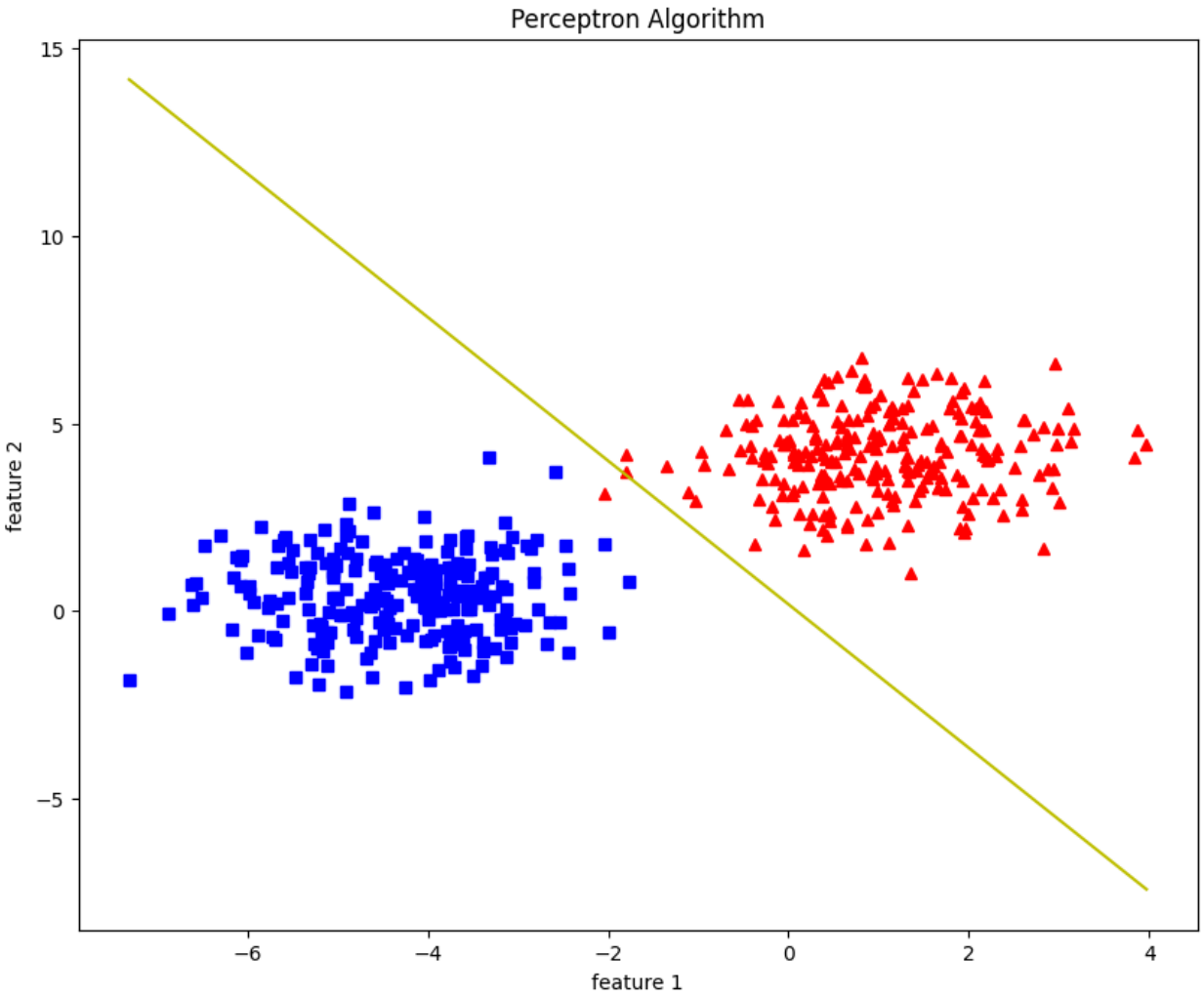
در این تصویر، در ابتدا ورودی X در پارامترها (وزن‌ها) ضرب شده است. سپس با تابع torch.sum، عملیات جمع کردن حاصل ضرب‌ها انجام شده است و سپس با کمک تابع torch.heaviside، مقادیر وارد تابع پله شده است. قابل به ذکر است که تابع torch.heaviside مقادیر کمتر از ۰ را برابر ۰ و مقادیر بیشتر از ۱ را برابر با ۱ و انتخاب خروجی مقادیر بین این دو را به برنامه نویس می‌سپارد که در اینجا با torch.tensor([0], dtype=sym.stype) این مقدار برابر با ۰ قرار داده شده است.

```
[ ]  for i in range(num_epochs):
        y_out = calculate_output(X, parameters)
        errors = 0

        for j in range(X.shape[0]):
            if y[j,0] != y_out[j,0] :
                errors += 1
            ####################################
            ## code here
            ## implement the code for updating each parameter
            ## note that you should calculate all parameters together.
            ## for example, you are not allowed to implement like the bellow:
            ## parameters[0] = ...
            ## paramters[1] = ...
            ## it should be like the bellow:
            ## paramters = ...
            parameters = parameters + ((y[j,0] - y_out[j,0]) * X[j] * lr).unsqueeze(dim=1)
            ####################################

        print("in epoch : ", i, " errors : ", errors)
```

قسمت آپدیت وزن‌ها نیز با توجه به خواسته این قسمت (آپدیت همه با هم و نه جدا جدا) انجام شده است و unsqueeze(dim=1) برای تنظیم ابعاد خروجی قرار داده شده است.

```
in epoch :  0  errors :  250
in epoch :  1  errors :  98
in epoch :  2  errors :  7
in epoch :  3  errors :  6
in epoch :  4  errors :  6
in epoch :  5  errors :  6
in epoch :  6  errors :  6
in epoch :  7  errors :  3
in epoch :  8  errors :  3
in epoch :  9  errors :  3
in epoch :  10  errors :  3
in epoch :  11  errors :  3
in epoch :  12  errors :  2
in epoch :  13  errors :  2
in epoch :  14  errors :  2
in epoch :  15  errors :  2
in epoch :  16  errors :  2
in epoch :  17  errors :  2
in epoch :  18  errors :  1
in epoch :  19  errors :  1
```

همچنین مقادیر ارور نیز به این صورت بهبود بافته است.




این تصویر نیز نتیجه نهایی مدل آموزش دیده شده را نمایش می‌دهد.

**سوال ۴) الف)**

```
def forward(self, X):
    """
    Compute the net input for the given input features.

    Parameters:
    - X (numpy.ndarray): Input features.

    Returns:
    - net (numpy.ndarray): The computed net input.
    """
    # TODO: Implement the forward pass to compute the net input
    # Hint: Use the dot product of X and weights and add the bias.
    net = np.dot(X, self.weight) + self.b
    return net
```

تصویر بالا، قسمت فوروارد را نشان می‌دهد که خروجی از ضرب ورودی در وزن‌ها و سپس جمع با بایاس حاصل می‌شود.

```
def step(self, X, net, y):
    """
    Update weights and bias based on the error.

    Parameters:
    - X (numpy.ndarray): Input feature vector.
    - net (float): Computed net input.
    - y (float): Actual target value.

    Returns:
    - float: The largest weight change to monitor convergence.
    """
    # TODO: Update weights and bias based on the error
    # Hint: Use the Adaline learning rule. Calculate delta_w and delta_b.

    self.b = self.b + self.lr * np.mean(np.subtract(y, net))
    self.weight = self.weight + np.dot(X.transpose(),(net - y)) * self.lr
```

این قسمت نیز پیاده سازی تابع step را نشان می‌دهد که با فرمولی که در اسلایدها بود وزن‌ها آپدیت می‌شوند.

```python
def train(self, X, Y, max_epochs=10):
    """
    Train the Adaline model using the provided data.

    Parameters:
    - X (numpy.ndarray): Input features for training.
    - Y (numpy.ndarray): Target labels for training.
    - max_epochs (int): Maximum number of training epochs (default is 10).
    """
    # TODO: Train the model until the stopping condition or max_epochs
    # Hint: Loop through epochs and update weights for each sample.
    previous_acc = np.nan

    for i in range(max_epochs):
        output = self.predict(X)
        self.step(X, output, Y)
```

در این بخش نیز آموزش مدل را داریم که به تعداد epoch های خواسته شده، برروی ورودی آموزش می‌دهیم. توجه شود که در مرحله آموزش از تابع فعالسازی پله استفاده نمی‌کنیم و خروجی مستقیما برای آپدیت استفاده می‌گردد.

```python
def activation_function(self, X):
    """
    Apply the activation function (step function) to the net input.

    Parameters:
    - X (numpy.ndarray): Net input values.

    Returns:
    - numpy.ndarray: Activated output values.
    """

    return np.vectorize(lambda x : 0 if x < 0 else 1)(X)
```

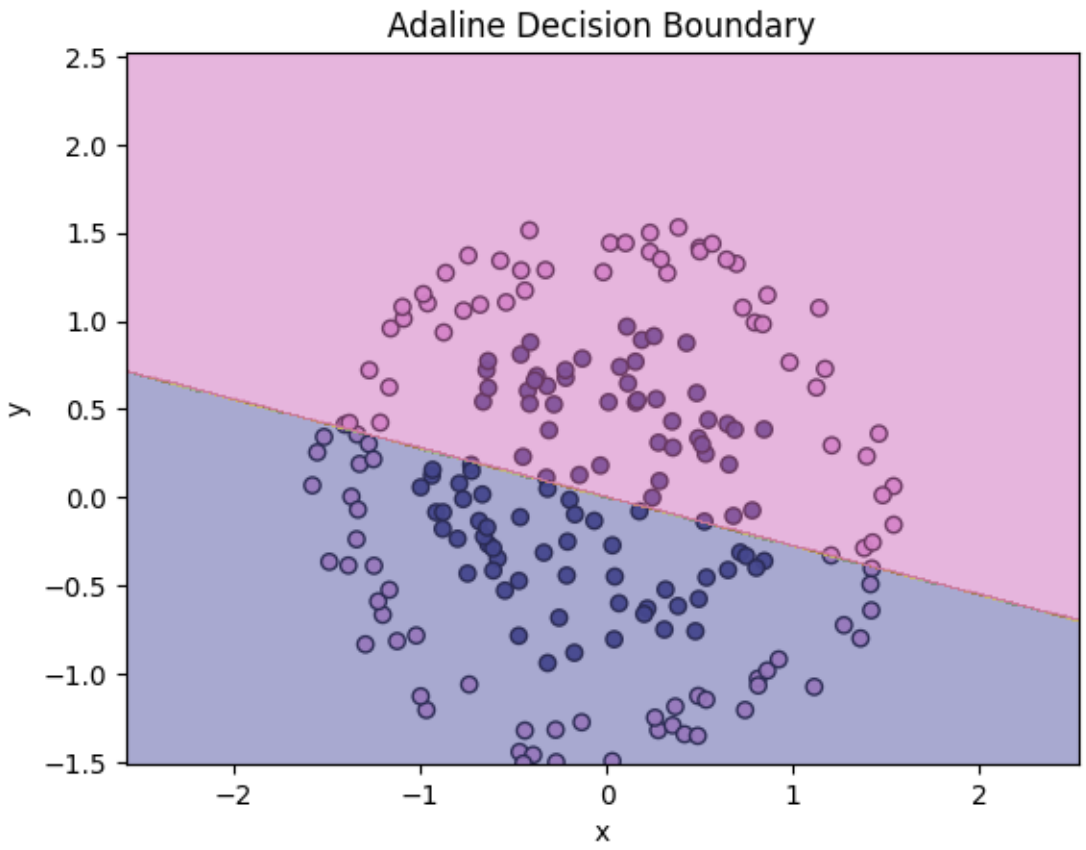
در اینجا برروی تک تک مقادیر با کمک تابع np.vectorize تابع فعالسازی اعمال می‌شود.

```
    def predict(self, X):
        """
        Predict class labels for new input data.

        Parameters:
        - X (numpy.ndarray): Input features for prediction.

        Returns:
        - numpy.ndarray: Predicted class labels.
        """
        # TODO: Predict class labels for new inputs
        forward = self.forward(X)
        acc_fn = self.activation_function(forward)
        return acc_fn
```

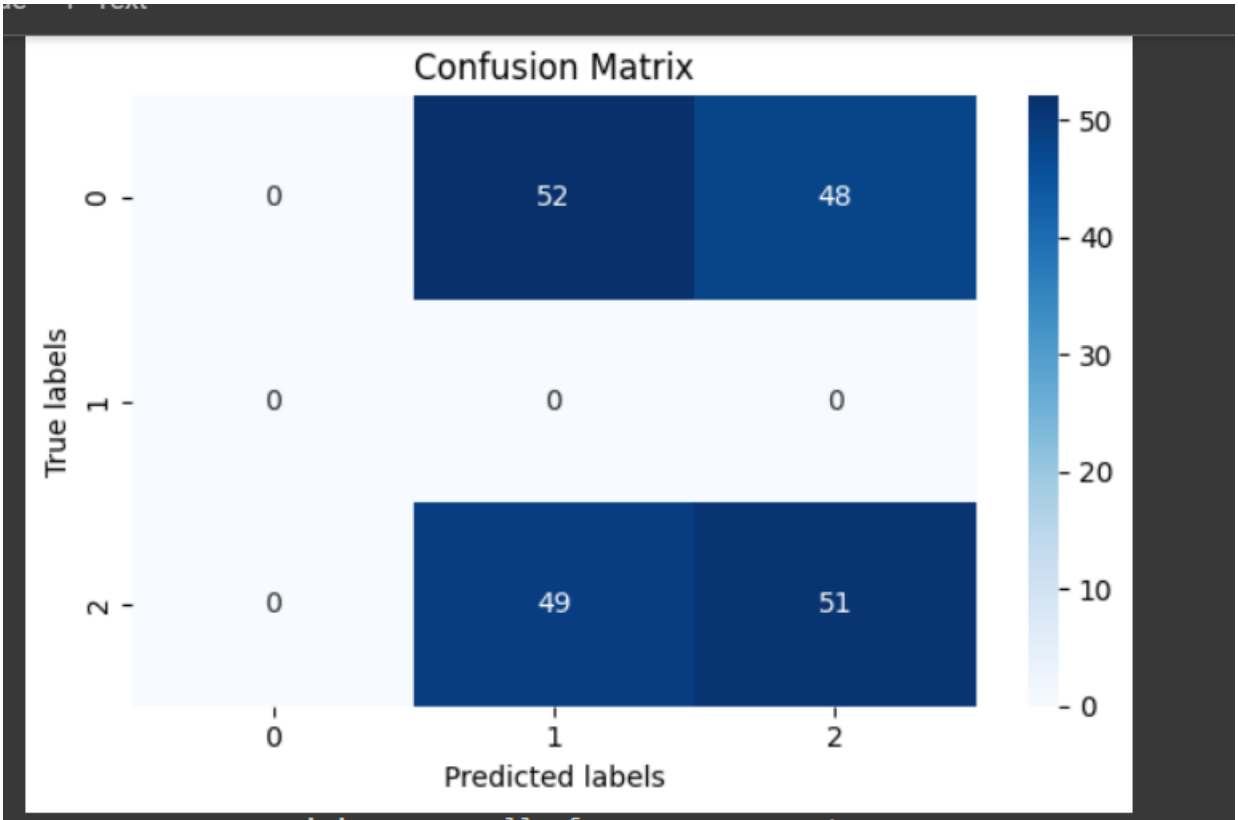
در قسمت پیش بینی نیز در ابتدا ورودی به مدل داده شده (که بدون تابع فعالسازی است)، سپس تابع فعالسازی برروی آن اعمال می‌شود تا خروجی معلوم گردد.

```
    def accuracy(self, X, y):
        """
        Calculate the accuracy of the predictions.

        Parameters:
        - X (numpy.ndarray): Input features for prediction.
        - y (numpy.ndarray): Actual target labels.

        Returns:
        - float: The accuracy of the model.
        """
        # TODO: Compute the accuracy of the predictions

        output = self.predict(X)
        acc = np.sum(output == y)
        return acc
```

در این بخش نیز دقت محاسبه می‌شود. به این صورت که خروجی گرفته شده، با لیبل مقایسه شده و تعداد مقادیر صحیح جمع می‌گردد.




تصویر بالا خروجی نهایی مدل نشان داده می‌شود. مدل ما بدلیل داشتن تنها یک نورون، فقط قادر به جداسازی تک خطی است و این خروجی، کاملا مطابق با انتظارات ما است که این مدل بیشتر از این قادر به جداسازی نیست.




| | precision | recall | f1-score | support |
|---|---|---|---|---|
| -1.0 | 0.00 | 0.00 | 0.00 | 100 |
| 0.0 | 0.00 | 0.00 | 0.00 | 0 |
| 1.0 | 0.52 | 0.51 | 0.51 | 100 |
| accuracy | | | 0.26 | 200 |
| macro avg | 0.17 | 0.17 | 0.17 | 200 |
| weighted avg | 0.26 | 0.26 | 0.26 | 200 |

تصویر بالا نیز متریک های مختلف را نمایش می‌دهد.

## سوال ۴)ب)

```
def initialize_weights(self, sm):
    """
    Initialize weights and biases for the Madaline model.

    Parameters:
    - sm (int): Size of the input data.

    This method initializes weights for the first and second layers, as well as biases.
    """
    # ToDo: Implement the weight initialization method.
    # Hints:
    # 1. Initialize weights for the first layer.
    # 2. Initialize biases for the first layer.
    # 3. Initialize weights and biases for the second layer.

    self.weights = np.random.random(size=(sm,self.num_neurons_layer1))
    self.biases = np.random.random(size=(self.num_neurons_layer1,))

    self.weights_layer2 = np.ones(shape=(self.num_neurons_layer1, self.num_neurons_layer2))
    self.biases_layer2 = np.zeros(shape=(self.num_neurons_layer2,))
```

تصویر بالا، مقداردهی اولیه‌ی وزن‌ها را نشان می‌دهد که با توجه به تعداد ویژگی‌های ورودی و تعداد نورون‌ها در هر لایه مقداردهی شده اند. همچنین به دلیل استفاده از نورون and در خروجی، وزن‌های لایه دوم مقداردهی برابر با ۱ داشته اند.

```
def apply_activation_function(self, net):
    """
    Apply the activation function to the net input.

    Parameters:
    - net (numpy.ndarray): Net input values.

    Returns:
    - numpy.ndarray: Activated output values.
    """

    # ToDo: Implement the activation function. (Step Function)
    return np.vectorize(lambda x : -1 if x<=0 else 1)(net)
```

تابع فعالسازی نیز (مشابه قسمت الف) زده شده است.

```
180     def train(self):
181         """
182         Train the Madaline model on the input data.
183 
184         This method implements the training loop for the Madaline model.
185         """
186         # ToDo: Implement the training loop.
187         # Hints:
188         # 1. Shuffle the dataset and reset the index.
189         # 2. Initialize necessary variables.
190         # 3. You might want to perform forward & backward prop.
191 
192         self.initialize_weights(self.inputs.shape[1])
193 
194         class_0 = []
195         class_1 = []
196         for i in range(self.max_iter):
197             net, output = self.forward_propagation(self.weights, self.inputs, self.biases, False)
198             self.update_weights(self.inputs, self.target, net, output)
199 
200         for i in range(self.inputs.shape[0]):
201             output = self.predict(self.inputs[i])
202             temp = {'x' : self.inputs[i,0], 'y' : self.inputs[i,1]}
203             if output == 1:
204                 class_1.append(temp)
205             else:
206                 class_0.append(temp)
207 
208         class_0.append({'x' : self.inputs[i,0], 'y' : self.inputs[i,1]})
209         df0 = pd.DataFrame(class_0)
210         df1 = pd.DataFrame(class_1)
211         self.plot_error([], df0, df1)
212         return self.weights, self.biases, []
```

قسمت آموزش نیز به این صورت زده شده است که در ابتدا مدل آموزش دیده (سطرهای 196 تا 198) و سپس داده‌ها پیش بینی شده و جدا می‌شوند (سطرهای 200 تا 206) و مقادیر پیش بینی شده به DataFrame تبدیل می‌شوند.

```
    def predict(self, inputs):
        """
        Predict class labels for the given input data.

        Parameters:
        - inputs (numpy.ndarray): Input data for prediction.

        Returns:
        - numpy.ndarray: Predicted class labels.
        """
        # ToDo: Implement the prediction function.
        # Me : Change this method completely. don't use forward_propagation here
        net = np.dot(inputs, self.weights) + self.biases
        net_activated = self.apply_activation_function(net)
        output = 1 if np.all(net_activated == 1) else -1
        return output
```

بخش پیش بینی نیز به این صورت وجود دارد. در ابتدا ورودی‌ها در وزن‌ها ضرب و با بایاس جمع شده، سپس فعالسازی شده و به ازای هر هر نقطه، لیبل آن خروجی داده می‌شود.

```
    def calculate_error(self, target, output):
        """
        Calculate the error based on the target and predicted output.

        Parameters:
        - target (numpy.ndarray): Actual target values.
        - output (numpy.ndarray): Predicted output values.

        Returns:
        - float: Calculated mean squared error.
        """
        # ToDo: Implement the error calculation.
        m = target.shape[0]
        error = 1/m * np.sum( (target-output) ** 2)
        return error
```

در این بخش نیز مقدار خطا با استفاده از روش MSE، محاسبه و خروجی داده می‌شود.

قابل به ذکر است که توابع دیگر نیز تکمیل شده است ولی به دلیل عدم خروجی مورد نظر و عدم یادگیری مدل، احتمال اشتباه در آنها وجود دارد و توضیح داده نمی‌شوند. همچنین به دلیل عدم خروجی صحیح مدل، خروجی مدل نیز بررسی نمی‌شود.