به نام خدا

دانشگاه تهران - دانشگاه صنعتی خواجه نصیرالدین طوسی تهران

دانشکده مهندسی برق و کامپیوتر

شبکه‌های آدالاین و مادالاین

نام و نام خانوادگی	محمدجواد احمدی
شمارهٔ دانشجویی	۴۰۱۰۰۰۸۶

فهرست مطالب

۱ پاسخ پرسش دوم ۴
۱.۱ AdaLine ۴
۱.۱.۱ قسمت الف ۴
۲.۱.۱ قسمت ب ۶
۳.۱.۱ قسمت ج ۱۳
۴.۱.۱ قسمت د ۱۶
۲.۱ MadaLine ۱۷
۱.۲.۱ قسمت الف ۱۷
۲.۲.۱ قسمت ب ۱۹
۳.۲.۱ قسمت ج ۲۰
۴.۲.۱ قسمت د ۳۱

فهرست تصاویر

۱ نمونه نمودار پراکندگی دو دسته دادهٔ تعریف‌شده Adaline ۶
۲ نمودار تابع خطای آدالاین به‌ازای فراپارامترهای مختلف ۱۲
۳ نمودار نمایش داده‌ها و خط جداساز حاصل از آموزش آدالاین به‌ازای فراپارامترهای مختلف ۱۳
۴ نمونه نمودار پراکندگی دو دسته دادهٔ تعریف‌شده Adaline ۱۴
۵ نمودار تابع خطای آدالاین به‌ازای فراپارامترهای مختلف ۱۴
۶ نمودار نمایش داده‌ها و خط جداساز حاصل از آموزش آدالاین به‌ازای فراپارامترهای مختلف ۱۵
۷ نمونه نمودار پراکندگی دو دسته دادهٔ تعریف‌شده Adaline (حالت سخت و پیچیده) ۱۵
۸ نمودار تابع خطای آدالاین به‌ازای فراپارامترهای مختلف ۱۶
۹ نمودار نمایش داده‌ها و خط جداساز حاصل از آموزش آدالاین به‌ازای فراپارامترهای مختلف ۱۷
۱۰ یک مادالاین با دو آدالاین در لایهٔ مخفی و یک آدالاین در خروجی. ۱۸
۱۱ نمونه نمودار پراکندگی دو دسته دادهٔ تعریف‌شده MadaLine ۲۰
۱۲ نتایج مربوط به استفاده از الگوریتم مادالاین (۳ نورون) ۲۹
۱۳ نتایج مربوط به استفاده از الگوریتم مادالاین (۳ نورون و نرخ یادگیری 0.01) ۲۹
۱۴ نتایج مربوط به استفاده از الگوریتم مادالاین (۴ نورون) ۲۹
۱۵ نتایج مربوط به استفاده از الگوریتم مادالاین (۴ نورون و نرخ یادگیری 0.0005) ۳۰
۱۶ نتایج مربوط به استفاده از الگوریتم مادالاین (۱۰ نورون و نرخ یادگیری 0.01) ۳۰
۱۷ نتایج مربوط به استفاده از الگوریتم مادالاین (۱۰ نورون و نرخ یادگیری 0.0001) ۳۰

فهرست جداول

۱ توزیع داده‌ها در هر دسته ۴

پرسش ۲. شبکه‌های AdaLine و MadaLine

۱ پاسخ پرسش دوم

۱.۱ AdaLine

توضیح پوشهٔ کدهای AdaLine

کدهای مربوط به این قسمت، علاوه بر پوشهٔ محلی کدها در این لینک آورده شده است.

۱.۱.۱ قسمت الف

برای تولید داده‌های به‌صورت نشان‌داده‌شده در شکل خواسته‌شده در صورت سوال (شکل)، از دستور `numpy.random.normal` استفاده می‌کنیم. از این دستور به‌صورت زیر استفاده می‌شود:

```
numpy.random.normal(loc, scale, size)
```

که در آن داریم:

- `loc`: میانگین توزیع
- `scale`: انحراف معیار توزیع
- `size`: تعداد نمونه‌ها

برای این که در هر بار اجرا داده‌هایی یکسان تولید شود از دستور زیر استفاده کرده‌ایم:

```
np.random.seed(100)
```

حال می‌خواهیم داده‌های مربوط به هر دسته را به‌صورتی که در صورت سوال بیان شده است (جدول ۱) تولید کنیم: از طرفی

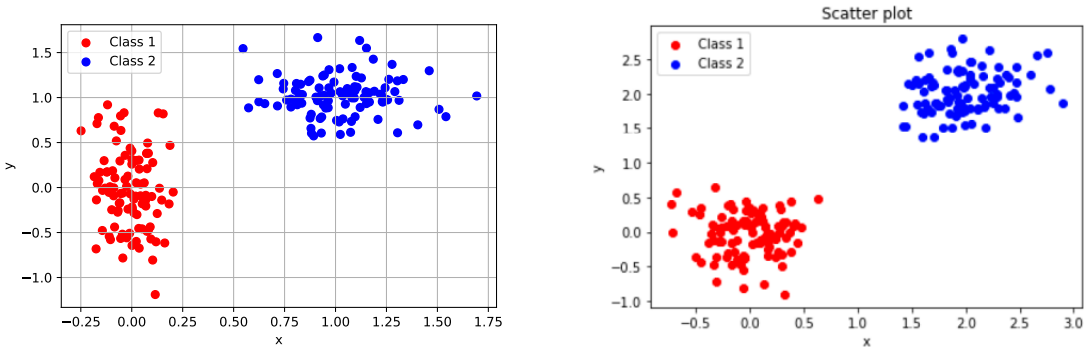
جدول ۱: توزیع داده‌ها در هر دسته

	دستهٔ اول		دستهٔ دوم	
متغیر	x	y	x	y
میانگین	0	0	1	1
انحراف معیار	0.1	0.4	0.2	0.2
تعداد داده	100	100	100	100

این اطلاعات داده‌شده با شکل ۲ موجود در صورت سوال (تصویر ۱۱(آ) در همین پاسخ‌ها) هم‌خوانی ندارد. واضح‌ترین تفاوت این است که میانگین کلاس دوم در تصویر ۱۱(آ)، دو است؛ درحالی‌که این میانگین در اطلاعات موجود در جدول ۱، یک بیان شده است. به هرحال، ما از دستورات برنامهٔ ۱ استفاده کردیم. نتیجه به صورتی است که در تصویر ۱۱(ب) آورده شده است.

Program 1: Code 2.1.1

```
np.random.seed(0)

avg_x_1 = 0
sd_x_1 = 0.1
Class1_x = np.random.normal(avg_x_1, sd_x_1, 100)

avg_y_1 = 0
sd_y_1 = 0.4
Class1_y = np.random.normal(avg_y_1, sd_y_1, 100)

Class1_Data = list(zip(Class1_x,Class1_y))
Class1_Label = np.ones(100)
Class1 = list(zip(Class1_Data,Class1_Label))

avg_x_2 = 1
sd_x_2 = 0.2
Class2_x = np.random.normal(avg_x_2, sd_x_2, 100)

avg_y_2 = 1
sd_y_2 = 0.2
Class2_y = np.random.normal(avg_y_2, sd_y_2, 100)

Class2_Data = list(zip(Class2_x,Class2_y))
Class2_Label = np.zeros(100) - np.ones(100)
Class2 = list(zip(Class2_Data,Class2_Label))

plt.scatter(Class1_x, Class1_y, c ="red", linewidths = 1)
plt.scatter(Class2_x, Class2_y, c ="blue", linewidths = 1)

plt.xlabel("x")
plt.ylabel("y")
plt.legend(["Class 1" , "Class 2"])
plt.grid(True)  # add grid
plt.savefig("Q211figure1.pdf")
plt.show()
```



(آ) هدف صورت سوال (ب) توزیع به‌دست‌آمده

شکل ۱: نمونه نمودار پراکندگی دو دسته دادهٔ تعریف‌شده Adaline در صورت سوال و پاسخ‌ها.

۲.۱.۱ قسمت ب

قبل از هرچیز تابع `Prepare_data` (برنامهٔ ۲) را برای آماده‌سازی، ترکیب و مخلوط‌کردن (بُرزدن) داده‌ها تعریف می‌کنیم.

Program 2: Code 2.1.2.1

```
def PrepareData(Class1,Class2):
    random.seed(0)
    AllData = Class1 + Class2
    random.shuffle(AllData)
    df = pd.DataFrame (AllData, columns = ['datapoints','labels'])
    x = df['datapoints']
    y = df['labels']
    return x,y

x, y = PrepareData(Class1,Class2)
```

در ادامه، با توجه به کتاب، به پیاده‌سازی الگوریتم مربوط به Adaline می‌پردازیم:

- تابع `RandomInitialize` (برنامهٔ ۳) را برای مقداردهی اولیهٔ اوزان و بایاس‌ها تعریف می‌کنیم. برای این که در هر بار اجرا داده‌هایی یکسان تولید شود از دستور `np.random.seed(50)` استفاده کرده‌ایم. هم‌چنین برای آن که به توصیهٔ کتاب، اوزان تولیدشده تا جای ممکن کوچک باشند، آن‌ها را در ضریب `sm` که می‌تواند عدد کوچکی مانند 0.01 باشد ضرب می‌کنیم.

Program 3: RandomInitialize Function

```
def RandomInitialize(sm):
    np.random.seed(100)
    w = np.random.rand(1,2) * sm
```

```
    b = np.zeros(1)
    print(f"Initial w={w}, Initial b={b}")
    return w,b
```

- در گام بعد، اوزان و ورودی‌ها را به‌صورت برداری در هم ضرب می‌کنیم. ضرب برداری امکان پردازش موازی و سرعت بالاتر را به ما می‌دهد. با توجه به ابعاد اوزان و ورودی‌ها، حاصل این ضرب اسکالر خواهد بود.

$$y_{in} = net = b + \sum_{i=1}^{n} w_i x_i \rightarrow net = b + np \cdot \mathrm{dot}(w, x) \tag{۱}$$

هم‌چنین برای داشتن قابلیت مقایسه بین حالات مختلف، از دو تابع فعال‌ساز sgn و tanh استفاده می‌کنیم. مجموعهٔ دستورات مربوط به این گام در برنامهٔ ۴ نوشته شده است.

Program 4: ForwardPath Functions

```
def sgn(net):
    if net >= 0:
        return 1
    else:
        return -1

def tanh(net):
    return np.tanh(net)

def Forward(w,x,b,ActFunc):
    net = np.dot(w,x) + b
    if ActFunc == 'sgn':
        h = sgn(net)
    elif ActFunc == 'tanh':
        h = tanh(net)
    return h, net
```

- برای به‌روزرسانی‌های موردنیاز از روابط آورده‌شده در کتاب (رابطهٔ ۲) الهام می‌گیریم و آن را با توجه به اصل بهتربودن عملیات ریاضیاتی برداری پیاده‌سازی می‌کنیم. ضریب alpha را هم مطابق روابط کتاب، به‌صورت یک فراپارامتر مهم برای نرخ یادگیری در دستورات تعبیه می‌کنیم. بدیهی است که در حالت استفاده از تابع فعال‌ساز tanh روابط دستخوش تغییراتی خواهند شد.

$$\begin{aligned} b(new) &= b(old) + \alpha\,(t - y_{in}) \\ w_i(new) &= w_i(old) + \alpha\,(t - y_{in})\,x_i. \end{aligned} \tag{۲}$$

توابع هزینه را هم مطابق خواست سوال با توجه به رابطهٔ $\frac{1}{2}(t - net)^2$ پیاده‌سازی کرده‌ایم. دستورات مربوط به این گام در برنامهٔ ۵ نوشته شده است.

Program 5: UpdateCost Functions

```
def DeltaUpdate(w,b,x,t,net,alpha):
```

```
    t = t.reshape(1,)
    x = x.reshape((1,2))
    delta_w = alpha * np.dot((t - net), x)
    delta_b = alpha * (t - net)
    w = w + delta_w
    b = b + delta_b

    return w,b

def sgnCost(t,net):
    error = 0.5 * np.power((t-net),2)
    return error

def tanhUpdate(w,b,x,t,h,alpha,gamma):
    t = t.reshape(1,)
    x = x.reshape((1,2))
    diff_w = alpha * (1-h**2) * gamma * np.dot((t - h), x)
    diff_b = alpha * (1-h**2) * gamma * (t - h)
    w = w + diff_w
    b = b + diff_b

    return w,b

def tanhCost(t,gamma,net):
    error = 0.5 * np.power( t - gamma * np.tanh(net),2)
    return error
```

- با فراخوانی گام‌های قبلی در تابع، شرط توقف را به این صورت تعریف می‌کنیم که اگر یکی از حالات رسیدن به بیشینهٔ تعداد ایپاک تعریف‌شده و یا کم‌ترشدن تابع هزینهٔ تعریف‌شده از یک مقدار بسیار کوچک رخ داد، فرآیند آموزش با برگشت‌دادن خط جداکننده اتمام یابد. برای این منظور برنامهٔ ۶ تعریف شده است. تابع به صورتی نوشته شده است که به سادگی بتوان آن را به‌ازای فراپارامترهای مختلف آزمایش کرد.

Program 6: AdaLine Functions

```
def SeparationLine(start,end,w,b):
    x= np.linspace(start,end)
    y = -(w[0][0] * x + b)
    y = y / w[0][1]
    return x,y

def Adaline(x,y,max_iter,learning_rate,actfunc,samples):
    cost_list = []
    eps = 0.00005 # End criteria
    sm = 0.01 # In order to make small weights
    gamma = 0.005  #Hyperparamter for tanh actfunc
    w,b = RandomInitialize(sm)          #Step 0
```

```
    print("Hyperparams are: ",f"eps={eps}, max_iter={max_iter}, learning_rate={learning_rate
    }, actfunc={actfunc}, sm={sm}, gamma={gamma}")
    for i in range(max_iter):           #Step 1
        h, net = Forward(w,np.asarray(x[i%samples]),b,actfunc) #Step 3
        if actfunc == 'sgn':
            cost = sgnCost(np.asarray(y[i%samples]),net)
        elif actfunc == 'tanh':
            cost = tanhCost(np.asarray(y[i%samples]),net,gamma)
        #After an Epoch
        if i % samples == 0 and i!=0:   #Step5
            cost_list.append(cost)
            error = np.mean(cost_list)
            print('Epoch %d / %d - Error: %f' % (len(cost_list), int(max_iter/samples), cost
    ))
            print('w:', w)
            print('b:', b)
            if error <= eps:
                return w,b, cost_list
        if actfunc == 'sgn':
            w,b = DeltaUpdate(w,b,np.asarray(x[i%samples]),np.asarray(y[i%samples]),net,
    learning_rate)
        elif actfunc == 'tanh':
            w,b
        w,b = tanhUpdate(w,b,np.asarray(x[i%samples]),np.asarray(y[i%samples]),h,
    learning_rate, gamma) #Step 4
    return w,b, cost_list                #Step 6
```

برای آزمایش و نمایش نتیجهٔ تابع اتلاف و ترسیم خط جداساز به‌ازای فراپارامترهای مختلف دستورات برنامهٔ ۷ را نوشته‌ایم. این دستورات یک مدل آدالاین را با استفاده از چندین فراپارامتر آموزش می‌دهد. در ابتدا فراپارامترها را تعریف می‌کنیم. این فراپارامترها شامل این‌هاست: حداکثر تعداد بارهایی است الگوریتم یادگیری اجرا می‌شود، یک لیست از مقادیر نرخ یادگیری برای اجرای الگوریتم، و یک لیست از توابع فعال‌ساز مورد استفاده برای تولید خروجی. در ادامه و برای نمایش نمودار خطای آموزش، زیرنمودارهایی به‌ازای این فراپارامترها تشکیل می‌دهیم و عنوان شکل را تنظیم می‌کنیم. سپس، دو لیست خالی برای ذخیره‌سازی اوزان و بایاس‌ها می‌سازیم. در ادامه، برای هر ترکیب از فراپارامترها، یک مدل آدالاین را با استفاده از تابع مربوطه (برنامهٔ ۶) آموزش داده می شود و خطای آموزش برای هر ایپاک و برای هر یک از مقادیر فراپارامتری در زیرنمودار متناظر با آن نمایش داده می‌شود. در انتها نیز دستوراتی برای تنظیم برچسب محورها، شبکهٔ خطوط، تنظیم فونت و اضافه‌کردن مناسب عناوین و توضیحات به نمودار نوشته شده است. درنهایت هم به ذخیره‌سازی و نمایش مقادیر خطا، اوزان و بایاس می‌پردازیم. وزن و بایاس نهایی مدل را به‌صورت جداگانه نیز نمایش داده‌ایم. نتیجه به‌صورتی است که در شکل ۲ و شکل ۳ نشان داده شده است. همان‌طور که مشاهده می‌شود، در بیش‌ترِ حالات و فراپارامترهای مورد استفاده، نمودار خطا و خط تفکیک‌کننده، جداسازی مناسبی را رقم زده‌اند.

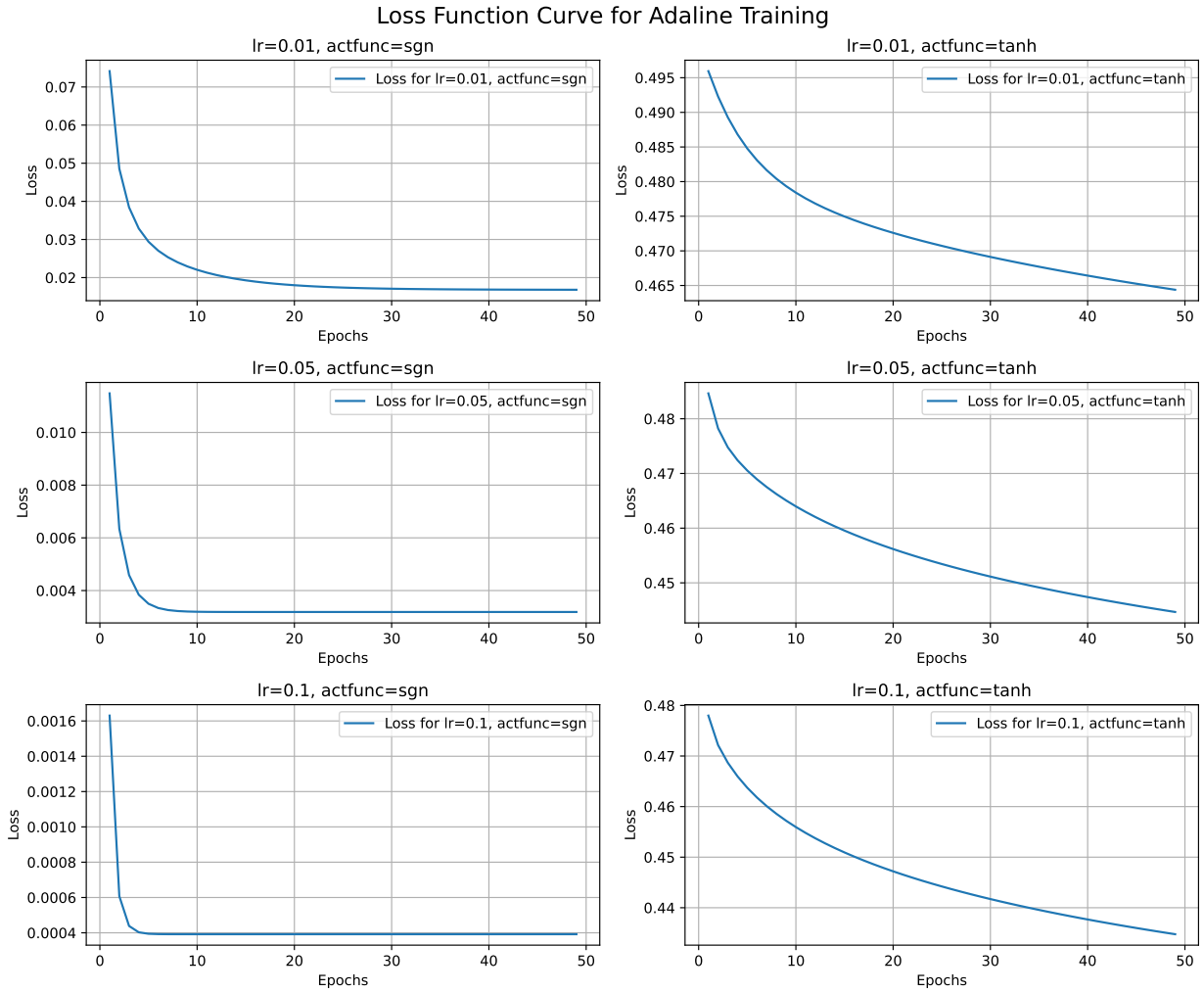
Program 7: AdaLine Functions

```
# define hyperparameters
max_iter = 10000
```

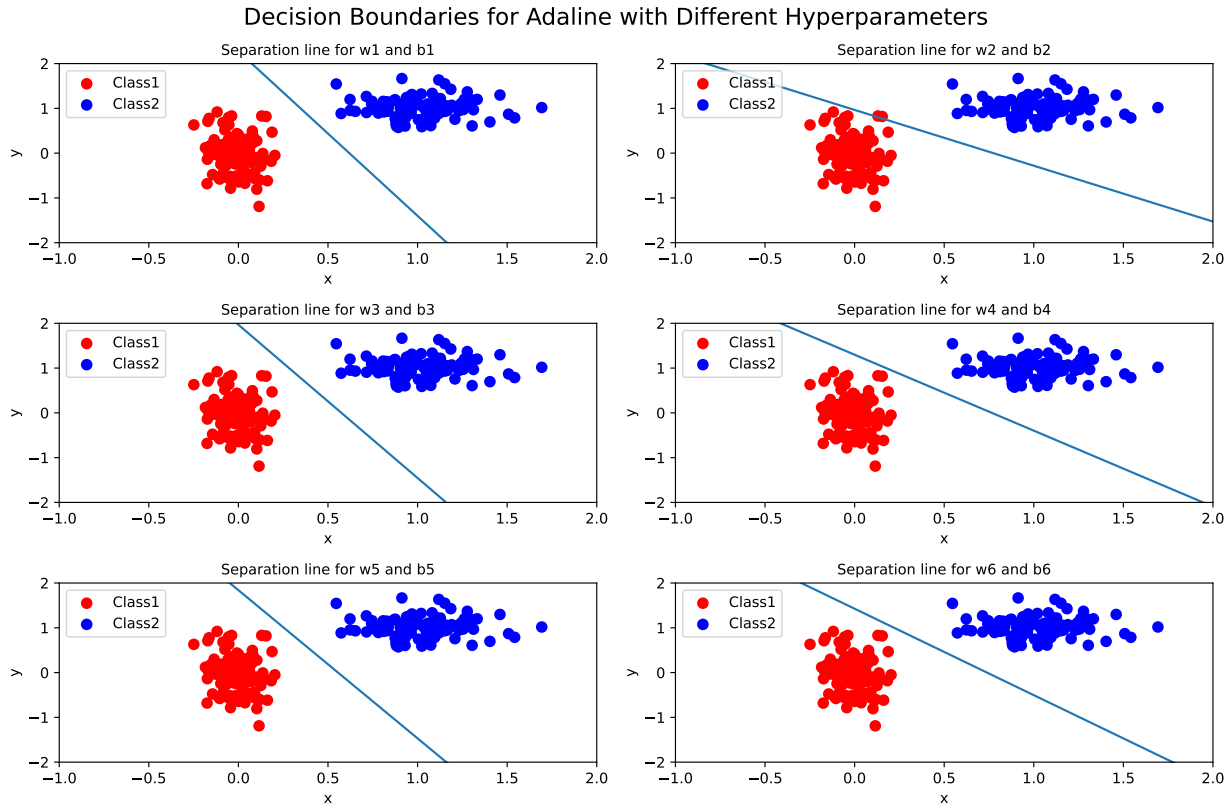
```
learning_rates = [0.01, 0.05, 0.1]
actfuncs = ['sgn', 'tanh']

# create subplots
fig, axs = plt.subplots(len(learning_rates), len(actfuncs), figsize=(12, 10))

# set the figure title
fig.suptitle('Loss Function Curve for Adaline Training', fontsize=16)

# define empty lists for weights and biases
weights = []
biases = []

# iterate over hyperparameters and plot the error curves
for i, lr in enumerate(learning_rates):
    for j, actfunc in enumerate(actfuncs):
        w, b, error_lst = Adaline(x, y, max_iter=max_iter, learning_rate=lr, actfunc=actfunc,
    samples=200)
        itr = range(1, len(error_lst) + 1)
        axs[i, j].plot(itr, error_lst, label='Loss for ' f'lr={lr}, actfunc={actfunc}')

        # add axis labels
        axs[i, j].set_title(f'lr={lr}, actfunc={actfunc}', fontsize=12)
        axs[i, j].set_xlabel('Epochs')
        axs[i, j].set_ylabel('Loss')

        # add grid
        axs[i, j].grid(True)

        # adjust tick label font size
        axs[i, j].tick_params(axis='both', which='major', labelsize=10)

        # add legend
        axs[i, j].legend(loc='upper right')

        # save weights and biases
        weights.append(w)
        biases.append(b)

        # print weights and bias
        print(f'wi: {w}x + {b}')

# adjust subplot spacing
fig.tight_layout()

```

```
# save the figure
plt.savefig("Q22figure2.pdf")

# show the plot
plt.show()

# create subplots for decision boundaries
fig2, axs2 = plt.subplots(3, 2, figsize=(12, 8))

# set the figure title
fig2.suptitle('Decision Boundaries for Adaline with Different Hyperparameters', fontsize=16)


# plot the weight and bias separation lines

for i, (w, b) in enumerate(zip(weights, biases)):
    px1, px2 = SeparationLine(-3, 3, w, b)

    plt.subplot(3, 2, i + 1)
    plt.scatter(Class1_x, Class1_y, c="red", linewidths=2)
    plt.scatter(Class2_x, Class2_y, c="blue", linewidths=2)
    plt.plot(px1, px2)

    plt.xlim(-1, 2)
    plt.ylim(-2, 2)
    plt.xlabel("x")
    plt.ylabel("y")
    plt.legend(["Class1", "Class2"], loc='upper left')

    plt.title(f"Separation line for w{i+1} and b{i+1}", fontsize=10)

plt.tight_layout()
plt.savefig("Q22figure3.pdf")
plt.show()
```

از شکل ۲ مشخص است که تابع اتلاف در طول ایپاک‌های مختلف روندی کاهشی دارد. این که با تعداد محدودی ایپاک می‌توان به اتلاف بسیار پایین دست پیدا کرد نشان می‌دهد که تفکیک داده‌ها از هم راحت بوده و نیازی به مدل پیچیده وجود ندارد. هم‌چنین همان‌طور که از غالب اشکال موجود در شکل ۳ برمی‌آید، خط جداکننده به خوبی داده‌ها را از هم جدا کرده است. این موضوع می‌تواند به این دلیل باشد که داده‌ها به‌صورت خطی تفکیک‌پذیر هستند و تعداد نمونه‌های کلاس‌ها برابرند. هم‌چنین عدم موفقیت در برخی حالات می‌تواند به تفاوت در توزیع کلاس‌ها برگردد. از مقایسهٔ فراپارامترهای مختلف می‌توان به این نتیجه رسید که بهترین نتیجه موقعی حادث می‌شود که از فعال‌ساز sgn و نرخ یادگیریِ alpha=0.01 استفاده می‌کنیم. لازم به ذکر است که در صورت تعریف تعداد ایپاک کم‌تر برای آموزش الگوریتم، ممکن بود این حد از موفقیت حاصل نشود؛ اما از آن‌جا که این قسمت از برنامه به‌صورت پویا نوشته شده است، نتایج خوبی حاصل شده است.



شکل ۲: نمودار تابع خطای آدالاین به‌ازای فراپارامترهای مختلف.

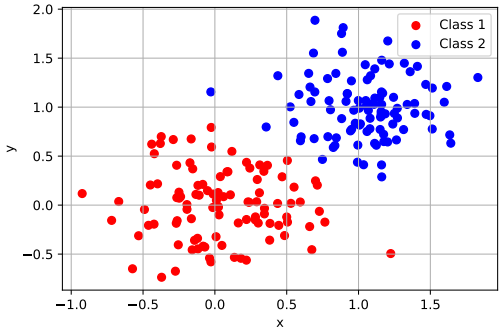


شکل ۳: نمودار نمایش داده‌ها و خط جداساز حاصل از آموزش آدالاین به‌ازای فراپارامترهای مختلف.

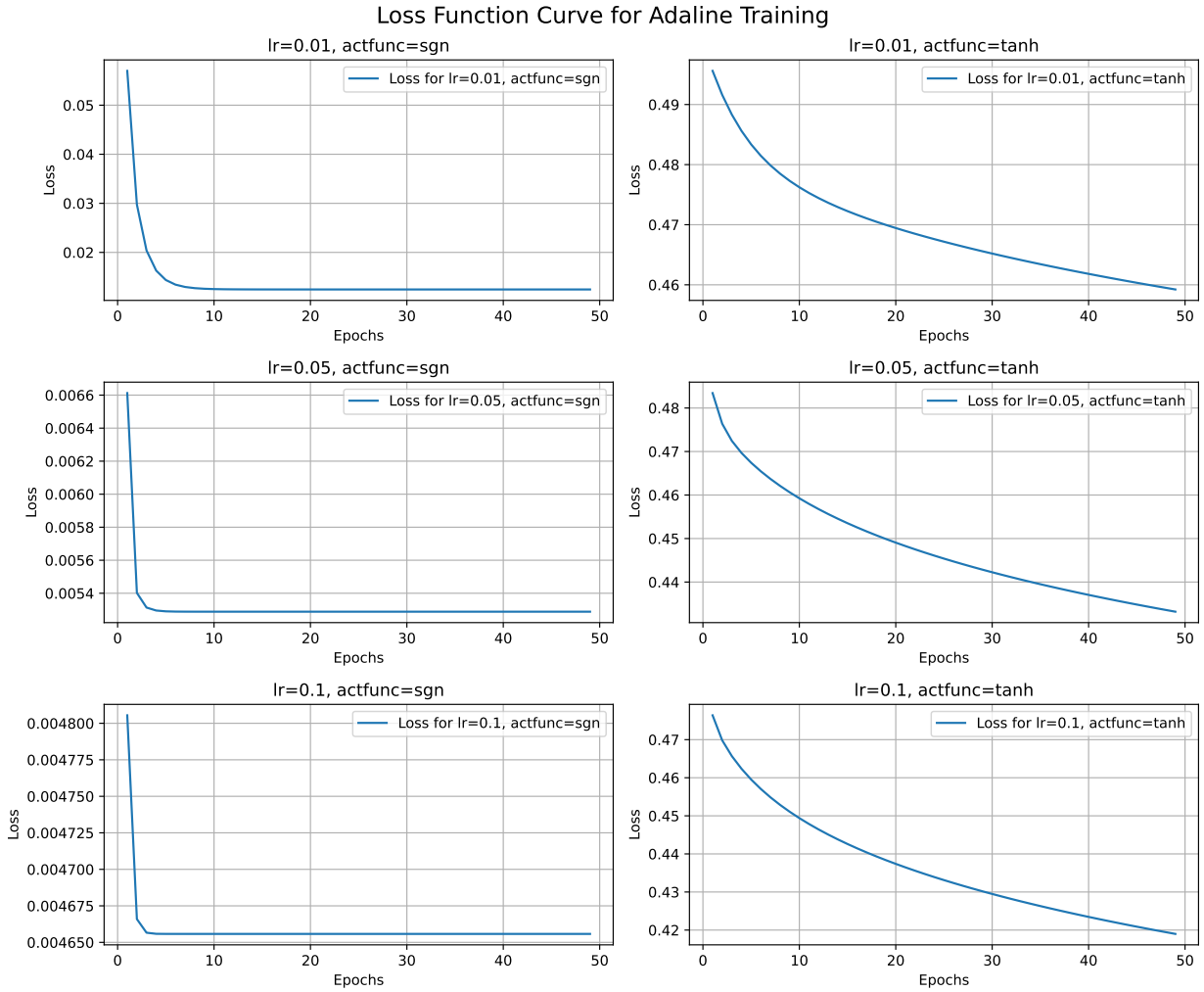
۳.۱.۱ قسمت ج

در این قسمت با صرف‌نظر از تکرار دستورات، صرفاً به نمایش نتایج به‌دست آمده می‌پردازیم. نتیجهٔ نمایش توزیع داده‌ها به صورتی است که در شکل ۵ آورده شده است. همان‌طور که مشاهده می‌شود توزیع کلاس‌ها در این قسمت کمی نامتوازن است و فاصلهٔ میان کلاس‌ها بسیار به هم نزدیک است. نتایجی که با اجرای برنامهٔ ۷ به دست آمده است در شکل ۵ و شکل ۶ نمایش داده شده است. همان‌طور که مشاهده می‌شود به علت پیچیدگی بیش‌تر توزیع داده‌ها نسبت به قسمت ب، نتایج مربوط به تابع اتلاف کاهش چندانی پیدا نکرده است، و خط جداساز نسبت به حالت قبل با حاشیهٔ امن کم‌تری عمل کرده است.

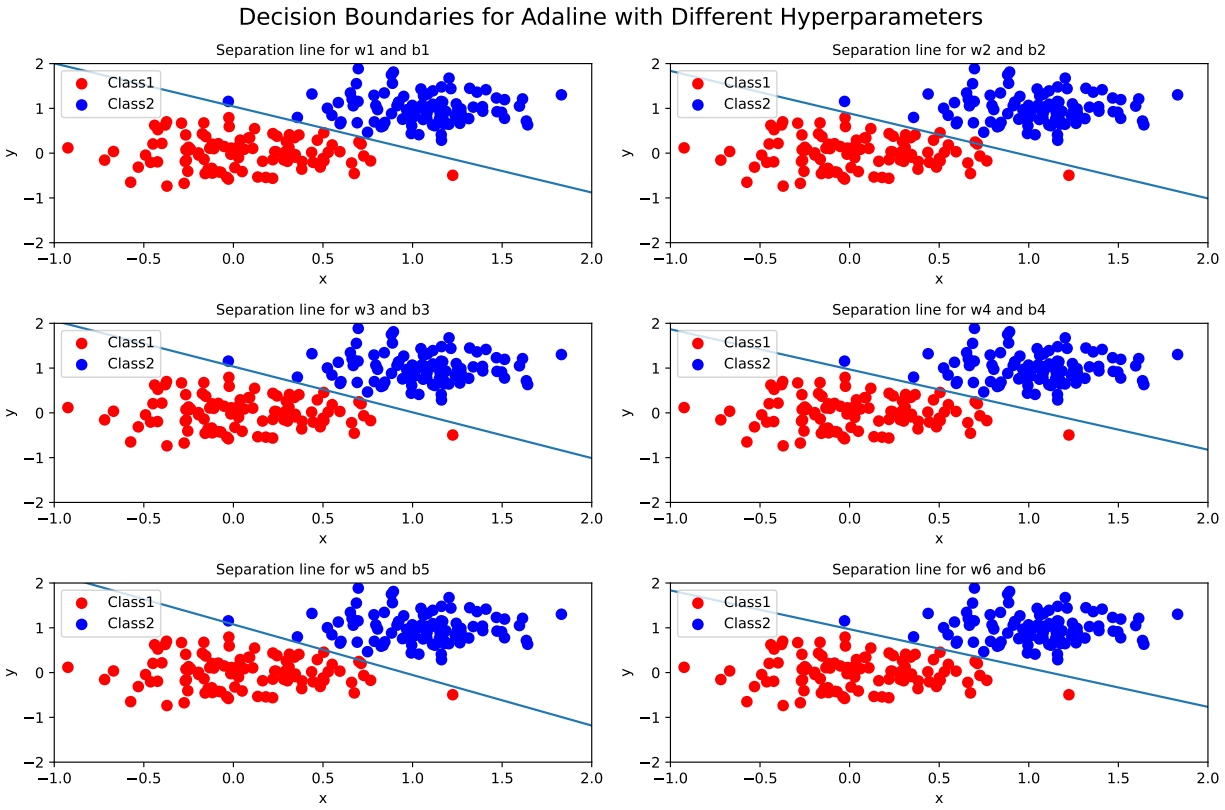
حال با تغییر `np.random.seed` توزیع داده‌ها را کمی پیچیده‌تر و به‌صورتی که در شکل ۷ نمایش داده شده است درمی‌آوریم. مشاهده می‌شود که در جاهایی کلاس‌ها تداخل مکانی پیدا می‌کنند و کار بسیار سخت و پیچیده می‌شود. نتیجه در این حالت به صورتی است که در شکل ۸ و شکل ۹ نمایش داده شده و چندان مطلوب نیست.



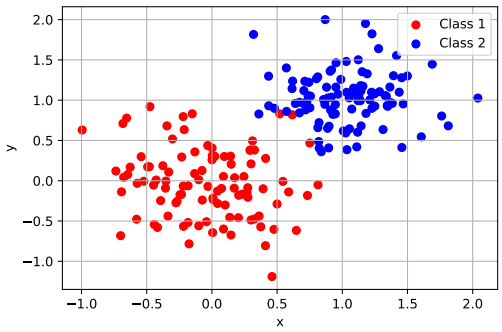
شکل ۴: نمونه نمودار پراکندگی دو دسته دادهٔ تعریف‌شده Adaline.



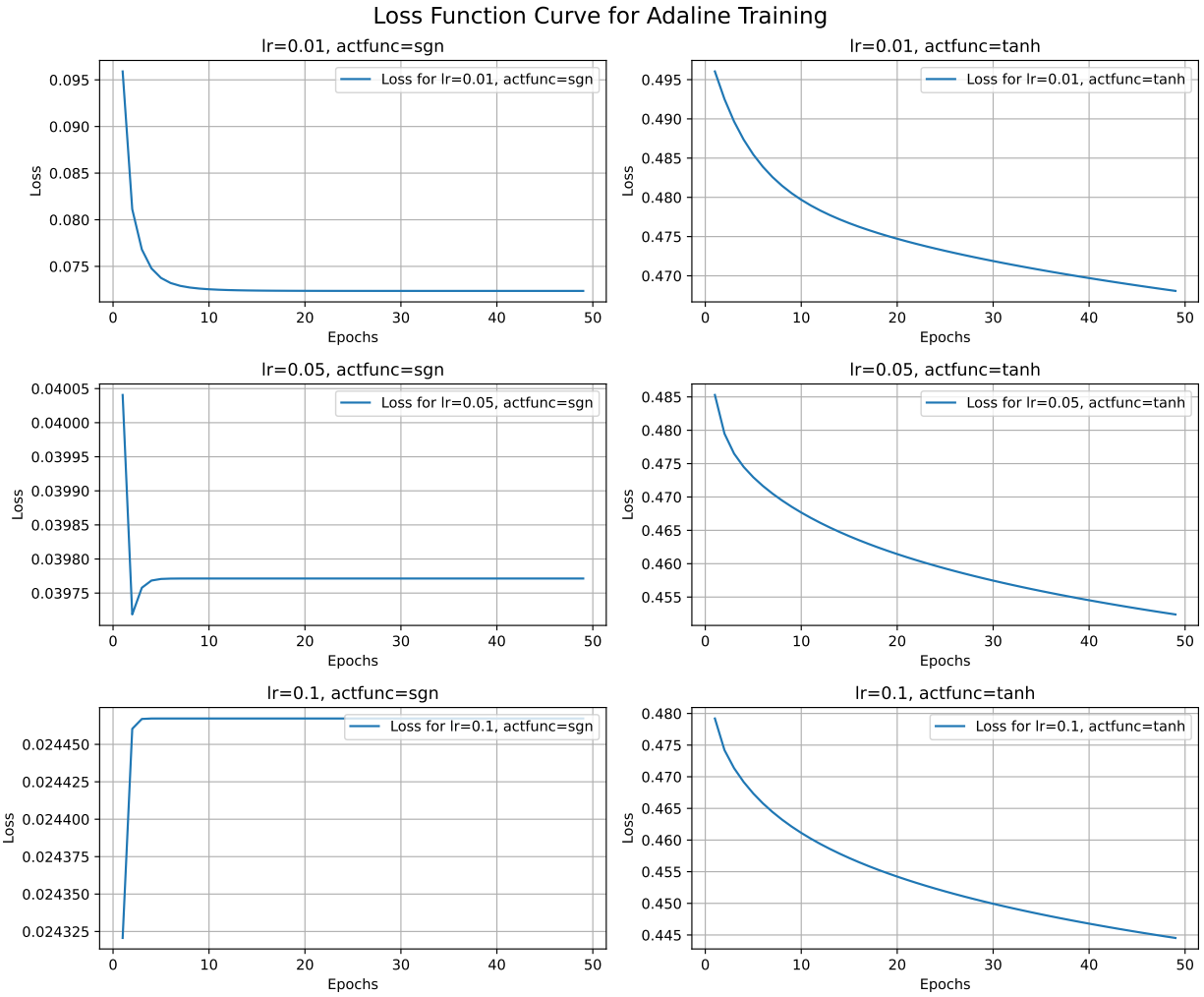
شکل ۵: نمودار تابع خطای آدالاین به‌ازای فراپارامترهای مختلف.



شکل ۶: نمودار نمایش داده‌ها و خط جداساز حاصل از آموزش آدالاین به‌ازای فراپارامترهای مختلف.



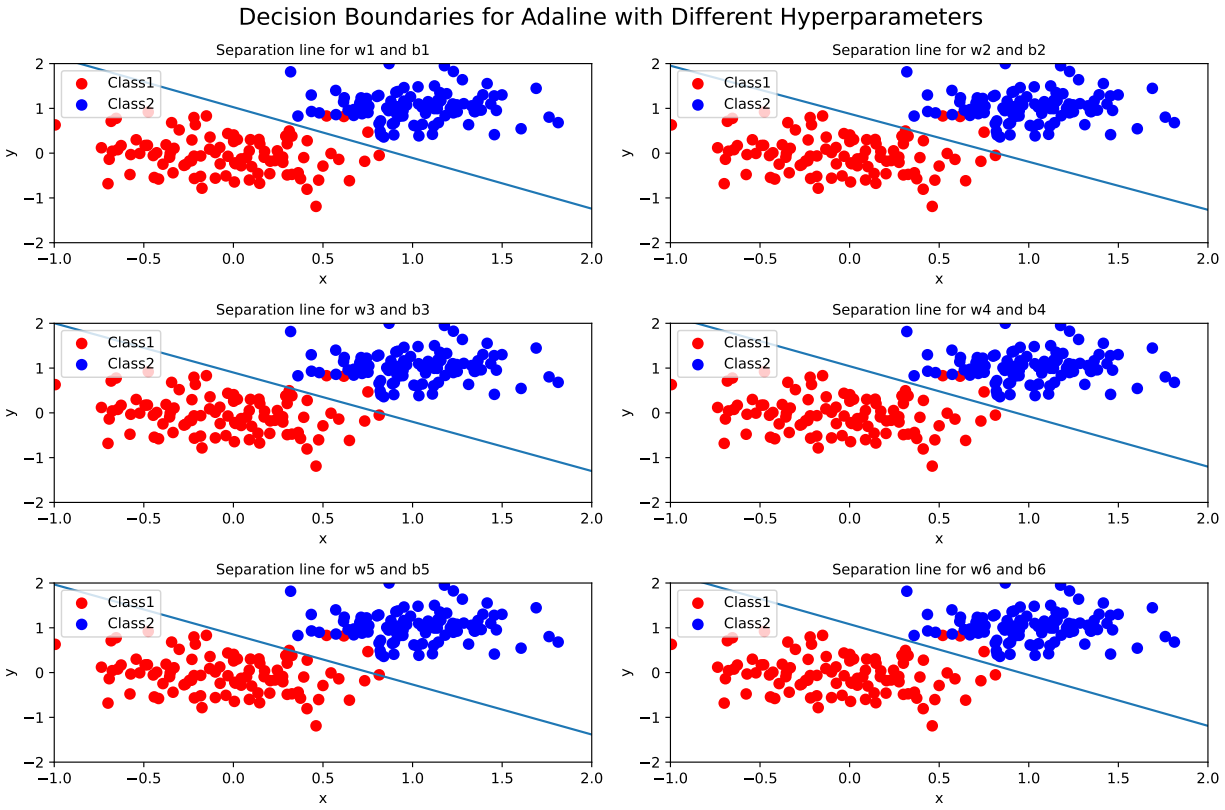
شکل ۷: نمونه نمودار پراکندگی دو دسته دادهٔ تعریف‌شده Adaline (حالت سخت و پیچیده).



شکل ۸: نمودار تابع خطای آدالاین به‌ازای فراپارامترهای مختلف.

۴.۱.۱ قسمت د

با مقایسهٔ نتایج به‌دست آمده مشخص است که داده‌های قسمت ب به‌صورت کاملاً خوبی از هم تفکیک شده‌اند. این به علتی است که حاشیهٔ زیادی بین دو کلاس وجود دارد. این در حالی است که داده‌های قسمت ج در تفکیک‌پذیری مشکل دارند. مخصوصاً وقتی با تغییر مقدار تصادفی داده‌ها به صورتی درآوردیم که کلاس‌ها داخل یکدیگر رفتند این پیچیدگی مضاعف باعث بهم‌ریختن و عدم امکان تفکیک مناسب در برخی از حالات شد.



شکل ۹: نمودار نمایش داده‌ها و خط جداساز حاصل از آموزش آدالاین به‌ازای فراپارامترهای مختلف.

۲.۱ MadaLine

توضیح پوشهٔ کدهای MadaLine

کدهای مربوط به این قسمت، علاوه بر پوشهٔ محلی کدها در این لینک آورده شده است.

۱.۲.۱ قسمت الف

الگوریتم MRI برای آموزش مادالاین (MadaLine): شکل ۱۰ از کتاب را در نظر می‌گیریم. در الگوریتم MRI که شیوهٔ اصلی آموزش مادالاین است، فقط اوزان مربوط به آدالاین‌های مخفی تنظیم می‌شوند و اوزان واحدهای خروجی ثابت هستند. تابع فعال‌ساز برای واحدهای Z_1، Z_2 و Y به صورتی است که در ؟؟ آورده شده است.

- گام صفرم: اوزان را مقداردهی اولیه می‌کنیم. برای اوزان اولیهٔ آدالاین از مقادیر کوچک تصادفی استفاده می‌کنیم. نرخ یادگیری α را هم یک مقدار کوچک (همانند الگوریتم آموزش آدالاین) در نظر می‌گیریم.
- گام اول: تا زمانی که شرایط توقف برقرار نباشد، گام‌های دوم تا هشتم را تکرار می‌کنیم.
- گام دوم: برای هر جفت آموزشی دوتایی $s : T$ گام‌های سوم تا هفتم را انجام می‌دهیم.

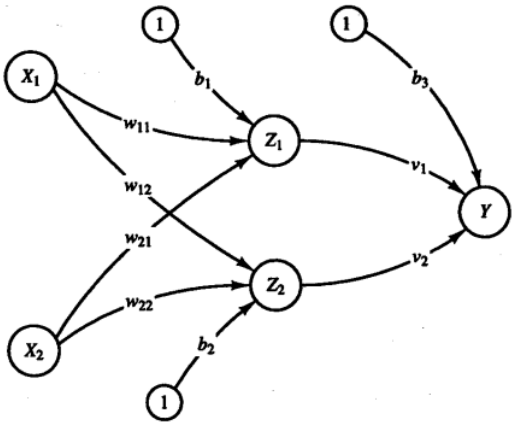


Figure 2.24 A MADALINE with two hidden ADALINES and one output ADALINE.

شکل ۱۰: یک مادالاین با دو آدالاین در لایهٔ مخفی و یک آدالاین در خروجی.

- گام سوم: فعال‌سازهای واحدهای ورودی را تعیین می‌کنیم:

$$x_i = s_i \tag{۳}$$

- گام چهارم: ورودی شبکه به هر واحد آدالاین مخفی را محاسبه می‌کنیم:

$$\begin{aligned} z_{in_1} &= b_1 + x_1 w_{11} + x_2 w_{21} \\ z_{in_2} &= b_2 + x_1 w_{12} + x_2 w_{22} \end{aligned} \tag{۴}$$

- گام پنجم: خروجی هر واحد آدالاین مخفی را تعیین می‌کنیم:

$$\begin{aligned} z_1 &= f\left(z_{in_1}\right) \\ z_2 &= f\left(z_{in_2}\right) \end{aligned} \tag{۵}$$

- گام ششم: خروجی شبکه را تعیین می‌کنیم:

$$\begin{aligned} y_{in} &= b_3 + z_1 v_1 + z_2 v_2 \\ y &= f\left(y_{in}\right). \end{aligned} \tag{۶}$$

- گام هفتم: خطا را تعیین و اوزان را به‌روز می‌کنیم. اگر $y = t$ باشد اوزان به‌روز نمی‌شوند و در غیر این صورت اگر $t = 1$ بود، اوزانِ روی Z_J؛ یعنی واحدی که به ورودی شبکهٔ آن به صفر نزدیک‌تر است به‌روز می‌شود:

$$\begin{aligned} b_J(\text{new}) &= b_J(\text{old}) + \alpha\left(1 - z_{in_J}\right), \\ w_{iJ}(\text{new}) &= w_{iJ}(\text{old}) + \alpha\left(1 - z_{in_J}\right) x_i \end{aligned} \tag{۷}$$

و اگر $t = -1$ بود، اوزانِ روی تمام واحدهای Z_k که ورودی شبکه مثبت دارند به‌روز می‌شود:

$$\begin{aligned} b_k(\text{new}) &= b_k(\text{old}) + \alpha\left(-1 - z_{in_k}\right), \\ w_{ik}(\text{new}) &= w_{ikJ}(\text{old}) + \alpha\left(-1 - z_{in_k}\right) x_i \end{aligned} \tag{۸}$$

- گام هشتم: شرایط توقف را آزمایش می‌کنیم. اگر تغییر اوزان متوقف شده است یا به سطح قابل قبولی رسیده است و یا اگر به‌روزرسانی اوزان در گام دوم به سقف تعداد تکرارهای مشخص‌شدخ رسیده است، کار را متوقف می‌کنیم و در غیر این صورت ادامه می‌دهیم.

۲.۲.۱ قسمت ب

در گام اول، مجموعه‌داده را روی گوگل‌درایو بارگذاری می‌کنیم. فراخوانی این فایل بدون نیاز به Mountکردن و استفاده از gdown در محیط گوگل‌کولب ممکن است با خطا مواجه شود که با استفاده از دستورات زیر مشکل حل می‌شود (منبع) .

```
!pip install --upgrade --no-cache-dir gdown
!gdown 1NRwDhqrhXF8tRy1HKe9QofJZjPBnHV1r
```

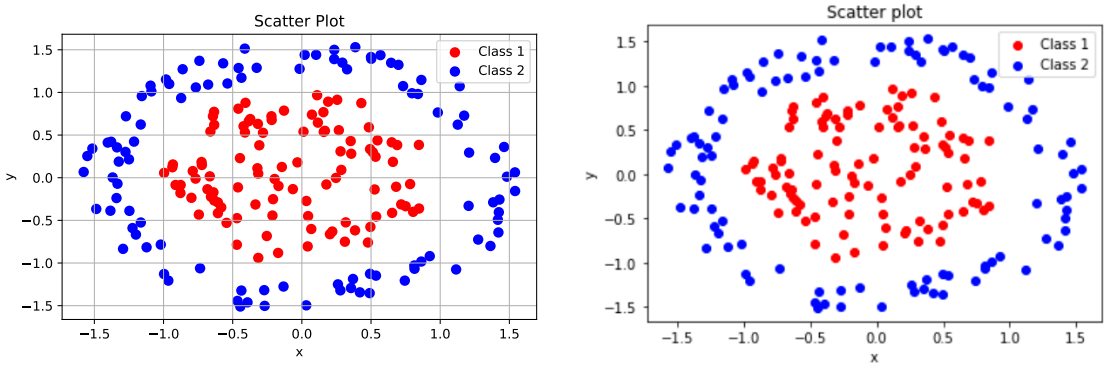
در ادامه، دستورات زیر را برای آماده‌سازی داده‌ها استفاده می‌کنیم. در این دستورات ابتدا فایل مربوطه را می‌خوانیم. با استفاده از پارامتر header=None مشخص می‌کنیم که عنوان و هدری در فایل وجود ندارد و سپس در قسمت names عناوین ستون‌ها را تعیین می‌کنیم که به‌ترتیب شامل x، y و label است. در ادامه، ترتیب سطرهای دیتافریم را به‌صورت تصادفی تغییر می‌دهیم. برای این منظور از تابع sample با پارامتر frac=1 استفاده شده است که به معنای استفاده از تمامی سطرهای دیتافریم به‌صورت تصادفی است. سپس، ستون‌های اول و دوم دیتافریم را انتخاب می‌کنیم و به‌عنوان دیتافریمی جدید و در قاب‌های جدید ذخیره می‌کنیم. در گام بعدی، ستون برچسب‌ها را دستخوش تغییراتی می‌کنیم. در الگوریتم آموزش مادالاین مقدار خروجی تابع فعال‌سازی همیشه یکی از دو مقدار +۱ و −۱ است. در این‌جا، برای آن‌که برای مقدار خروجی ۰ هم بتوانیم این تابع را به کار ببریم، مقدار آن را به −۱ تبدیل کرده‌ایم. در واقع با bipolarکردن خروجی، توانایی شبکه در تشخیص دو دسته را بهبود داده‌ایم.

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

df = df.sample(frac = 1)
df

data = df[['x','y']]
x = data.to_numpy()
x.shape[0]

# Convert to Bipolar
label = df[['label']]
t = label.to_numpy()
t[np.isclose(t, 0)] = -1

df0 = df.loc[df['label']  == 0 ]
df1 = df.loc[df['label']  == 1]
```

با آماده‌سازی داده‌ها دستورات زیر را برای رسم نمودار مورد نظر این سوال به کار می‌بریم و نتایج به صورتی است که در شکل ۱۱ آورده شده است. همان‌طور که مشاهده می‌شود هدف این قسمت از سوال برآورده شده و شکل‌ها هم‌خوان اند.

```
import matplotlib.pyplot as plt

```

```python
# 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()

# save the figure as PDF
plt.savefig("scatter_plot.pdf")

# show the plot
plt.show()
```



(ب) توزیع به‌دست‌آمده (آ) هدف صورت سوال

شکل ۱۱: نمونه نمودار پراکندگی دو دسته دادهٔ تعریف‌شده MadaLine در صورت سوال و پاسخ‌ها.

۳.۲.۱ قسمت ج

در این قسمت قصد داریم به پیاده‌سازی الگوریتم معرفی‌شده در قسمت ب بپردازیم و سپس با استفاده از آن داده‌های خود را در کلاس‌های متفاوت از هم تفکیک کنیم. در تمامی دستورات و توابع سعی کردیم اصل استفادهٔ برداری برای محاسبات بهینه را

لحاظ کرده‌ایم. تابع هزینه در هر اپاک را هم میانگین خطا روی تمامی نمونه‌ها در نظر گرفته‌ایم. اوزان اولیه را کوچک و اوران و باياس ثابت لایهٔ خروجی را به‌ترتیب یک و تعداد نورون‌های منفی‌یک در نظر گرفته‌ایم؛ چراکه این لایه منطق OR دارند و با این کار فقط هنگامی مقدار خروجی منفی می‌شود که تمامی نورون‌ها منفی‌یک شده باشند. ابتدا با استفاده از دستورات زیر و با استفاده از کتابخانهٔ pandas در پایتون، داده‌های یک دیتافریم را به صورت تصادفی مخلوط کرده، شناسه ردیف‌های آن را بازتنظیم کرده و نمونه جدیدی از دیتافریم بازگردانده می‌کند که در آن ترتیب ردیف‌ها به صورت تصادفی تغییر کرده است.

```
import matplotlib.pyplot as plt

# 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()

# save the figure as PDF
plt.savefig("scatter_plot.pdf")

# show the plot
plt.show()
```

سپس برنامهٔ ۸ را تعریف می‌کنیم. این تابع در پایتون، یک خط جداکننده بین دو دسته از داده‌ها را محاسبه می‌کند. برای این کار، ابتدا با استفاده از تابع `linspace` در کتابخانهٔ numpy، یک رشته از اعداد به تعداد `end.start` ایجاد می‌کند که در فاصله `start` تا `end` قرار دارند. سپس با استفاده از ضرایب `w` و `b` که به ترتیب وزن‌های مدل و عرض خط جداکننده هستند، معادلات خط را برای همه اعداد این رشته حساب می‌کند و نقاط مختلف خط را در یک رشته `y` ذخیره می‌کند. در نهایت، برای رسم خط جداساز، به ازای هر مقدار `x`، مقدار متناظر در رشته `y` را محاسبه کرده و جفت مرتب `(x, y)` را بازمی‌گرداند.

Program 8: Find Seperation Line

```
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
```

در ادامه مشابه قبل تابعی برای مقداردهی اولیه پارامترهایی مانند وزن و بایاس ایجاد می‌کنیم و یک متغیر هم برای کوچک‌سازی این مقادیر در نظر می‌گیریم. این تابع سه پارامتر ورودی به عنوان ورودی دریافت می‌کند. پارامتر اول عامل مقیاس برای وزن‌ها است، پارامتر دوم تعداد واحدهای ورودی و پارامتر سوم تعداد واحدهای خروجی مدل است. در ابتدای تابع، برای تولید نتایج قابل تکرار، یک سبد تصادفی مشخص تنظیم شده است. سپس، وزن‌ها و بایاس‌های مربوط به لایه‌های ورودی و خروجی ایجاد می‌شوند.

سپس، یک تابع فعال‌ساز ساده را تعریف می‌کنیم. این تابع یک ورودی دریافت می‌کند و برای آن یک تابع فعال‌سازی را اعمال می‌کند. تابع فعال‌سازی این کار را با استفاده از تابع `np.where` انجام می‌دهد. اگر ورودی بزرگتر یا مساوی با صفر باشد، تابع ۱ را برمی‌گرداند، در غیر این صورت تابع ۱- را برمی‌گرداند. در نهایت، خروجی تابع فعال‌سازی به عنوان خروجی تابع تعریف‌شده بازگردانده می‌شود.

```
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

```

```python
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
```

تابع مسیر پیش‌خور را نیز با هدف پیاده‌سازی یک لایهٔ پیشرو در شبکه‌ی عصبی تعریف می‌کنیم. این تابع به دو مرحله تقسیم می‌شود. در مرحلهٔ اول، اگر پرچم should_reshape تنظیم شده باشد، ورودی‌ها به یک آرایه ۲ در ۱ با مقدارهای مربوطه تغییر شکل می‌دهد. سپس در مرحلهٔ دوم، یک محاسبهٔ خطی روی ورودی‌ها با استفاده از وزن‌ها انجام می‌شود واضافه می‌شود (با استفاده از np.dot، ضرب داخلی ماتریسی بین وزن‌ها و بردار ورودی را محاسبه می‌کند و به بایاس اضافه می‌کند). سپس خروجی حاصل از تابع فعال‌سازی عبور داده می‌شود. در نهایت، تابع خروجی شامل دو مقدار است: ترکیب خطی بر روی ورودی‌ها با استفاده از وزن‌ها و بایاس و همچنین خروجی حاصل از اعمال تابع فعال‌سازی به ترکیب خطی:

```python
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
```

در ادامه تابعی برای به‌روزرسانی اوزان و بایاس‌ها تشکیل می‌دهیم. این تابع، وزن‌های شبکه عصبی را با توجه به خروجی شبکه و هدف مورد نظر، به روزرسانی می‌کند. برای این منظور، ورودی‌های شبکه، همراه با بایاس‌ها و وزن‌ها، و همچنین خروجی شبکه و مقدار ورودی آن به عنوان ورودی‌های تابع در نظر گرفته می‌شوند. در صورتی که خروجی شبکه با هدف یکسان باشد، وزن‌ها و بایاس‌ها بدون تغییر برگردانده می‌شوند. اگر خروجی شبکه با هدف متفاوت باشد، تابع ابتدا بررسی می‌کند که هدف مورد نظر، ۱ یا ۱- است. اگر هدف برابر با ۱ باشد و خروجی شبکه برابر با ۱- باشد، وزن و بایاس نورونی که بیشترین خروجی را دارد، برای تغییر مقدار به کار می‌رود. این تغییرات شامل یک مقدار اضافی به بایاس و یک ماتریس ضربی برای وزن‌ها است که در نهایت به بایاس‌ها و وزن‌ها اضافه می‌شود. اگر هدف برابر با ۱- باشد و خروجی شبکه برابر با ۱ باشد، وزن‌ها و بایاس‌ها به گونه‌ای تغییر می‌کنند که خروجی شبکه به سمت ۱- حرکت کند. برای این منظور، مقدار تفاضل بین مقدار واقعی خروجی و هدف با ضریب یادگیری ضرب شده و به بایاس‌ها و وزن‌ها اضافه می‌شود. در نهایت، وزن‌ها و بایاس‌های به‌روزرسانی شده به عنوان خروجی تابع برگردانده می‌شوند. هم‌چنین تابعی برای محاسبهٔ مقدار خطا یا اتلاف ایجاد می‌کنیم. این تابع برای محاسبهٔ خطا در پیش‌بینی یک مقدار خروجی مورد نظر از یک الگوریتم یادگیری به کار می‌رود. ورودی‌های این تابع به ترتیب شامل دو آرایه یا لیست با اندازهٔ یکسان می‌باشد. آرایه یا لیست اول، مقدار هدف یا مقدار واقعی را دربرمی‌گیرد، و آرایه یا لیست دوم، مقدار پیش‌بینی شده توسط الگوریتم یادگیری را شامل می‌شود. این تابع، با استفاده از فرمول محاسبه‌ی خطای میانگین مربعات، میان دو مقدار هدف و پیش‌بینی شده، خطای پیش‌بینی را محاسبه و به عنوان خروجی باز می‌گرداند.

```python
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
```

در انتها تابعی را برای پیش‌بینی تعریف می‌کنیم. این تابع مسئول پیش‌بینی خروجی یک مدل با ورودی‌ها و وزن‌ها و بایاس‌های مشخص است. این تابع ابتدا با اعمال ورودی به شبکه، مقدار خروجی را بدست می‌آورد. در ادامه این خروجی، وارد لایه دوم شبکه عصبی می‌شود و با اعمال وزن‌ها و انحراف‌های لایه دوم، خروجی نهایی را تولید می‌کند. این تابع با توجه به این که یک تابع پیش‌بینی است، مقدار پیش‌بینی‌شده را برای ورودی‌های مختلف محاسبه کرده و به صورت یک لیست از خروجی‌ها، برمی‌گرداند.

```python
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
```

در انتها تابع کلی الگوریتم آموزش را با توجه به توابعی که پیش‌تر تعریف کردیم، پیاده‌سازی می‌کنیم. در این تابع ابتدا مقادیر مورد نیاز آغازگری اولیه می‌شوند و شرایط توقف مشخص می‌شوند. سپس با توجه هم‌چنین، این تابع به صورتی نوشته شده است که در هر دوره، علاوه بر محاسبهٔ خطا، اتلاف و دقت، وزن و بایاس خطوط را هم برمی‌گرداند و در نهایت نتایج را در قالب نمودار خطا، ماتریس درهم‌ریختگی و نمایش داده‌ها به‌همراه خطوط جداکننده برمی‌گرداند.

```
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.savefig('error_plot.pdf')
                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
```

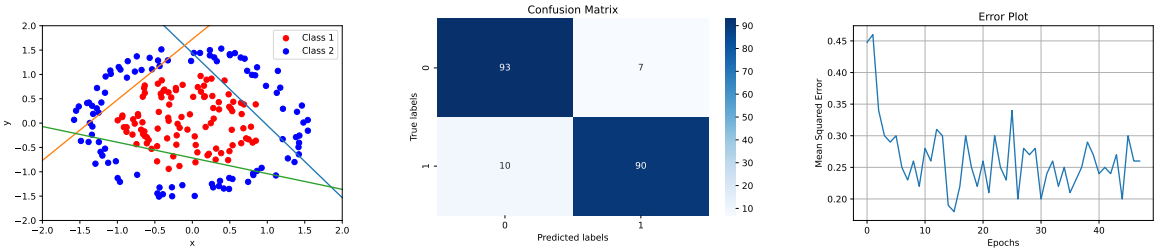
```
                for i in range(num_neurons_layer1):
                    px1, px2 = find_decision_boundary(-2, 2, weights[i], biases[i])
                    plt.plot(px1, px2)

                plt.xlabel("x")
                plt.ylabel("y")
                plt.legend(["Class 1" , "Class 2"])
                plt.xlim([-2, 2])
                plt.ylim([-2, 2])
                plt.savefig('error_plot1.pdf')
                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()
                # Save plot as PDF
                plt.savefig("confusion_matrix.pdf")
                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.savefig('error_plot.pdf')
        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.savefig('error_plot1.pdf')
        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()
        # Save plot as PDF
        plt.savefig("confusion_matrix.pdf")
        plt.show()
        predicted_output = predict(inputs, target,weights,biases,num_neurons_layer1)
        print(classification_report(target, predicted_output))

    return weights, biases, error_list
```

نتایج برای حالتی که از ۳، ۴ و ۱۰ نورون استفاده شده است، به‌ترتیب در شکل ۱۲، شکل ۱۴ و شکل ۱۶ نمایش داده شده است. در کلیهٔ نتایجِ قسمت ج که طول گام ذکر نشده طول گام بهینه عدد 0.0001 بوده و مثلاً در حالتی که از سه نورون استفاده شده اگر به‌جای استفاده از طول گام 0.0001 از طول گام بزرگ‌تری (0.01) استفاده کنیم نتایج به‌صورت شکل ۱۳ خواهد بود که نشان‌دهندهٔ افت نتایج است. حال اگر طول گام را برای ۴ نورون از 0.0001 به 0.0005 افزایش دهیم مشاهده می‌کنیم که نتایج به صورتی خواهد بود که در شکل ۱۵ نشان داده شده و ۴ نورون برای جداکردن کلاس‌های این داده‌ها کافی بوده است. به‌عنوان آخرین آزمایش اگر نرخ یادگیری (طول گام) را برای حالتی که از ۱۰ نورون استفاده می‌کنیم را به 0.0001 کاهش دهیم نتایج به صورتی خواهد شد که در شکل ۱۷ نمایش داده شده و همان‌طور که مشخص است بدتر خواهد شد.

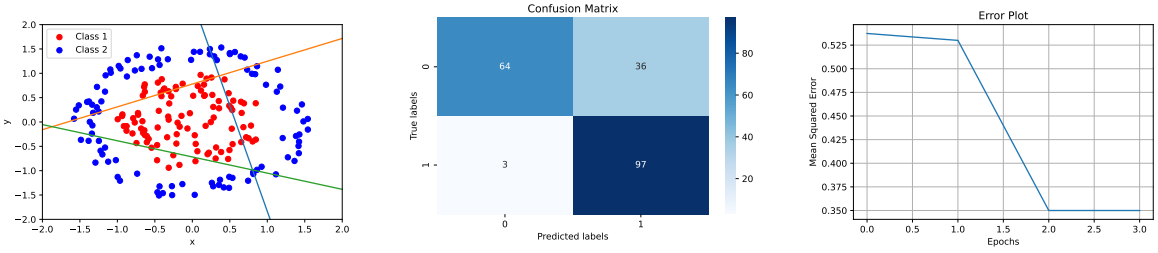
نتایج شاخصه‌ای مربوط به تمامی حالت‌های بالا در زیر آورده شده است:

```
# 3 neuron, lr = 0.0001:
              precision    recall  f1-score   support

        -1.0       0.90      0.93      0.92       100
         1.0       0.93      0.90      0.91       100

    accuracy                           0.92       200
```



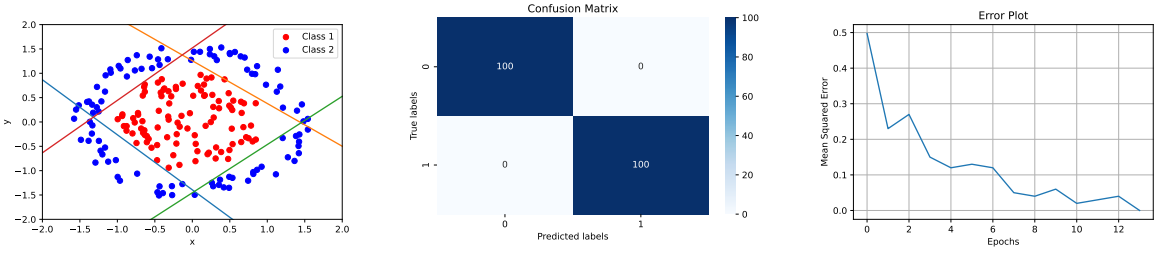
(آ) نمودار خطا (دقت) (ب) ماتریس درهم‌ریختگی (ج) نمایش خطوط

شکل ۱۲: نتایج مربوط به استفاده از الگوریتم مادالاین (۳ نورون).



(آ) نمودار خطا (دقت) (ب) ماتریس درهم‌ریختگی (ج) نمایش خطوط

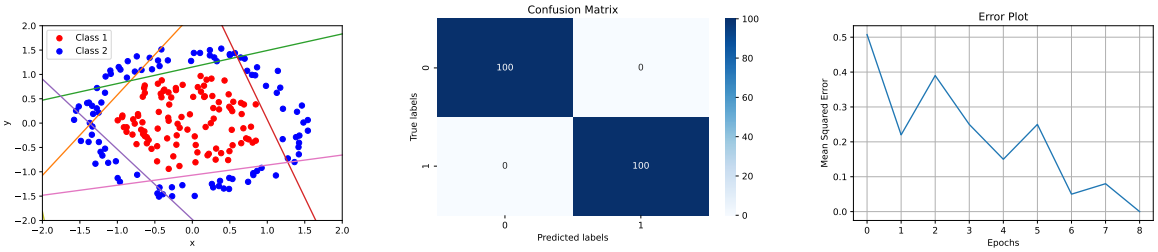
شکل ۱۳: نتایج مربوط به استفاده از الگوریتم مادالاین (۳ نورون و نرخ یادگیری 0.01).



(آ) نمودار خطا (دقت) (ب) ماتریس درهم‌ریختگی (ج) نمایش خطوط

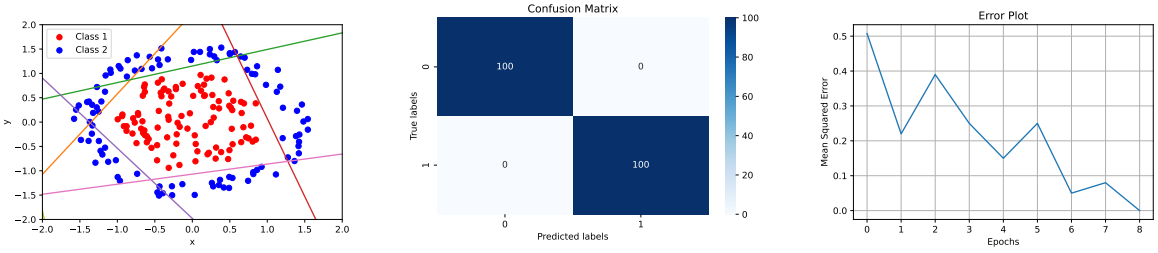
شکل ۱۴: نتایج مربوط به استفاده از الگوریتم مادالاین (۴نورون).

```
   macro avg       0.92      0.92      0.91       200
weighted avg       0.92      0.92      0.91       200

# 3 neuron, lr = 0.0001:
              precision    recall  f1-score   support

        -1.0       0.96      0.64      0.77       100
         1.0       0.73      0.97      0.83       100

    accuracy                           0.81       200
```



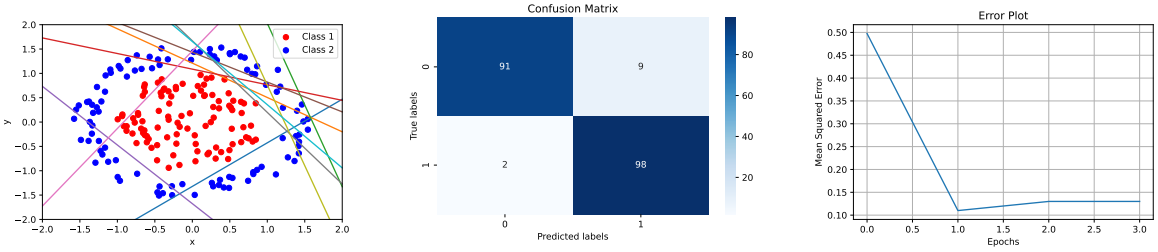
(آ) نمودار خطا (دقت) (ب) ماتریس درهم‌ریختگی (ج) نمایش خطوط

شکل ۱۵: نتایج مربوط به استفاده از الگوریتم مادالاین (۴ نورون و نرخ یادگیری 0.0005).



(آ) نمودار خطا (دقت) (ب) ماتریس درهم‌ریختگی (ج) نمایش خطوط

شکل ۱۶: نتایج مربوط به استفاده از الگوریتم مادالاین (۱۰ نورون و نرخ یادگیری 0.01).



(آ) نمودار خطا (دقت) (ب) ماتریس درهم‌ریختگی (ج) نمایش خطوط

شکل ۱۷: نتایج مربوط به استفاده از الگوریتم مادالاین (۱۰ نورون و نرخ یادگیری 0.0001).

```
   macro avg       0.84      0.80      0.80       200
weighted avg       0.84      0.81      0.80       200

# 4 neuron, lr = 0.0001:
              precision    recall  f1-score   support

        -1.0       1.00      0.97      0.98       100
         1.0       0.97      1.00      0.99       100

    accuracy                           0.98       200
```

```
   macro avg       0.99      0.98      0.98       200
weighted avg       0.99      0.98      0.98       200

# 4 neuron, lr = 0.0005:
              precision    recall  f1-score   support

        -1.0       1.00      1.00      1.00       100
         1.0       1.00      1.00      1.00       100

    accuracy                           1.00       200
   macro avg       1.00      1.00      1.00       200
weighted avg       1.00      1.00      1.00       200

# 3 neuron, lr = 0.0001:


# 10 neuron, lr = 0.01:
              precision    recall  f1-score   support

        -1.0       1.00      1.00      1.00       100
         1.0       1.00      1.00      1.00       100

    accuracy                           1.00       200
   macro avg       1.00      1.00      1.00       200
weighted avg       1.00      1.00      1.00       200

# 10 neuron, lr = 0.0001:
              precision    recall  f1-score   support

        -1.0       0.98      0.91      0.94       100
         1.0       0.92      0.98      0.95       100

    accuracy                           0.94       200
   macro avg       0.95      0.95      0.94       200
weighted avg       0.95      0.94      0.94       200
```

۴.۲.۱ قسمت د

افزایش تعداد نورون در الگوریتم مادالاین ممکن است منجر به بهبود تفکیک کردن کلاس‌ها و افزایش دقت شود، اما این به صورت قطعی نیست و بستگی به داده‌ها و مسئله دارد. در برخی موارد، افزایش تعداد نورون‌ها می‌تواند منجر به بیش‌برازش شود و باعث بدتر شدن عملکرد الگوریتم شود. با توجه به این که مادالاین از الگوریتم‌های یادگیری ماشین است، تغییر طول گام ممکن است تاثیر بسیار زیادی در عملکرد الگوریتم داشته باشد. در صورتی که طول گام بسیار کوچک باشد، الگوریتم به سرعت می‌تواند به حداقل محلی همگرایی کند و در نتیجه به دقت بالاتری برسد، اما به صورت همزمان زمان اجرای الگوریتم نیز افزایش

پیدا خواهد کرد. از طرف دیگر، اگر طول گام بسیار بزرگ باشد، الگوریتم ممکن است به سرعت به نقطه همگرایی برسد، اما این همراه با یک دقت پایین‌تر است. بنابراین، برای بهبود عملکرد الگوریتم، باید تلاش کنیم تا بهترین مقدار طول گام را برای هر مسئله مشخص کنیم.

در مسألهٔ ما، با افزایش تعداد نورون‌ها، مدل بهتر می‌تواند خطوط جداکننده را پیدا کند و سریع‌تر می‌تواند خطا را کاهش دهد. هم‌چنین، در حالتی که تعداد نورون‌ها بیش‌تر است به نوعی سختی کار روی خطوط بیش‌تری توزیع می‌شود و نیاز به تغییر وزن کم‌تری هست. تمامی نتایج و شکل‌های آورده‌شده در قسمت ج با تنظیم فراپارامترها و در بهترین حالت خود به‌دست آمده اند. علاوه بر این‌ها، با توجه به تعیین شروط مختلف توقف آموزش، تعداد ایپاک طی‌شده برای رسیدن به حد مناسب خطا در نمودارهای مربوط به تابع اتلاف در قسمت ج آورده شده‌اند. مشاهده می‌شود که هرچه تعداد نورون بیش‌تر باشد آزادی عمل الگوریتم بیش‌تر بوده و در ایپاک کم‌تری به نتیجهٔ مطلوب دست پیدا می‌کند. مثلاً مشاهده می‌شود که در یک مقایسه با طول گام یکسان، با سه نورون این اتفاق در حدود ۵۰ ایپاک رخ می‌دهد؛ درحالی‌که، با ۴ نورون این اتفاق در تعداد ایپاک کم‌تری رخ می‌دهد. بنابراین، هرچه تعداد نورون بیش‌تر باشد تعداد تکرار لازم احتمالاً کم‌تر است. طول گام انتخاب‌شده نیز تأثیرگذار است. هم‌چنین در شکل ۱۶ مشاهده می‌شود که تنها از ۵ خط برای جداسازی کامل کلاس‌ها استفاده شده است که این به‌معنای عدم نیاز به نورون (خط) بیش‌تر و به‌روزنشدن آن‌هاست.