به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق

تمرین سری اول

نام و نام خانوادگی	محسن فیاض
شماره دانشجویی	**810100524**
تاریخ ارسال گزارش	**جمعه، 20 اسفند 1400**

فهرست گزارش سوالات

سوال 1 – Mcculloch Pitts

ابتدا کلاسی برای نورون Mcculloch Pitts می‌نویسیم تا با استفاده از آن، شبکه مورد نیاز را بسازیم.

```python
class McCullochPitts:
    def __init__(self, pos_weight, neg_weight, excitatory_list, activation_threshold=0):
        """
        A connection path is excitatory if the weight on the path is positive; other-
        wise it is inhibitory. All excitatory connections into a particular neuron have
        the same weights. (Laurene V. Fausett p27)
        """
        self.pos_weight = pos_weight
        self.neg_weight = neg_weight
        self.activation_threshold = activation_threshold
        self.excitatory_list = excitatory_list

    def activation_function(self, g):
        """
        The activation of a McCulloch-Pitts neuron is binary. That is, at any time
        step, the neuron either fires (has an activation of 1) or does not fire (has an activation of 0).
        Each neuron has a fixed threshold such that if the net input to the neuron
        is greater than the threshold, the neuron fires. (Laurene V. Fausett p26, 27)
        """
        return g >= self.activation_threshold

    def forward(self, inputs: list):
        assert(len(inputs) == len(self.excitatory_list))
        sum = 0
        for input, excitatory in zip(inputs, self.excitatory_list):
            if excitatory:
                sum += input * self.pos_weight
            else:
                sum += input * self.neg_weight
        return self.activation_function(sum)
```

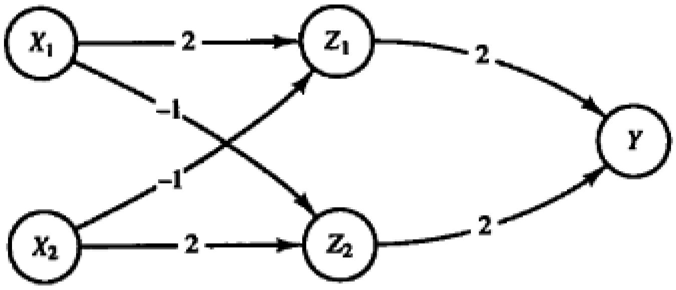
کد 1 - نورون Mcculloch Pitts

این کد بر اساس توضیحات صفحه 26 و 27 کتاب نوشته شد. از قصد به شکلی نوشته شد که محدودیت موجود در این نورون را اعمال کند و فقط بتوان یک وزن مثبت و یک وزن منفی داشت. با همین نورون می‌توان گیت های منطقی بسیاری را ساخت که چند نمونه از آن را ساختیم.

```python
class AndGate(McCullochPitts):
    def __init__(self):
        super().__init__(pos_weight=1, neg_weight=0, excitory_list=[1, 1], activation_threshold=2)

class OrGate(McCullochPitts):
    def __init__(self):
        super().__init__(pos_weight=2, neg_weight=0, excitory_list=[1, 1], activation_threshold=2)

class XorGate():
    def forward(self, inputs: list):
        assert(len(inputs) == 2)
        z1 = McCullochPitts(pos_weight=2, neg_weight=-1, excitory_list=[1, 0],
                            activation_threshold=2).forward(inputs)
        z2 = McCullochPitts(pos_weight=2, neg_weight=-1, excitory_list=[0, 1],
                            activation_threshold=2).forward(inputs)
        y = McCullochPitts(pos_weight=2, neg_weight=0, excitory_list=[1, 1],
                           activation_threshold=2).forward([z1, z2])
        return y
```

کد 2 - گیت‌های منطقی

نمایش نورون این گیت‌ها و وزنهای مورد نظر و ترشولد آن در زیر آمده است. این گیت‌ها پایه ساخت باقی مدار هستند و در ادامه دیگر هر کدام از اینها را با نماد منطقی آن نمایش می‌دهیم و طبیعتا می‌توان به جایش نمایش نورونی را گذاشت.



شکل 1 - شبکه Mcculloch Pitts گیت XOR

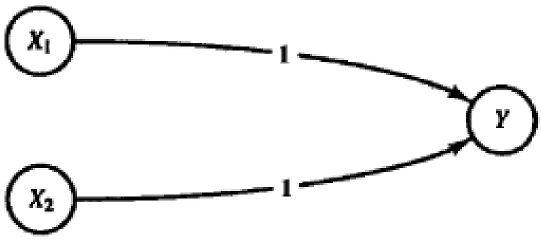


Figure 1.14 A McCulloch-Pitts neuron to perform the logical AND function.

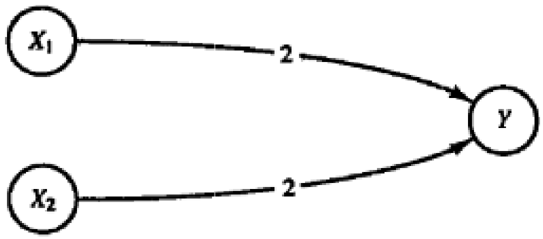


Figure 1.15 A McCulloch-Pitts neuron to perform the logical OR function.

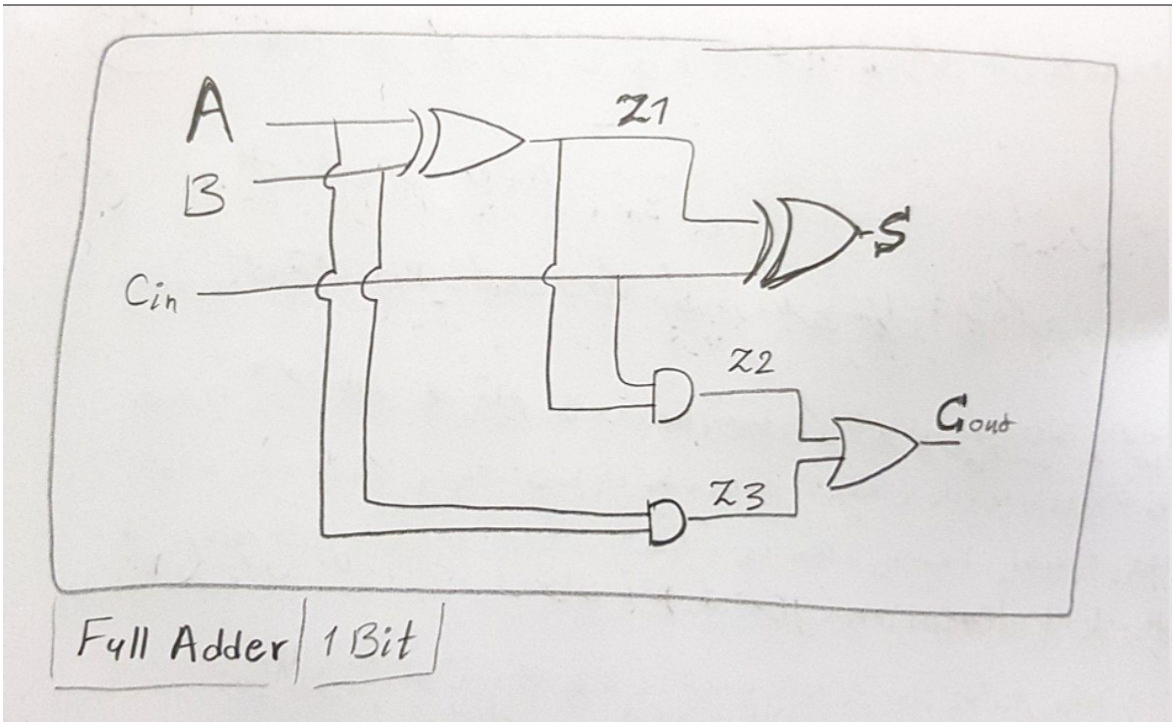
شکل 2 - شبکه Mcculloch Pitts گیت OR و AND

ابتدا طبق https://www.geeksforgeeks.org/full-adder-in-digital-logic و تصویر Full Adder logic circuit یک Full Adder با استفاده از گیت های XOR و AND و OR که بالاتر تعریف کردیم می‌سازیم.

```
def full_adder_1bit(A, B, Cin=0):
    z1 = XorGate().forward([A, B])
    S = XorGate().forward([z1, Cin])
    z2 = AndGate().forward([z1, Cin])
    z3 = AndGate().forward([A, B])
    C = OrGate().forward([z2, z3])
    return S, C
```

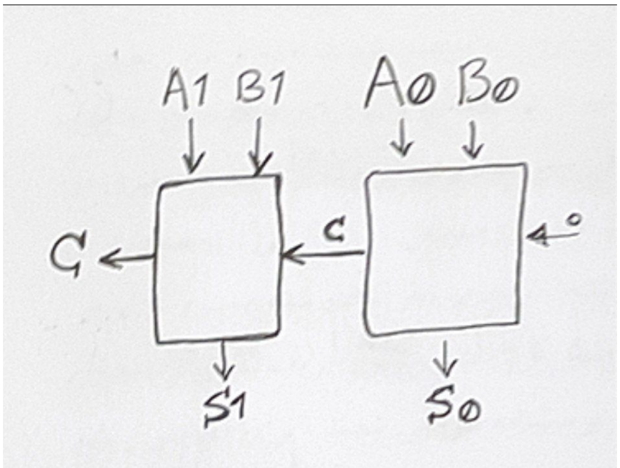
کد 3 - فول ادر یک بیتی

ساختار این Full Adder یک بیتی در زیر آمده است. هر کدام از گیت‌های استفاده شده بالاتر به شکل نورونی نشان داده شدند. در ادامه برای ساخت نسخه دوبیتی نیز از این بخش استفاده می‌کنیم.



شکل 3 - شبکه Full Adder یک بیتی

برای ساخت نسخه دوبیتی به شکل زیر، دو نسخه یک بیتی را کنار هم قرار می‌دهیم. ترتیب ورودی و خروجی‌ها نیز در شکل مشخص است.

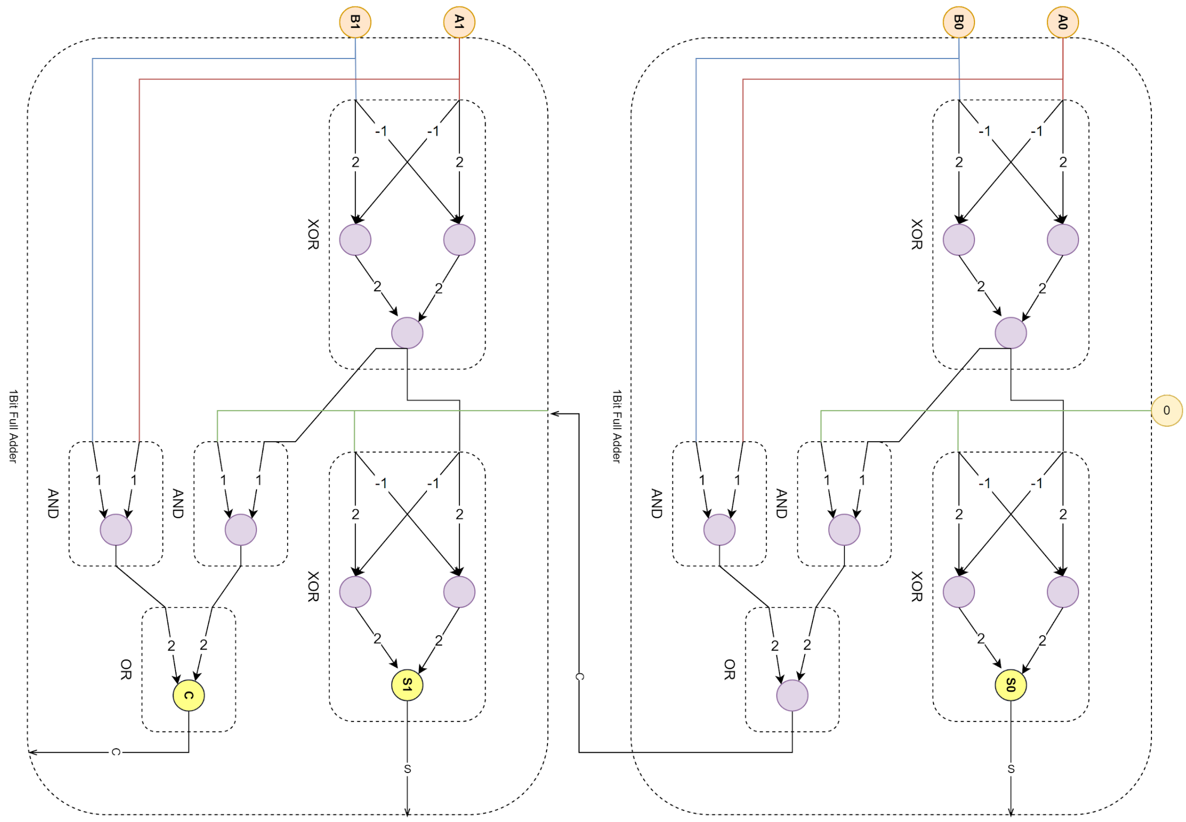


شکل 4 - شبکه Full Adder دو بیتی

```python
def full_adder_2bit(A1, A0, B1, B0, Cin=0):
    S0, Cmid = full_adder_1bit(A0, B0)
    S1, C = full_adder_1bit(A1, B1, Cmid)
    return S1, S0, C
```

کد 4- فول ادر دو بیتی

در آخر اگر مراحل کشیده شده را جایگذاری کنیم به شکل کامل زیر می‌رسیم. در این شکل threshold همه نورون ها 2 در نظر گرفته شده است.



شکل 5 - شبکه کامل Full Adder دو بیتی

و در آخر شبیه‌سازی را اجرا می‌کنیم و Truth Table را به دست می‌آوریم.

```
import itertools
binaries = list(itertools.product([0, 1], repeat=4))
print(" A1 A0 B1 B0 ->C S1S0")
for b in binaries:
    S1, S0, C = full_adder_2bit(*b)
    print(b, " ", C*1, S1*1, S0*1)

### Result ###
A1 A0 B1 B0 -> C S1S0
(0, 0, 0, 0)   0 0 0
(0, 0, 0, 1)   0 0 1
(0, 0, 1, 0)   0 1 0
(0, 0, 1, 1)   0 1 1
(0, 1, 0, 0)   0 0 1
(0, 1, 0, 1)   0 1 0
(0, 1, 1, 0)   0 1 1
(0, 1, 1, 1)   1 0 0
(1, 0, 0, 0)   0 1 0
(1, 0, 0, 1)   0 1 1
(1, 0, 1, 0)   1 0 0
(1, 0, 1, 1)   1 0 1
(1, 1, 0, 0)   0 1 1
(1, 1, 0, 1)   1 0 0
(1, 1, 1, 0)   1 0 1
(1, 1, 1, 1)   1 1 0
```

کد 5- اجرای شبیه‌سازی و مشاهده خروجی

در این بخش ابتدا تمام حالتهای ورودی را ساختیم و سپس خروجی را به ازای هر کدام چاپ کردیم که مشخص است این کار به درستی انجام شده است.

سوال ۲ – Adaline

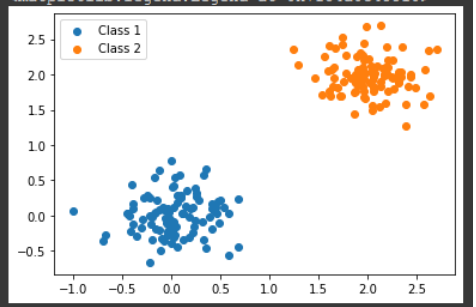
الف)

برای تعریف دو دسته داده از np.random.normal استفاده می‌کنیم که داده‌های رندوم که از توزیع نرمال با مشخصات توضیح داده شده انتخاب شده‌اند به ما بر می‌گرداند.

a) Data Distribution + Scatter Plot

```python
import matplotlib.pyplot as plt
import numpy as np


c1_x = np.random.normal(loc=0, scale=0.3, size=100)
c1_y = np.random.normal(loc=0, scale=0.3, size=100)
c2_x = np.random.normal(loc=2, scale=0.3, size=100)
c2_y = np.random.normal(loc=2, scale=0.3, size=100)
plt.scatter(x=c1_x, y=c1_y, label="Class 1")
plt.scatter(x=c2_x, y=c2_y, label="Class 2")
plt.legend()
```

<matplotlib.legend.Legend at 0x7f54a0c49910>



شکل 6 - نمودار پراکندگی دو دسته داده تعریف شده

ب)

ابتدا طبق توضیحات کتاب، Adaline را پیاده سازی می‌کنیم.

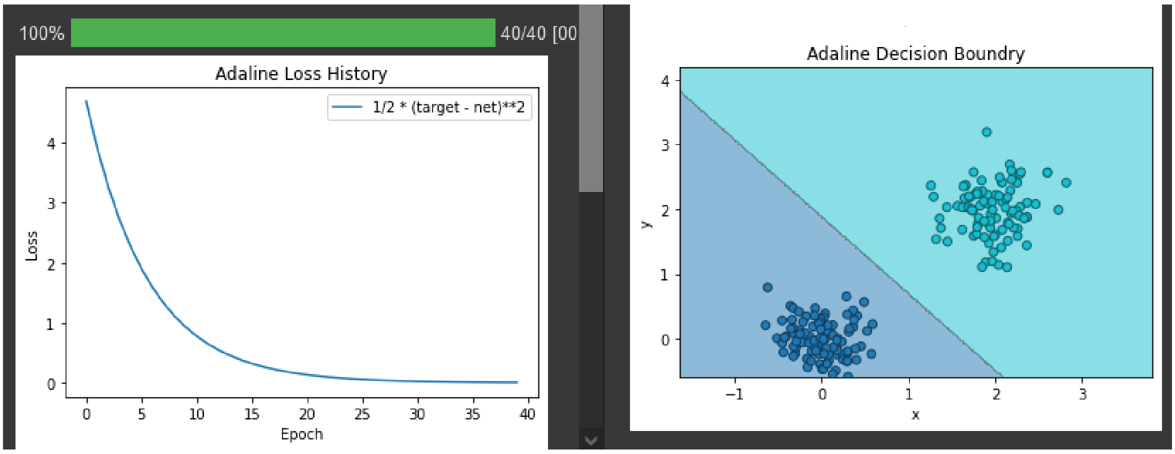
```python
class Adaline():
    def __init__(self, in_features=2, lr=0.2, stop_tolerance=1e-10):
        """
        Initialize weights. (Small random values are usually used.)
        Set learning rate alpha (Laurene V. Fausett p82)
        """
        self.weights = np.random.random(in_features)
        self.bias = np.random.random()
        self.lr = lr
        self.stop_tolerance = stop_tolerance
        self.loss_history = []

    def forward(self, xi):
        """Compute net input to output unit: y_in = b + E xi.wi """
        net = self.bias + np.dot(xi, self.weights)
        return net

    def step(self, xi, net, target):
        """
        Update bias and weights, i = 1, . . . , n:
        b(new) = b(old) + a(t -- y_in).
        wi(new) = wi(old) + a(t -- y_in)xi . (y_in=net)
        """
        grad = target - net
        self.bias += self.lr * grad
        self.weights += self.lr * grad * xi
        return max(abs(self.lr * grad), max(abs(self.lr * grad * xi)))

    def calc_loss(self, targets, nets):
        return np.mean(1/2 * (np.array(targets) - np.array(nets))**2)

    def train(self, X, Y, max_epochs=10):
        """If the largest weight change that occurred in Step 2 is
        smaller than a specified tolerance, then stop; otherwise continue.
        """
        self.loss_history = []
        max_change = 10e10
        from sklearn.utils import shuffle
        X, Y = shuffle(X, Y, random_state=0)
        for epoch in tqdm(range(max_epochs)):
            targets, nets = [], []
            for x, y in zip(X, Y):
                net = self.forward(x)
                step_max_change = self.step(x, net, y)
                max_change = min(max_change, step_max_change)
                targets.append(y)
                nets.append(net)
            self.loss_history.append(self.calc_loss(targets, nets))
            if max_change < self.stop_tolerance:
                print(max_change, self.stop_tolerance)
                break

    def activation_function(self, g):
        return 1 if g >= 0 else -1

    def predict(self, X):
        Y = []
        for x in X:
            net = self.forward(x)
            Y.append(self.activation_function(net))
        return np.array(Y)

    def plot_loss(self):
        plt.plot(self.loss_history, label="1/2 * (target - net)**2")
        plt.legend()
        plt.title("Adaline Loss History")
```

```python
        plt.xlabel("Epoch")
        plt.ylabel("Loss")
        plt.show()

    def plot_scatter(self, X, Y):
        plt.scatter(
            X[:, 0],
            X[:, 1],
            c=Y,
            cmap='tab20b',
            edgecolors="k",
            alpha=0.95,
        )
        h = 0.01  # step size in the mesh
        xx, yy = np.meshgrid(np.arange(min(X[:, 0]) - 2, max(X[:, 0]), h) + 1,
                             np.arange(min(X[:, 1]) - 1, max(X[:, 1]), h) + 1)
        # Plot the decision boundary
        Z = self.predict(np.c_[xx.ravel(), yy.ravel()])
        # Put the result into a color plot
        Z = Z.reshape(xx.shape)
        cm = plt.cm.tab20b
        plt.contourf(xx, yy, Z, cmap=cm, alpha=0.5)
        plt.title("Adaline Decision Boundry")
        plt.xlabel("x")
        plt.ylabel("y")
        plt.show()
```

کد 6- پیاده‌سازی Adaline

همچنین پس از مرحله آموزش که وزنهای w و bias آموزش دیدند توابعی نوشته شد برای اینکه نمودار تغییرات خطا و همچنین مرز تصمیم مدل آموزش دیده مشخص شود.

در ادامه نتیجه اجرا آمده است.

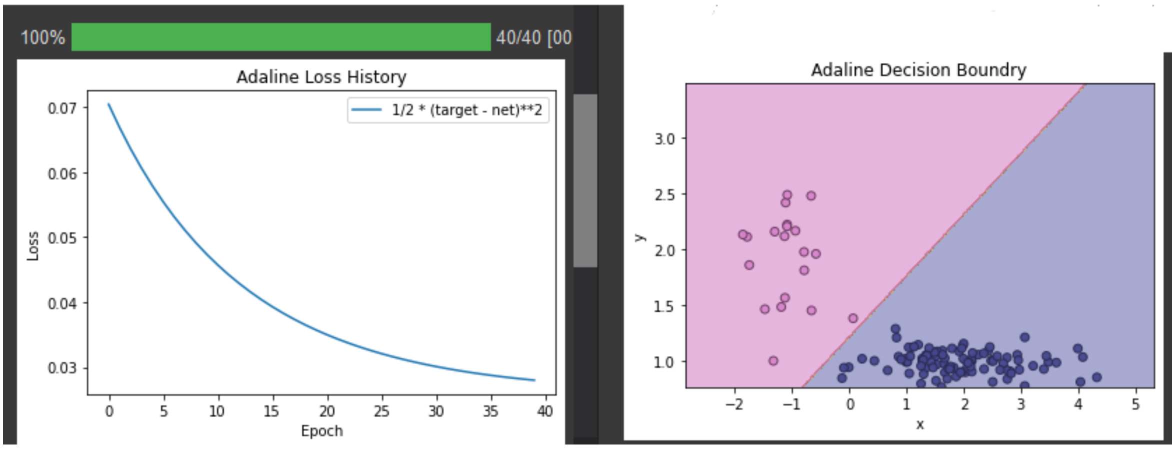


شکل 7 - نمودار خطا و مرز تصمیم مدل Adaline

همانطور که دیده می‌شود و با توجه به توضیحات درس می‌توان دید که حرکت در جهت گرادیان خطا باعث شده است که در طول زمان خطا کاسته شود. و در آخر می‌بینیم که مرز تصمیم مناسبی که توانسته تمام داده‌ها را به خوبی جداسازی کند ایجاد شده است. این کار به علت اینکه مدل به درستی کار می‌کند و همچنین داده به صورت خطی قابل جداسازی بوده است اتفاق افتاده.

ج)

در این بخش با مشخصه های جدید دو دسته داده را ایجاد کرده و مرحله قبل را تکرار کردیم.



شکل 8 - نمودار خطا و مرز تصمیم مدل Adaline با داده جدید

در این حالت هم می‌بینیم که با وجود سخت تر شدن مرزبندی و نزدیک شدن دو کلاس باز هم Adaline توانست مرز خوبی بین این دو کلاس پیدا کند.البته طبق رندوم بودن داده‌ها و همچنین رندوم initialization وزن‌ها ممکن است این اتفاق همواره رخ ندهد.

سوال Madaline – 3

الف)

روش MRI که روش اولیه آموزش است تنها وزن‌های لایه پنهان را تغییر می‌دهد و وزنهای لایه خروجی ثابت هستند. اما در MRII تمام وزن‌ها تغییر می‌کنند. در این سوال از روش MRI استفاده می‌کنیم و لایه خروجی صرفا عمل OR منطقی را انجام می‌دهد و تنها وزنهای بخش میانی یا hidden آموزش می‌بینند. طبق صفحه 90 کتاب تنها زمانی وزنهای مورد نظر آپدیت می‌شوند که خطا رخ داده باشد. در این حالت اگر مدل باید 1 می‌داده ولی -1 داده است، چون خروجی را OR گذاشتیم پس تمام نورون های مخفی خروجی منفی 1 داشته‌اند و برای اصلاح این موضوع آن که از همه به 0 نزدیک تر بوده را آپدیت میکنیم تا خروجی 1 بسازد و نتیجه 1 شود. بالعکس اگر مدل باید -1 می‌داده ولی 1 داده است، باید تمامی نورون‌های مخفی آپدیت شوند تا به زیر 0 بروند تا OR آنها -1 که خروجی مطلوب است را بسازد.

ب)

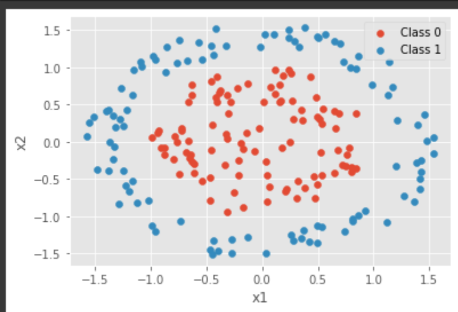
نتیجه بارگذاری و کشیدن نمودار داده‌ها در ادامه آمده است.

```
[29]
df = pd.read_csv("Question3.csv", header=None)
df
```

	0	1	2
0	0.459694	-0.470583	0.0
1	0.797385	-0.343030	0.0
2	0.235270	0.961296	0.0
3	0.765453	-0.177644	0.0
4	-0.335577	-0.313893	0.0
...	...	...	...
195	0.744066	-1.206548	1.0
196	-0.457547	1.286227	1.0
197	-1.020000	-0.783926	1.0
198	1.363429	-0.800250	1.0
199	-1.246702	-0.388615	1.0

200 rows × 3 columns

```
with plt.style.context("ggplot"):
    plt.scatter(x=df.loc[df[2] == 0][0],
                y=df.loc[df[2] == 0][1], label="Class 0")
    plt.scatter(x=df.loc[df[2] == 1][0],
                y=df.loc[df[2] == 1][1], label="Class 1")
    plt.xlabel("x1")
    plt.ylabel("x2")
    plt.legend()
```



شکل 9 - بارگذاری و نمایش داده‌ها

ج)

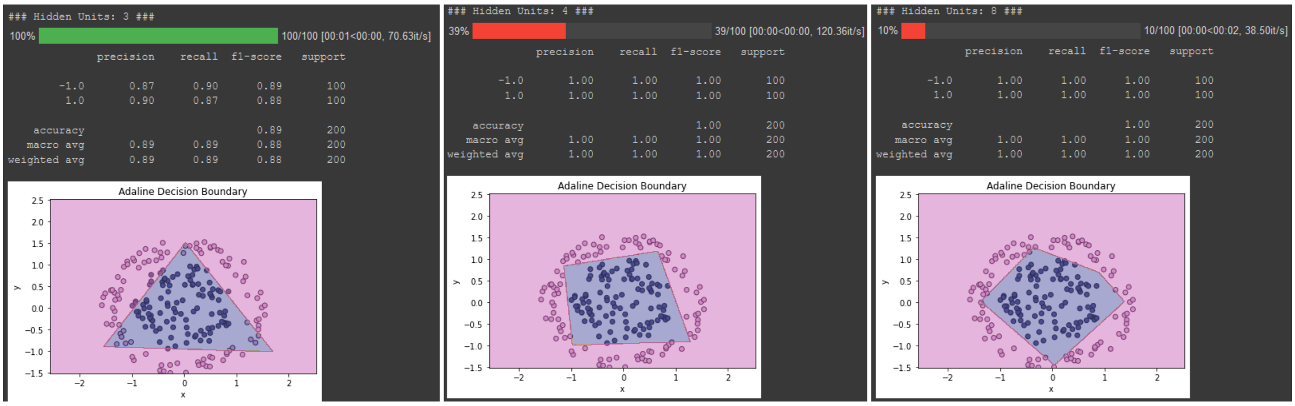
کد پیاده سازی طبق توضیحات کتاب در ادامه آمده است. باید توجه داشت که برای OR آخر بسته به تعداد نورون لایه پنهان باید وزن های لایه آخر را مقداردهی کرد.

```python
class Madaline:
    def __init__(self, in_features=2, hidden_units=3, lr=0.2, stop_tolerance=1e-10):
        """
        Initialize weights:
        Weights v i and v2 and the bias b3 are set as described;
        small random values are usually used for ADALINE weights.
        """
        self.weights = np.random.random([in_features, hidden_units])
        self.output_weights = np.array([1/hidden_units] * hidden_units)
        self.bias = np.random.random(hidden_units)
        self.output_bias = (hidden_units - 1)/hidden_units
        self.lr = lr
        self.stop_tolerance = stop_tolerance
        self.loss_history = []

        self.hidden_units = hidden_units
        self.in_features = in_features

    def activation_function(self, g):
        return ((g >= 0) - 0.5) * 2  # [0,1] -> [-1, 1]

    def forward(self, x):
        """Compute net input to each hidden ADALINE unit
        Determine output of each hidden ADALINE unit
        Determine output of net
        """
        x = x.reshape(1, len(x))  # (1, i)
        z_in = self.bias + x @ self.weights  # (1, i)@(i, h)->(1, h)
        z = self.activation_function(z_in)
        y_in = self.output_bias + z @ self.output_weights # (1, h)@(h, 1)->(1)
        y = self.activation_function(y_in)[0]

        return y, z_in, y_in

    def predict(self, X):
        Y = []
        for x in X:
            y, z_in, y_in = self.forward(x)
            Y.append(y)
        return np.array(Y)

    def eval(self, X, Y):
        preds = []
        for x, t in zip(X, Y):
            y, z_in, y_in = self.forward(x)
            preds.append(y)
        from sklearn.metrics import classification_report
        print(classification_report(Y, preds))

    def step(self, y, z_in, t, x):
        z_in = z_in[0]  # flatten
        if t != y:  # If t = y, no weight updates are performed
            if t == 1:  # If t = 1, then update weights on Zj, the unit whose net input is closest to 0
                close_zero_idx = np.argmin(abs(z_in))
                self.weights[:, close_zero_idx] += self.lr * (1 - z_in[close_zero_idx]) * x
                self.bias[close_zero_idx] += self.lr * (1 - z_in[close_zero_idx])
            elif t == -1:  # If t = --1, then update weights on all units Zk that have positive net input
                for h in range(self.hidden_units):
                    if z_in[h] > 0:
                        self.weights[:, h] += self.lr * (-1 - z_in[h]) * x
                        self.bias[h] += self.lr * (-1 - z_in[h])

    def train(self, X, Y, max_epochs=40):
        """If the largest weight change that occurred in Step 2 is
```

```python
        smaller than a specified tolerance, then stop; otherwise continue.
        """
        self.loss_history = []
        from sklearn.utils import shuffle
        X, Y = shuffle(X, Y, random_state=42)
        for epoch in tqdm(range(max_epochs)):
            # self.plot_scatter(X, Y)
            targets, nets = [], []
            for x, t in zip(X, Y):
                y, z_in, y_in = self.forward(x)
                self.step(y, z_in, t, x)
                targets.append(t)
                nets.append(y_in)
            self.loss_history.append(self.calc_loss(targets, nets))

    def calc_loss(self, targets, nets):
        return np.mean(1/2 * (np.array(targets) - np.array(nets))**2)

    def plot_loss(self):
        plt.plot(self.loss_history, label="1/2 * (target - net)**2")
        plt.legend()
        plt.title("Adaline Loss History")
        plt.xlabel("Epoch")
        plt.ylabel("Loss")
        plt.show()

    def plot_scatter(self, X, Y):
        plt.scatter(
            X[:, 0],
            X[:, 1],
            c=Y,
            cmap='tab20b',
            edgecolors="k",
            alpha=0.95,
        )
        h = 0.01  # step size in the mesh
        xx, yy = np.meshgrid(np.arange(min(X[:, 0]) - 2, max(X[:, 0]), h) + 1,
                             np.arange(min(X[:, 1]) - 1, max(X[:, 1]), h) + 1)
        # Plot the decision boundary
        Z = self.predict(np.c_[xx.ravel(), yy.ravel()])
        # Put the result into a color plot
        Z = Z.reshape(xx.shape)
        cm = plt.cm.tab20b
        plt.contourf(xx, yy, Z, cmap=cm, alpha=0.5)
        plt.title("Adaline Decision Boundary")
        plt.xlabel("x")
        plt.ylabel("y")
        plt.show()

for n in [3, 4, 8]:
    print(f"### Hidden Units: {n} ###")
    madaline = Madaline(in_features=2, hidden_units=n, lr=0.1)
    madaline.train(X=np.array(df[[0, 1]]), Y=np.array(df[2]), max_epochs=80)
    madaline.plot_loss()
    madaline.eval(X=np.array(df[[0, 1]]), Y=np.array(df[2]))
    madaline.plot_scatter(X=np.array(df[[0, 1]]), Y=np.array(df[2]))
```

کد 7- پیاده‌سازی Madaline

و در ادامه نتیجه اجرای کد برای سه حالت خواسته شده آمده است.



شکل 10 - نتیجه اجرای Madaline

د)

طبق نتایج قسمت قبل:

با 3 نورون پنهان:

- 100 ایپاک
- 89% دقت

با 4 نورون پنهان:

- 39 ایپاک
- 100% دقت

با 8 نورون پنهان:

- 10 ایپاک
- 100% دقت

همانطور که دیده می‌شود دقت با افزایش تعداد نورون پنهان که در اینجا معادل اضافه کردن خط جدا کننده است افزایش می‌یابد و از 4 نورون می‌توانیم کاملا دو کلاس را جدا کنیم.

از نظر ایپاک نیز هر چه نورون بیشتر باشد زودتر به نتیجه می‌رسیم و ایپاک کمتری نیاز است. علت این موضوع این است که با نورون بیشتر راحت تر می‌توانیم این داده را یاد بگیریم و هر چه نورون (خط جدا کننده) کمتر باشد باید با تلاش بیشتر محدوده خاصی که آن خطهای کم قرار گیرند را پیدا کرد که بیشتر طول می‌کشد. و در مورد 3 نورون چون داده با 3 خط قابل جداسازی

نیست هیچگاه به دقت کامل نمی‌رسیم. همچنین افزایش نورون‌ها باعث نرم تر شدن مرزها و در نتیجه generalization بهتر در این مورد می‌شود که با نورون‌های کمتر نمی‌توان انتظار داشت.

سوال 4 – Perceptron

طبق صفحه 61 کتاب ادامه می‌دهیم.

نرخ یادگیری = 0.3

x1,x2,x3 = 0,1,1

t = -1

Update Rule: $If\ y \neq t$: $w_i := w_i + \alpha t x_i$ and $b := b + \alpha t$

جدول 1 - تغییرات وزنها

w1	w2	w3	bias	$y_{in} = b + \Sigma x_i w_i$	y	αt
0.2	0.7	0.9	-0.7	-0.7+0.7+0.9 = 0.9	1	-0.3
0.2	0.4	0.6	-1.0	-1.0+0.4+0.6 = 0	1	-0.3
0.2	0.1	0.3	-1.3	-1.3+0.1+0.3 = -0.9	-1	-0.3
				چون y = t شد دیگر وزن‌ها تغییر داده نمی‌شوند		