به نام خدا

دانشگاه تهران
دانشکده مهندسی برق و کامپیوتر

شبکه های عصبی و یادگیری عمیق

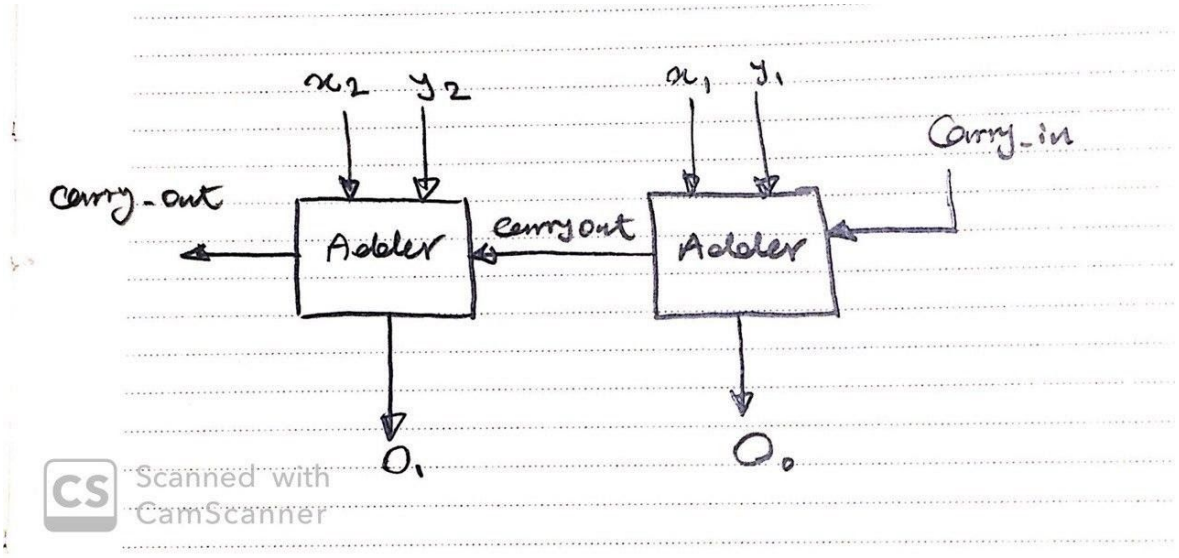
تمرین سری .....

نام و نام خانوادگی	فاطمه حقیقی
شماره دانشجویی	810195385
تاریخ ارسال گزارش	1398/12/16

فهرست گزارش سوالات

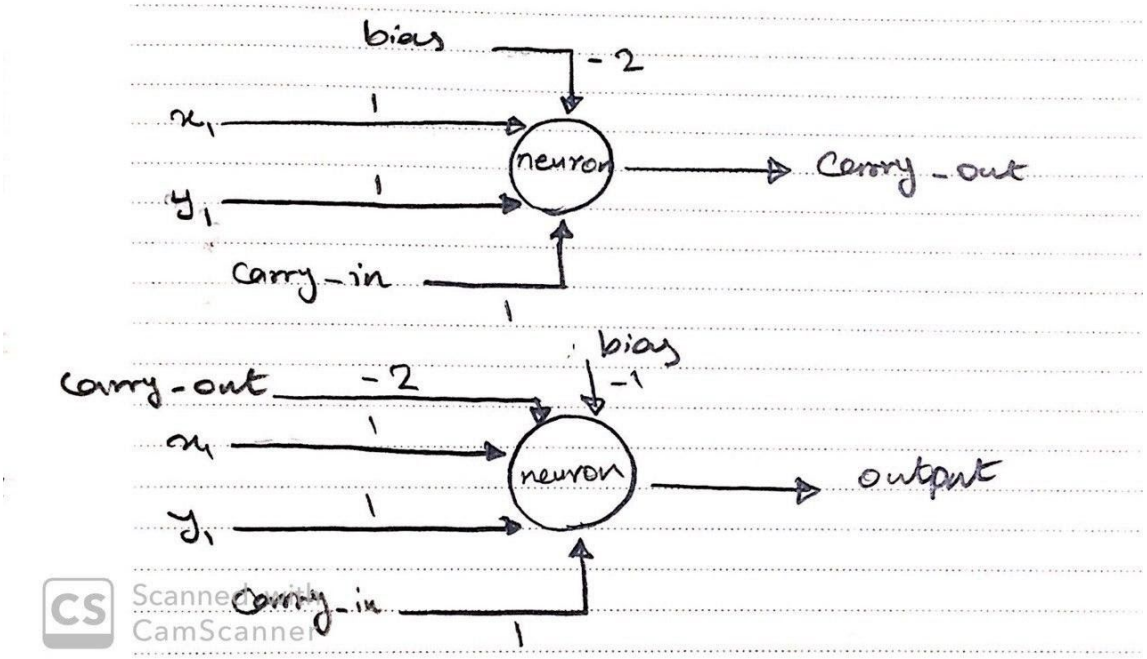
سوال 1 – طراحی full-adder با McCulloch-Pitts

همانطور که می دانیم،ساختار full adder برای دو بیت به صورت زیر می باشد:



شکل۱: طراحی full adder دو بیتی

شماتیک کلی برای پیاده سازی یک full adder دو بیتی در این سوال به صورت زیر است:



شکل۲: طراحی network برای full adder دو بیتی

پیاده سازی یک نورون McCulloch-Pitts به زبان پایتون به صورت زیر است:

```python
class Mac_pit_neuron():
    def __init__(self, input_value, weight_value):
        self.inputs = input_value
        self.weights = weight_value

    def calculate_output(self):
        out = 0
        for i in range(len(self.inputs)):
            out = out + self.inputs[i] * self.weights[i]
        if out >= 0 :
            return 1
        else:
            return 0
```

شکل۳: پیاده سازی نورون McCulloch-Pitts

پس از آن لازم است ساختار adderرا پیاده یازی کنیم که در آن دو نورون وجود دارد، یکی برای محاسبه ی caryy out و دیگری برای محاسبه ی output. پیاده سازی این ساختار به زبان پایتون به صورت زیر است:

```python
class Adder():
    def __init__(self, input1, input2, carry_in, bias):
        self.input1 = input1
        self.input2 = input2
        self.carry_in = carry_in
        self.bias = bias

    def set_input_value(self, in1, in2, carry):
        self.input1 = in1
        self.input2 = in2
        self.carry_in = carry

    def calculate_output(self):
        carry_neuron = Mac_pit_neuron([self.input1, self.input2, self.carry_in, self.bias], [1, 1, 1, -2])
        carry_neuron_output = carry_neuron.calculate_output()
        output_neuron = Mac_pit_neuron([self.input1, self.input2, self.carry_in, carry_neuron_output, self.bias]
                                       ,[1, 1, 1, -2, -1])
        return output_neuron.calculate_output(), carry_neuron_output
```

شکل۴: پیاده سازی adder

در مرحله ی بعد لازم است به وسیله ی adder ساختار full adderرا پیاده سازی کنیم که با توجه به تعداد بیت ها، full adder لازم را بسازد. پیاده سازی این ساختار به زبان پایتون به صورت زیر است:

```python
class Full_adder():
    def __init__(self, bit_number):
        self.bit_number = bit_number
        self.neurons = []

    def calculation(self, input1, input2):
        self.input1 = input1
        self.input2 = input2
        for i in range(self.bit_number):
            self.neurons.append(Adder(0, 0, 0, 1))
        carry = 0
        out = []
        for i in range(self.bit_number):
            out.append(0)
        for i in range(self.bit_number):
            self.neurons[i].set_input_value(input1[i], input2[i], carry)
            out[i] , carry = self.neurons[i].calculate_output()
        return out, carry
```

شکل۵: پیاده سازی full adder

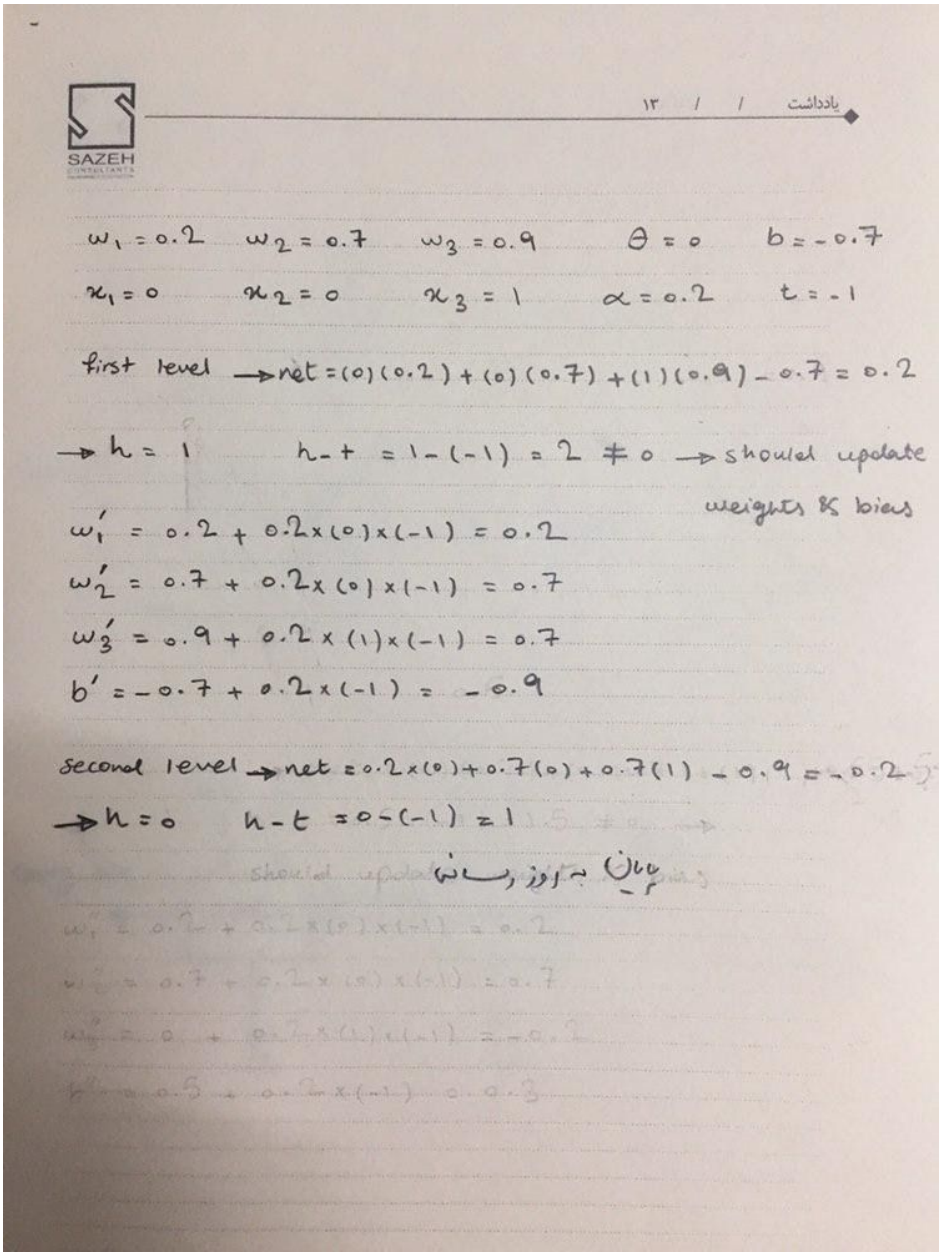
پس از اجرای کد و نمایش دادن نتیجه، حاصل محاسبات به صورت زیر خواهد بود، که در آن مقدار یک بیتی سمت راست carry out آن محاسبه است و مقدار دو بیتی خروجی جمع می باشد:

```
1   [0, 0] + [0, 0]  =  [0, 0] 0
2   [0, 0] + [0, 1]  =  [0, 1] 0
3   [0, 0] + [1, 0]  =  [1, 0] 0
4   [0, 0] + [1, 1]  =  [1, 1] 0
5   [0, 1] + [0, 0]  =  [0, 1] 0
6   [0, 1] + [0, 1]  =  [1, 0] 0
7   [0, 1] + [1, 0]  =  [1, 1] 0
8   [0, 1] + [1, 1]  =  [0, 0] 1
9   [1, 0] + [0, 0]  =  [1, 0] 0
10  [1, 0] + [0, 1]  =  [1, 1] 0
11  [1, 0] + [1, 0]  =  [0, 0] 1
12  [1, 0] + [1, 1]  =  [0, 1] 1
13  [1, 1] + [0, 0]  =  [1, 1] 0
14  [1, 1] + [0, 1]  =  [0, 0] 1
15  [1, 1] + [1, 0]  =  [0, 1] 1
16  [1, 1] + [1, 1]  =  [1, 0] 1
17
```

شکل۶: نتیجه ی اجرای full adder دو بیتی بر روس تمامی اعداد ممکن

سوال ۲ – به روزرسانی دستی شبکه ی perceptron

حل تشریحی این سوال به صورت زیر است:



شکل۷: به روز رسانی دستی شبکه ی perceptron

سوال 3 – پیاده سازی، بررسی و مقایسه عملکرد دو شبکه perceptronو Adeline

پیاده سازی کلاس های linear perceptron، perceptron_neuron و Adeline perceptron به صورت زیر است:

```python
class Perceptron_neuron():
    def __init__(self, weights, a, bias, epoch, function):
        self.weights = weights
        self.bias = bias
        self.a = a
        self.out = 0
        self.epoch = epoch
        self.update_counter = 0
        self.function = function

    def update_rule(self, inputs, t):
        pass

    def get_weights(self):
        return self.weights

    def get_bias(self):
        return self.bias

    def get_net_value(self, inputs):
        out = 0
        for i in range(len(inputs)):
            out += self.weights[0] * inputs[i]
            out += self.weights[1] * inputs[i]
        out += self.bias
        self.out = out
        return out

    def get_h_value(self, instance):
        return self.function(self.get_net_value(instance))

    def fit(self, inputs, target):
        if self.update_counter >= self.epoch:
            return 1
        else:
            outputs = []
            for instance in inputs:
                output = self.get_h_value(instance)
                outputs.append(output)
            return np.array_equal(output, target)
```

شکل۸: پیاده سازی کلاس perceptron_neuron

```python
class Linear_perceptron(Perceptron_neuron):
    def __init__(self, weights, a, bias, epoch, function):
        super().__init__(weights, a, bias, epoch, function)

    def update_rule(self, inputs, t):
        self.update_counter += 1
        for i in range(len(inputs)):
            h = self.get_h_value(inputs[i])
            if h - t[i] != 0:
                self.bias = self.bias + self.a * t[i]
                self.weights = self.weights + self.a * inputs[i] *t[i]
```

شکل۹: پیاده سازی کلاس linear_perceptron

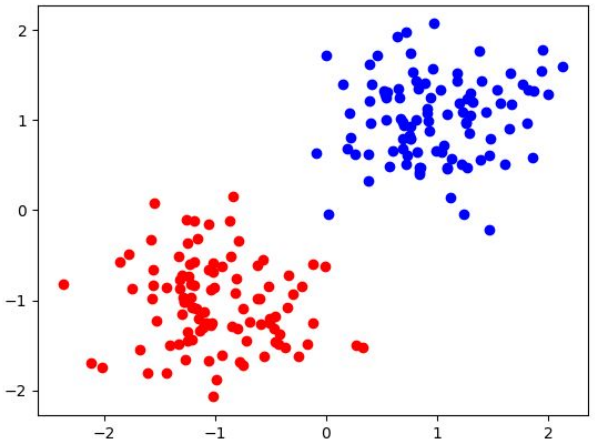
```python
class Adeline_perceptron(Perceptron_neuron):
    def __init__(self, weights, a, bias, epoch, function):
        super().__init__(weights, a, bias, epoch, function)

    def update_rule(self, inputs, t):
        self.update_counter += 1
        for i in range(len(inputs)):
            net = self.get_net_value(inputs[i])
            h = self.get_h_value(inputs[i])
            if h - t[i] != 0:
                self.bias = self.bias + self.a * (t[i] - net)
                self.weights = self.weights + self.a * inputs[i] * (t[i] - net)
```

شکل۱۰: پیاده سازی کلاس adeline_perceptron

همانطور که مشخص است،کلاس perceptron_neuron یک کلاس abstract است که کلاس های linear_perceptron و Adeline_perceptron از آن ارث می برند و هر کدام update_rule خود را overwrite می کنند.

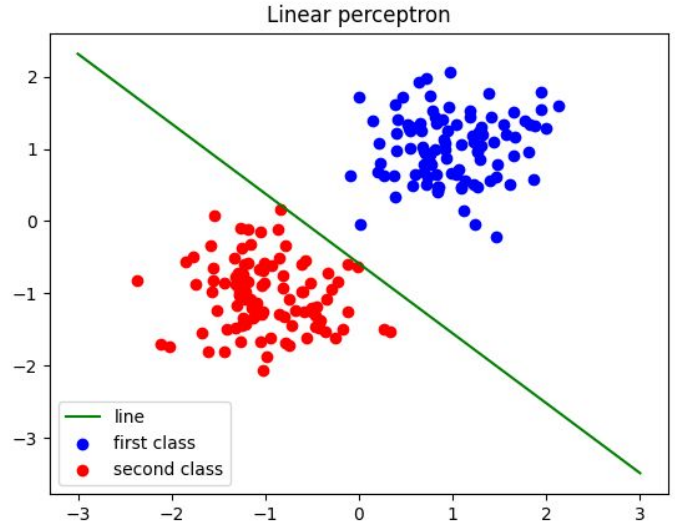
پاسخ سوالات برای مجموعه ی اول:

قسمت آ) تصویر این مجموعه داده در حالت عادی به صورت زیر است:

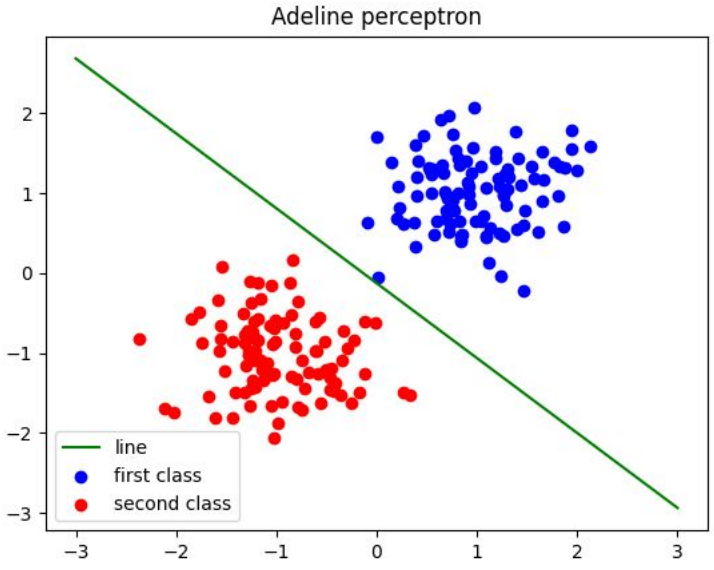


شکل۱۱:نمونه دای از نقاط درون هر دسته در مجموعه ی اول

خطوط جداساز برای مجموعه ی اول در حالت Linear perceptron و Adeline perceptron به صورت زیر است:



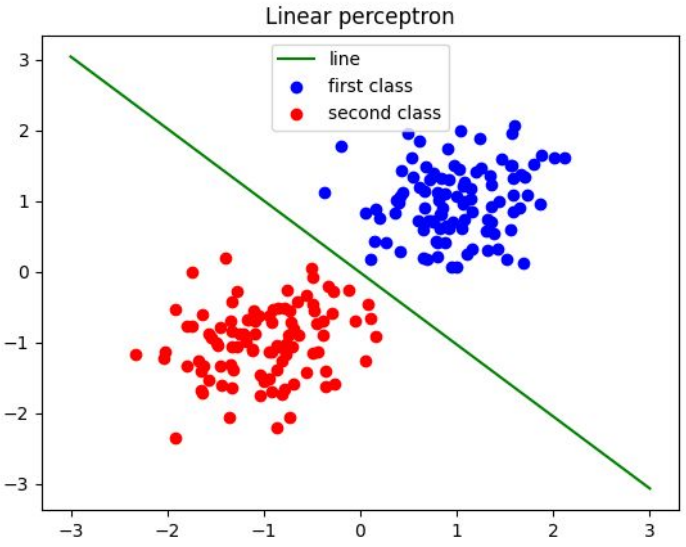
شکل۱۲: نمونه ای از نقاط درون دسته ی اول در حالت linear perceptron



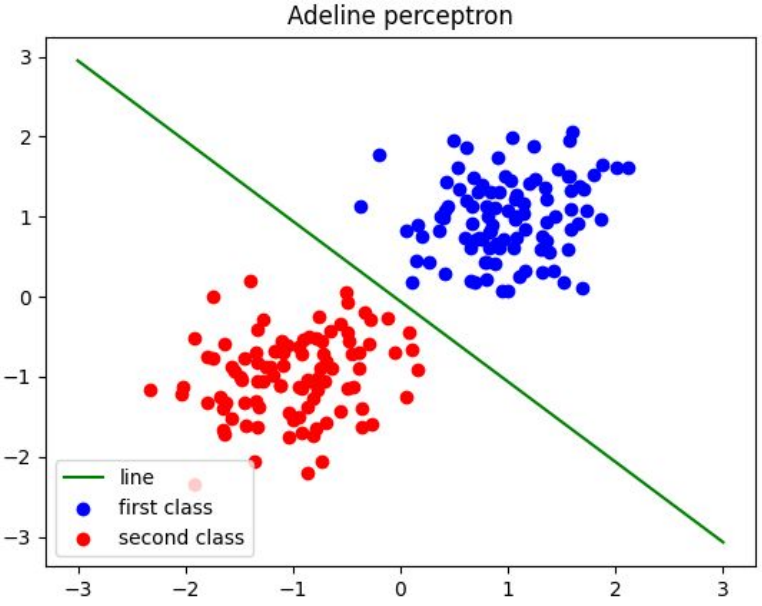
شکل۱۳: نمونه ای از نقاط درون دسته ی اول در حالت adeline perceptron

قسمت ب) به طور کلی می توان گفت در روش perceptron در صورتی که به یک جواب برسیم،الگوریتم متوقف می شود. در روش adeline perceptron در صورتی که یک شر پایان مناسب داشته باشیم،جداسازی بین دو کلاس با بیشترین margin صورت می گیرد.

قسمت ج) در نمونه های بالا a = 0.2 می باشد. در صورتی که a مقداری بزرگ باشد، در روش linear perceptron می توانیم یک جواب بیابیم اما در روش adeline perceptron الگوریتم converge نمی کند. می توان گفت در حالتی که a = 0.01 باشد الگوریتم adeline perceptronنیز converge خواهد کرد.



شکل۱۴: نمونه ای از نقاط درون دسته ی اول در حالت linear perceptron با a = 0.01

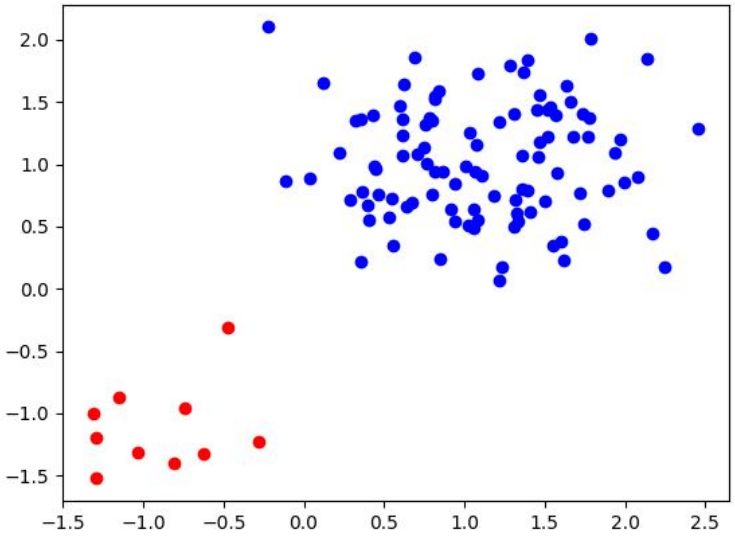


شکل۱۵: نمونه ای از نقاط درون دسته ی اول در حالت adeline perceptron با a = 0.01

به طور کلی زمان همگرایی به عواملی از جمله learning rate و threshhold وابسته است. در شرایطی که learning rate یکسان داشته باشیم، مدت زمان همگرایی adeline perceptron بیشتر خواهد بود زیرا این الگوریتم در جستجوی بهترین پاسخ می باشد.

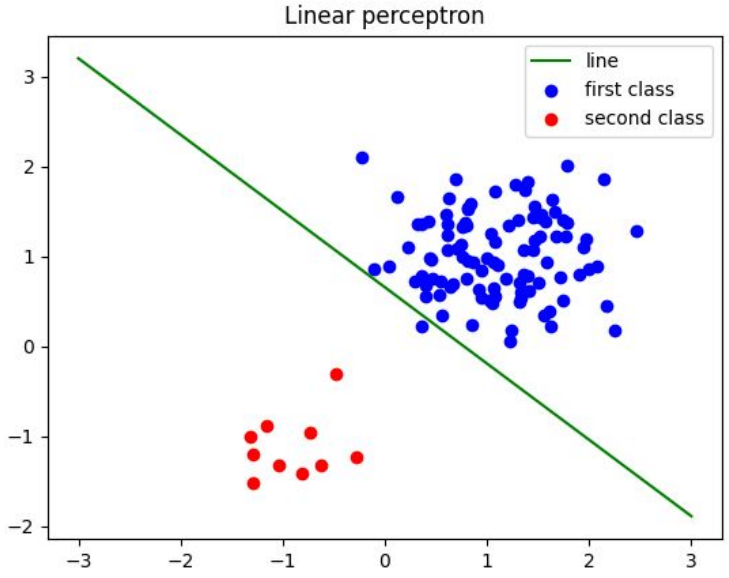
پاسخ سوالات برای مجموعه ی دوم:

قسمت آ)تصویر این مجموعه داده در حالت عادی به صورت زیر است:

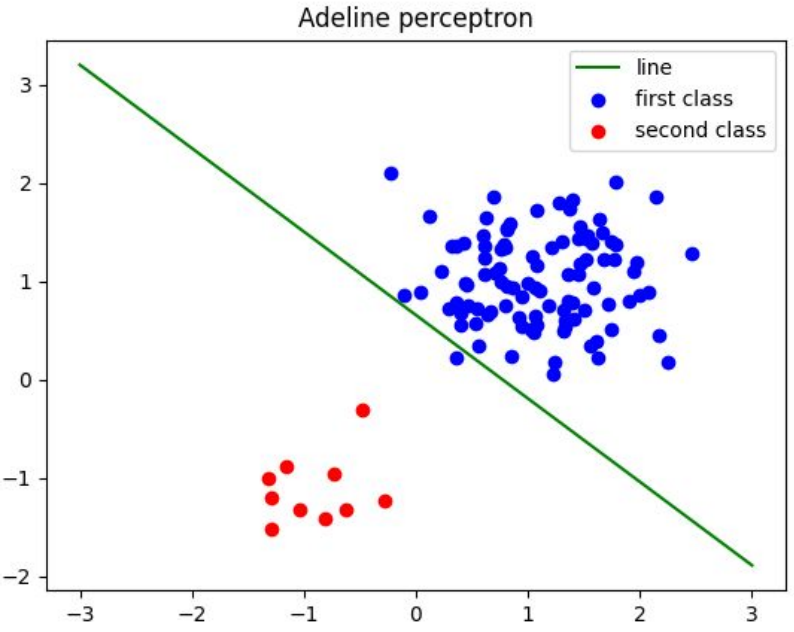


شکل۱۶:نمونه دای از نقاط درون هر دسته در مجموعه ی دوم

خطوط جداساز برای مجموعه ی دوم در حالت Linear perceptron و Adeline perceptron به صورت زیر است:



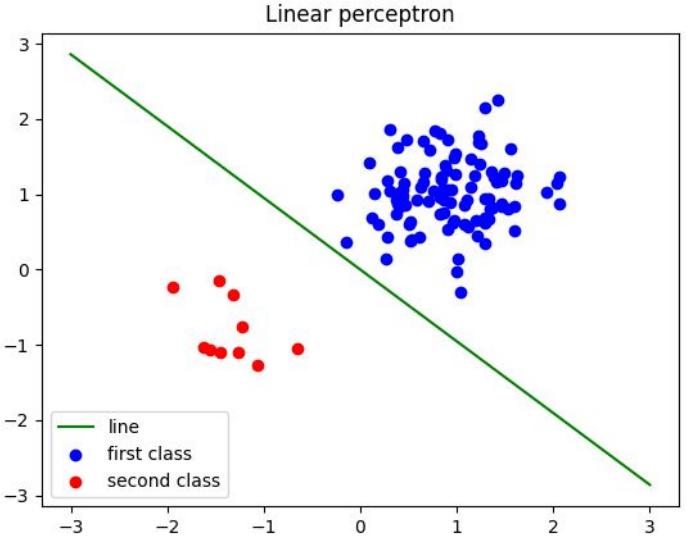
شکل۱۷: نمونه ای از نقاط درون دسته ی دوم در حالت linear perceptron



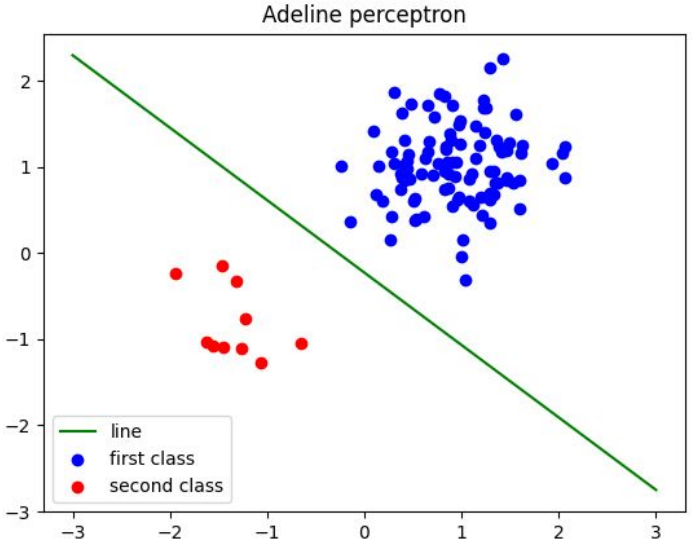
شکل۱۸: نمونه ای از نقاط درون دسته ی دوم در حالت adeline perceptron

قسمت ب) در صورتی که برابر بودن توزیع داده ها و وجود یک learning rate مناسب ، می توان گفت در این روش الگوریتم adeline perceptron همگرا می شود . با توجه به اینکه همچین شرایطی نداریم، نمی توان در باره ی همگرایی این الگوریتم به طور قطعی نظر داد.

قسمت ج) در این بخش نیز مانند مجموعه ی قبل، در صورتی که a مقداری بزرگ باشد، در روش linear perceptron می توانیم یک جواب بیابیم اما در روش adeline perceptron الگوریتم converge نمی کند. می توان گفت در حالتی که a = 0.5 باشد الگوریتم adeline perceptronنیز converge خواهد کرد.



شکل۱۹: نمونه ای از نقاط درون دسته ی دوم در حالت adeline perceptron با a = 0.5



شکل۲۰: نمونه ای از نقاط درون دسته ی دوم در حالت adeline perceptron با a = 0.5

به طور کلی زمان همگرایی به عواملی از جمله learning rate و threshhold وابسته است. در شرایطی که learning rate یکسان داشته باشیم، مدت زمان همگرایی adeline perceptron بیشتر خواهد بود زیرا این الگوریتم در جستجوی بهترین پاسخ می باشد.