به نام خدا

درس: یادگیری عمیق

استاد: دکتر فاطمی‌زاده

گزارش تمرین عملی شماره ۱

سیّدمحمّدامین منصوری طهرانی

۹۴۱۰۵۱۷۴

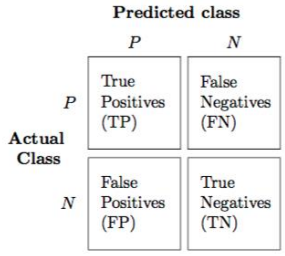
*توجه: لطفاً **فایل داده‌ای که پیوست شده(digits.csv)** را در کنار سایر کدها گذاشته و سپس آن‌ها را اجرا کنید. برای کار با داده‌ها، در سطر اول فایل، برای هر ستون نام وارد کرده‌ام.(feature_i) در غیر این‌صورت کد قابل اجرا نخواهد بود.

از بدست آوردن نتیجه الگوریتم GD در این‌جا خودداری کرده و صرفاً نتیجه‌ها را بیان می‌کنیم.

توضیح کلی کد: فایل‌های پیوست شده شامل یک کد main.py هستند که با اجرای آن تمام نتایج نمایش داده می‌شوند. در کنار این فایل کد کلاس پرسپترون برای هر ۴ تابع هزینه نوشته شده و قرار داده شده‌است. در ابتدای کد نیز این کلاس‌ها import می‌شوند. سپس فایل دیتا مطابق با تغییری که در ابتدای صفحه گفته شد لود می‌شود. شماره دانشجویی من ۹۴۱۰۵۱۷۴ بوده و رقم‌های من ۱ و ۴ خواهند بود و برای داشتن قابلیت adapt شدن به اعداد دیگر این دو رقم به صورت پارامتر تعیین می‌شوند. سپس اندیس داده‌هایی که برچسب‌شان هر یک از این دو عدد است را می‌یابیم و ۸۰ درصد آن‌ها را به عنوان ترین و ۲۰ درصد دیگر را به عنوان تست قرار می‌دهیم. سپس اندیس کل داده‌هایی که به عنوان ترین و تست انتخاب کردیم را مشخص می‌کنیم و در دو متغیر ریخته‌ایم.(train_indexes و test_indexes) بخش اول این اندیس‌ها عدد اول و بخش دوم عدد دوم را شامل می‌شوند. سپس برای ورودی دادن به طبقه‌بند آرایه برچسب‌ها را می‌سازیم.(در ادامه توضیح داده می‌شود چرا لازم است.) ضمناً اگر عدد اول بود به آن برچسب ۱ و عدد دوم را برچسب ۰ می‌زنیم.(این فقط یک نگاشت برای راحتی محاسبات است.) بعد از آن نیز ۱۶ ویژگی موجود برای هر داده‌ی train را به نحوی که برای ورودی طبقه‌بند مقرر است، آماده می‌کنیم و در training_inputs می‌ریزیم. سپس هر یک از ۴ پرسپترون را فراخوانده و شئ‌ای با ۱۶ ورودی (این نحوه انتخاب پس از توضیح کد طبقه‌بند مشخص می‌شود.) برای هر کدام می‌سازیم. بعد از آن هر ۴ شئ را با training_inputs و labels (که نگاشت به ۰ و ۱ یافته) آموزش می‌دهیم. ۱۶ ویژگی و برچسب آن‌ها را برای داده‌های تست نیز آماده می‌کنیم.(مطابق قبل، در دو متغیر test_inputs و test_labels ذخیره می‌شوند.)

در یک حلقه نیز، نتیجه پیش‌بینی شئ‌های طبقه‌بند برای داده‌های تست در ۴ آرایه result_i ذخیره می‌شود. در ادامه نیز درصد دقت هر مدل با تقسیم تعداد پیش‌بینی‌های درست از داده‌های تست به تعداد کل داده‌های تست تعیین شده و چاپ می‌شود.

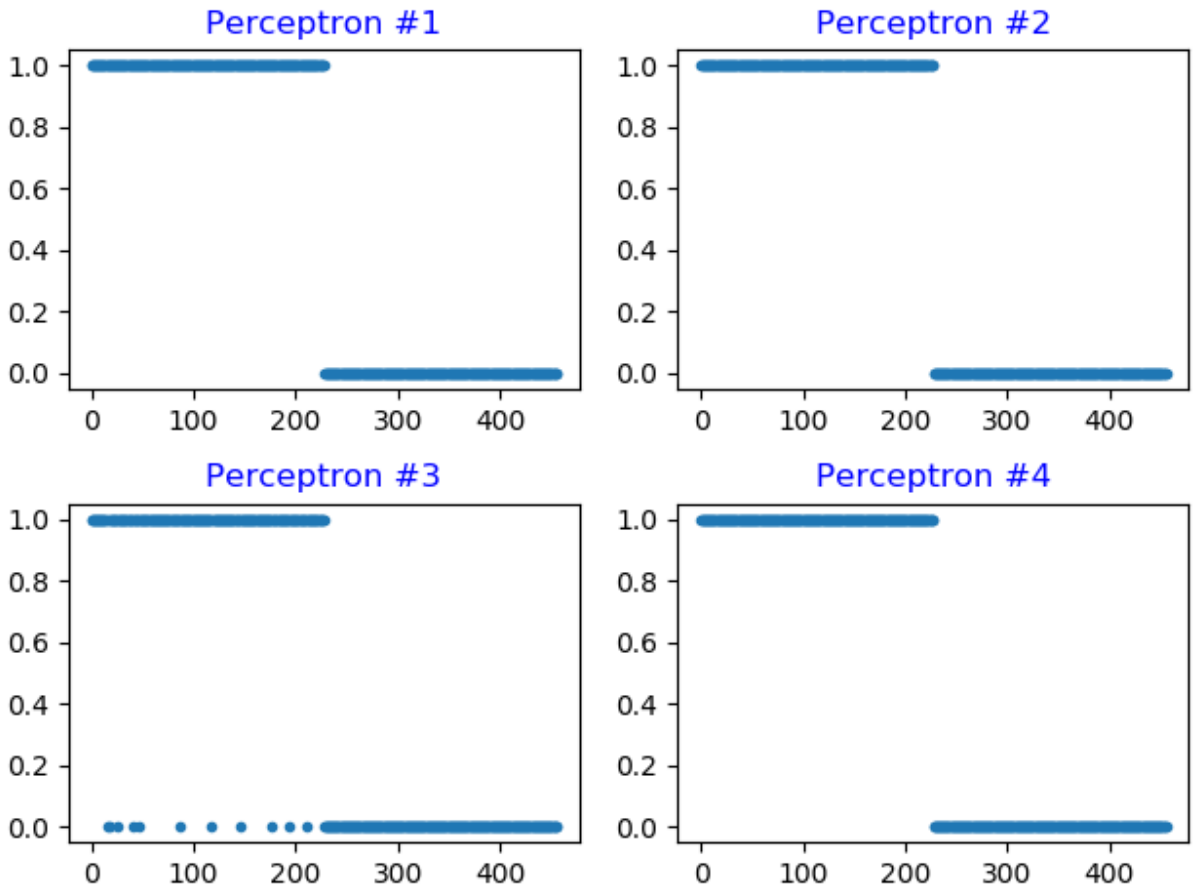
پس از چاپ شدن درصد دقت‌ها، نوبت به Confusion Matrix ها می‌رسد. برای هر ماتریس مدلی، چهار متغیر شمارش‌گر به شرح زیر وجود دارند که در ابتدا آن‌ها را صفر می‌کنیم. در حلقه‌ای برای هر مدل، اگر پیش‌بینی داده i ام ۱ باشد و برچسب واقعی هم ۱ باشد، TP زیاد شده، اگر برچسب واقعی ۰ باشد، FP زیاد شده، اگر پیش‌بینی داده ۰ باشد و برچسب واقعی ۱ باشد، FN زیاد



شده و در آخر اگر پیش‌بینی ۰ باشد و برچسب واقعی ۰، TN زیاد می‌شود. این قاعده برای هر ۴ مدل انجام شده و ماتریس مورد نظر استخراج شده‌است و با اجرای کد چاپ می‌شود.

در انتها نیز نمودار برچسب‌های پیش‌بینی شده هر مدل جداگانه و با هم برای مقایسه رسم می‌شود. ضمناً همان‌طور که پیش‌تر نگاشت گفته شد، برچسب ۱ معادل پیش‌بینی عدد ۱ و برچسب ۰ معادل پیش‌بینی عدد ۴ است.

نتیجه ۴ مدل با هم:



توضیح کد پرسپترون هر مدل:

برای این کلاس چند متغیر ورودی در نظر گرفته‌ام. تعداد وزن‌هایی که لازم است یاد بگیریم(در این مسأله ۱۶ تا)، تعداد دفعات تکرار و آپدیت کردن وزن‌ها، نرخ یادگیری(پارامتر آپدیت کردن وزن‌ها) و بسته به مدل چند پارامتر دیگر هم اضافه می‌شود. به این مقادیر اجازخ دسترسی در زمان runtime نیز می‌دهیم تا در صورت لزوم بتوانیم آن‌ها را تغییر دهیم. هم‌چنین دقت می‌کنیم برای بدست آوردن شکل ماتریسی صحیح بدون حضور جمله بایاس، یک وزن دیگر هم اضافه شده و ورودی این بعد ۱ می‌باشد.

پس از آن متد پیش‌بینی را برای این کلاس تعریف می‌کنیم. مقدار متغیر summation همان‌طور که واضح است، حاصل ضرب داخلی ورودی در بردار وزن‌ها(بدون وزن مربوط به جمله بایاس) به اضافه جمله بایاس می‌باشد. با توجه به این‌که تابع فعالیت را پله در نظر می‌گیریم، اگر این مجموع(که همان پیش‌بینی با این وزن‌های فعلی می‌باشد) از صفر بیشتر بود ۱ و در غیر این‌صورت مقدار ۰ را باز می‌گردانیم.

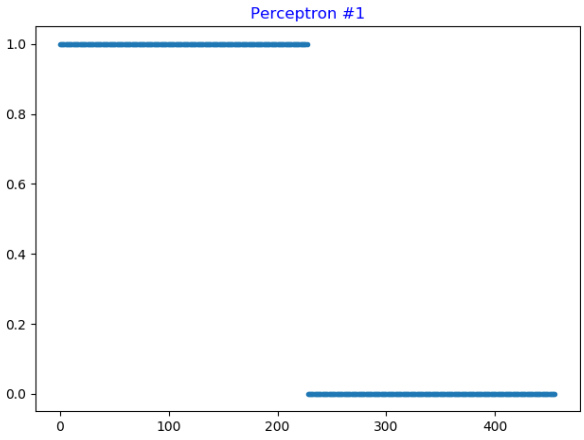
پس از آن نیز متد یادگیری را برای این کلاس تعریف می‌کنم. این متد دو متغیر داده‌های آموزش و برچسب باینری آن‌ها را برای آموزش دیدن می‌خواهد.(که نحوه تبدیل برچسب‌ها به باینری که در بالا توضیح داده شد این‌جا به کار می‌آید.) در ادامه این متد در صورتی که تعداد دفعات تکرار داده شده باشد از آن استفاده می‌کنیم و در غیر این‌صورت از مقدار default استفاده می‌شود.(آن را ۱۰۰۰ اختیار کردم.) در ادامه یک شئ که بتوان روی آن لوپ زد ایجاد می‌کنیم. این کار با زیپ کردن داده‌های آموزش و برچسب‌ها انجام می‌شود. در هر حلقه مقدار training_inputs در inputs و مقدار labels در label ذخیره می‌شود. سپس در ادامه همین حلقه، مقدار پیش‌بینی شده مدل با وزن‌های فعلی در prediction ذخیره می‌شود.(از آن‌جایی که sample mode یا batch mode بودن روش ذکر نشده از sample mode استفاده می‌کنیم.) پس از آن وزن‌ها با توجه به مدل که در هر مورد اشاره می‌شود تغییر می‌کنند و این حلقه به تعداد threshold تکرار می‌شود تا وزن‌ها بدست آیند.(البته وزن‌ها ممکن است قبل از رسیدن به تعداد تکرار تعیین شده نیز به مقدار صحیح رسیده باشند.)

در همه مدل‌ها نرخ یادگیری ۰,۰۱ تنظیم شده‌است. تعداد تکرارهای آپدیت شدن نیز ۱۰۰۰ تنظیم شده‌است. تفاوت کد پرسپترون‌ها در قسمت train آن‌ها است. تمام جمع‌ها در آپدیت کردن وزن‌ها بر روی داده‌هایی است که صحیح پیش‌بینی نشده‌اند.

مدل اول:

$$learning\ rule: \omega_{n+1} = \omega_n - \eta \sum \delta_x \vec{x}\ , where\ \eta\ is\ learning\ rate$$

نتیجه پیش‌بینی و کد:




```python
import numpy as np
import random
# Details about this code are presented in the report


class Perceptron1(object):

    def __init__(self, no_of_inputs, threshold=1000, learning_rate=0.01):
        self.threshold = threshold
        self.learning_rate = learning_rate
        self.weights = np.array([random.random() for _ in range(no_of_inputs + 1)])
        # self.weights = np.ones(no_of_inputs + 1)

    def predict(self, inputs):
        summation = np.dot(inputs, self.weights[1:]) + self.weights[0]
        if summation > 0:
            activation = 1
        else:
            activation = 0
        return activation

    def train(self, training_inputs, labels):
        for _ in range(self.threshold):
            for inputs, label in zip(training_inputs, labels):
                prediction = self.predict(inputs)
                delta = -np.sign((np.dot(inputs, self.weights[1:]) + self.weights[0]))
                if prediction != label:
                    self.weights[1:] += self.learning_rate * delta * inputs
                    self.weights[0] += self.learning_rate * delta
```

$$Precision = \%100$$

$$Confusion\ Matrix:$$

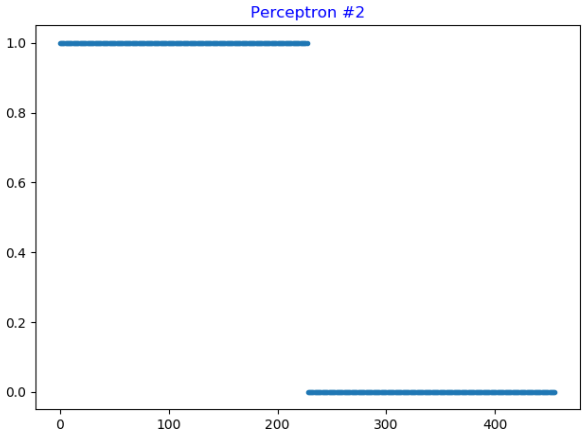
228	0
0	228

مدل دوم:

$learning\ rule$:

$$\omega_{n+1} = \omega_n - \eta\sum(\tanh(\omega^T\vec{x}) + (1 - \tanh^2\omega^T\vec{x})\omega^T\vec{x})\vec{x}\ ,where\ \eta\ is\ learning\ rate$$

نتیجه پیش‌بینی و کد:




```python
import numpy as np
import random
# Details about this code are presented in the report


class Perceptron2(object):

    def __init__(self, no_of_inputs, threshold=1000, learning_rate=0.01):
        self.threshold = threshold
        self.learning_rate = learning_rate
        self.weights = np.array([random.random() for _ in range(no_of_inputs + 1)])
        # self.weights = np.ones(no_of_inputs + 1)

    def predict(self, inputs):
        summation = np.dot(inputs, self.weights[1:]) + self.weights[0]
        if summation > 0:
            activation = 1
        else:
            activation = 0
        return activation

    def train(self, training_inputs, labels):
        for _ in range(self.threshold):
            for inputs, label in zip(training_inputs, labels):
                prediction = self.predict(inputs)
                summation = np.dot(inputs, self.weights[1:]) + self.weights[0]
                if prediction != label:
                    self.weights[1:] -= self.learning_rate * (np.tanh(summation) + summation *
                                                              (1 - (np.tanh(summation) ** 2))) * inputs
                    self.weights[0] -= self.learning_rate * (np.tanh(summation) + summation *
                                                             (1 - (np.tanh(summation) ** 2)))
```

$$Precision = \%100$$

$$Confusion\ Matrix:$$

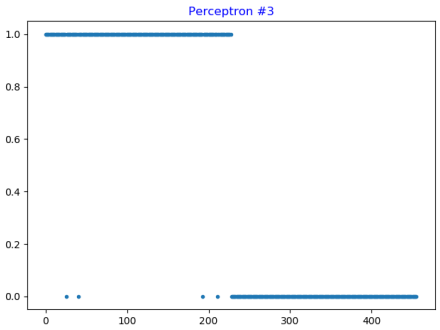
228	0
0	228

مدل سوم: مقدار c با جستجو در اینترنت و امتحان کردن مقادیر ۰٫۶ انتخاب شد. در کد نیز اضافه شده‌است.

$$learning\ rule:$$

$$\omega_{n+1} = \omega_n - \eta \begin{Bmatrix} \sum c\vec{x} & \omega^T\vec{x} > c \\ \sum -c\vec{x} & \omega^T\vec{x} < -c \\ \sum(\omega^T\vec{x})\vec{x} & elsewhere \end{Bmatrix}$$

نتیجه پیش‌بینی و کد:




```python
import numpy as np
import random
# Details about this code are presented in the report


class Perceptron3(object):

    def __init__(self, no_of_inputs, threshold=1000, learning_rate=0.01, c=0.6):
        self.threshold = threshold
        self.learning_rate = learning_rate
        self.c = c
        self.weights = np.array([random.random() for _ in range(no_of_inputs + 1)])
        # self.weights = np.ones(no_of_inputs + 1)

    def predict(self, inputs):
        summation = np.dot(inputs, self.weights[1:]) + self.weights[0]
        if summation > 0:
            activation = 1
        else:
            activation = 0
        return activation

    def train(self, training_inputs, labels):
        for _ in range(self.threshold):
            for inputs, label in zip(training_inputs, labels):
                prediction = self.predict(inputs)
                summation = np.dot(inputs, self.weights[1:]) + self.weights[0]
                if prediction != label:
                    if summation > self.c:
                        self.weights[1:] -= self.learning_rate * self.c * inputs
                        self.weights[0] -= self.learning_rate * self.c
                    if summation < -self.c:
                        self.weights[1:] -= self.learning_rate * self.c * -1 * inputs
                        self.weights[0] -= self.learning_rate * self.c * -1
                    else:
                        self.weights[1:] -= self.learning_rate * summation * inputs
                        self.weights[0] -= self.learning_rate * summation
```

$$Precision = \%99.12$$

$$Confusion\ Matrix:$$

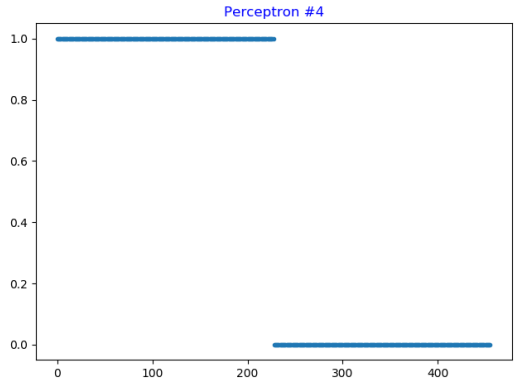
228	4
0	228

مدل چهارم: در این مدل از تابع $c(1-e^{-x^2})$ برای تقریب قدرمطلق در مقادیر کوچک استفاده شده‌است. برای برابری آن و پیوستگی مشتق آن دو پارامتر α و β باید تعیین می‌شدند که نحوه محاسبه آن‌ها در تمرین تئوری موجود است. مقدار بتا را ۰,۵ و آلفا را با توجه به آن ۱,۳۸ انتخاب کردم(حل معادله) و در پارامترهای ورودی کلاس اضافه شده‌اند.

$$learning\ rule:$$

$$\omega_{n+1} = \omega_n - \eta \begin{cases} \sum \beta \vec{x} & \omega^T \vec{x} > \alpha \\ -\sum \beta \vec{x} & \omega^T \vec{x} < -\alpha \\ -\sum \left(\frac{\beta}{\alpha}\right) e^{\alpha^2} \left(\omega^T \vec{x}\right) \vec{x}\, e^{-\left(\omega^T \vec{x}\right)^2} & elsewhere \end{cases}$$

نتیجه پیش‌بینی و کد:




```
import numpy as np
import random
# Details about this code are presented in the report
class Perceptron4(object):

    def __init__ (self, no_of_inputs, threshold=1000, learning_rate=0.01, beta=0.5, alpha=1.13838):
        self.threshold = threshold
        self.learning_rate = learning_rate
        self.beta = beta
        self.alpha = alpha
        self.weights = np.array([random.random() for _ in range(no_of_inputs + 1)])
        # self.weights = np.ones(no_of_inputs + 1)

    def predict(self, inputs):
        summation = np.dot(inputs, self.weights[1:]) + self.weights[0]
        if summation > 0:
            activation = 1
        else:
            activation = 0
        return activation

    def train(self, training_inputs, labels):
        for _ in range(self.threshold):
            for inputs, label in zip(training_inputs, labels):
                prediction = self.predict(inputs)
                summation = np.dot(inputs, self.weights[1:]) + self.weights[0]
                if prediction != label:
                    if summation > self.alpha:
                        self.weights[1:] -= self.learning_rate * self.beta * inputs
                        self.weights[0] -= self.learning_rate * self.beta
                    if summation < -self.alpha:
                        self.weights[1:] -= self.learning_rate * self.beta * -1 * inputs
                        self.weights[0] -= self.learning_rate * self.beta * -1
                    else:
                        self.weights[1:] -= self.learning_rate * summation * inputs * np.exp(-(summation ** 2)) *\
                                            np.exp(self.alpha ** 2) * self.beta / self.alpha
                        self.weights[0] -= self.learning_rate * summation * np.exp(-(summation ** 2)) *\
                                           np.exp(self.alpha ** 2) * self.beta / self.alpha
```

$$Precision = \%100$$

$Confusion\ Matrix$:

228	0
0	228

تفسیر و مقایسه: تفاوت تابع هزینه‌ها در تقریب تابع قدر مطلق و تابع علامت است. اگر تعداد تکرارها زیاد باشد انتظار تفاوتی در دقت نمی‌رود چون با مشاهده داده‌ها می‌توان به linear separability آن‌ها پی‌برد و لذا باید روش Gradient Descent همگرا شود. تفاوت فقط در سرعت است. در کد من عملیات آپدیت کردن به صورت sample mode پیاده‌سازی شده است و هم‌چنین آموزش تا تعداد تکرار خاصی انجام می‌شود، لذا امکان مقایسه سرعت همگرایی وجود ندارد. هرچه مشتق‌پذیری تقریب قدرمطلق و تابع علامت بیشتر باشد روش گرادیان دیسنت بهتر عمل می‌کند و به نظر می‌رسد همه تقریب‌ها قابل قبول بوده‌اند.(دقت همه ۱۰۰ درصد است به جز یکی که بیش از ۹۹ درصد است.) پس به طور کلی در صورتی که داده‌ها جدایی‌پذیر خطی باشند، انتظار تفاوت در پیش‌بینی نمی‌رود چون همگی همگرا می‌شوند.(قضیه همگرایی ورش برای توابع convex) البته از تفاوت پیچیدگی محاسباتی روش‌های مختلف نباید چشم‌پوشی کرد و در جایی که تعداد داده‌ها بسیار بیشتر باشد این عملیات می‌تواند تأثیر قابل توجهی بگذارد و باید دقت لازم را برای انتخاب تابع هزینه با تقریب مناسب در نظر داشت.