به نام خدا

گزارش پروژه اول هوش محاسباتی

ستایش ثانوی

۹۶۲۸۰۲۴

قدم اول :دریافت دیتاست:

در بخش اول دقیقا طبق کد داده شده پیش رفتم و تنها کلاس بندی ها را عوض کردم تا آرایه های داده های test_set , train_set را در کلاس های بعدی به راحتی استفاده کنم.

قدم دوم: محاسبه خروجی (feedForward) :

ابتدا داده ها را به کمک فرمول گفته شده یعنی به صورت ضرب و جمع ماتریسی/برداری و اعمال تابع سیگموئید از طریق کتابخانه numpy محاسبه کرده و با توجه به وزنی که برای هر لایه و لایه خروجی داریم میوه تشخیص داده شده از ورودی را با توجه به مکان عدد ۱ در سطر آرایه میفهمیم که کدام میوه را مدل ما با توجه به محاسبات بدست آورده و نهایتا با مقایسه آن با داده های ورودی در ۲۰۰ داده میتوانیم تعداد بار هایی که درست تشخیص داده را متوجه شویم ولی با توجه به اینکه اعداد برای وزن ها به صورت تصادفی انتخاب شده اند تعداد زیادی از داده ها به درستی تشخیص داده نمیشوند.

دقت را مطابق کد زیر محاسبه کردم:

```
all_count = 0
success_count = 0
for i in range(0,199):
all_count = all_count +1
input_data = dataset[i][0]
temp = FeedForward.layers(FeedForward, 102, 4, 150, 60, input_data)
if temp.argmax() == input_data.argmax():
success_count = success_count + 1
accuracy = success_count / all_count
```

مقدار دقت :

خط اول در شکل زیر معادل تعداد دفعاتی که از این ۲۰۰ بار به درستی تشخیص دادیم، خط بعدی کل داده هاست و خط آخر همان دقت بر حسب صدم میباشد که به طور متوسط ۲۵ درصد شد.

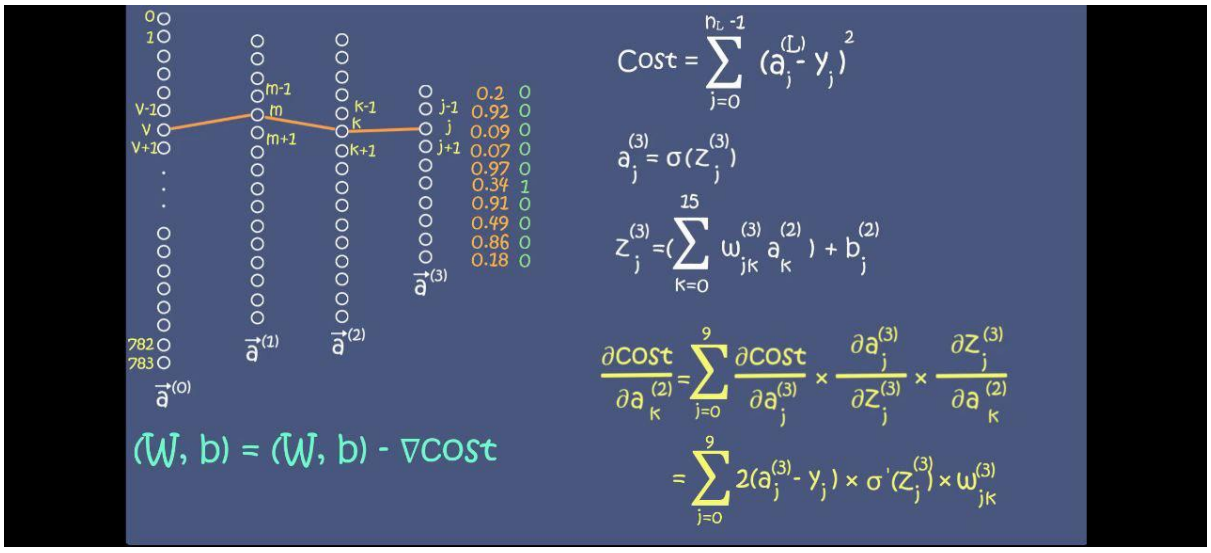
```
In [46]: runfile('C:/Users/Setayesh/Desktop/
hoosh_mohasebaty/prj1_ANN/ANN_Project_Assets/
FeesForward.py', wdir='C:/Users/Setayesh/Desktop/
hoosh_mohasebaty/prj1_ANN/ANN_Project_Assets')
Reloaded modules: dataSets
44
199
0.22110552763819097

In [47]: runfile('C:/Users/Setayesh/Desktop/
hoosh_mohasebaty/prj1_ANN/ANN_Project_Assets/
FeesForward.py', wdir='C:/Users/Setayesh/Desktop/
hoosh_mohasebaty/prj1_ANN/ANN_Project_Assets')
Reloaded modules: dataSets
55
199
0.27638190954773867
```

قدم سوم: پیاده سازی (Backpropagation) :

شبه کد گفته شده را دقیقا پیاده سازی کردم و برای گرفتم مشتقات گفته شده و گرادیان ها از backpropagation و مشتقات جزیی در هر لایه استفاده کردم و در این قسمت برای انجام جمع خروجی های نرون هر لایه از for استفاده کردم:

فرمول ها :




```
for image, label in batch:
#           compute the output (image is equal to a0)
```

خروجی لایه اول با اعمال تابع فعال سازی : `a1 = self.act_fun(W1.dot(image) + b1)`

خروجی لایه اول بدون اعمال تابع فعال سازی : `z1 = W1.dot(image) + b1`

```
            a2 = self.act_fun(W2.dot(a1) + b2)
            z2 = W2.dot(self.act_fun(a1)) + b2
            a3 = self.act_fun(W3.dot(a2) + b3)
            z3 = W3.dot(self.act_fun(a2)) + b3
=======#             backpropagation ========
#             weight for Last layer    گرادیان وزن لایه آخر از طریق حلقه تکرار و استفاده از فرمول های گفته شده:
            for j in range(output_):
                for k in range(hidden2_):
                    grad_W3[j, k] += 2 * (a3[j, 0] - label[j, 0]) * self.der_act_fun(z3[j, 0]) * a2[k, 0]
#             bias for Last layer
            for j in range(output_):
                    grad_b3[j, 0] += 2 * (a3[j, 0] - label[j, 0]) * self.der_act_fun(z3[j, 0])
#             activation for 3rd layer
            delta_3 = np.zeros((hidden2_, 1))
            for k in range(hidden2_):
                for j in range(output_):
                    delta_3[k, 0] += 2 * (a3[j, 0] - label[j, 0]) * self.der_act_fun(z3[j, 0]) * W3[j, k]
#             weight for 2rd layer
            for k in range(hidden2_):
                for m in range(hidden1_):
                    grad_W2[k, m] += delta_3[k, 0] * self.der_act_fun(z2[k,0]) * a1[m, 0]
#             bias for 2rd layer
            for k in range(hidden2_):
                    grad_b2[k, 0] += delta_3[k, 0] * self.der_act_fun(z2[k,0])
#             activation for 2nd layer
            delta_2 = np.zeros((hidden1_, 1))
            for m in range(hidden1_):
                for k in range(hidden2_):
                    delta_2[m, 0] += delta_3[k, 0] * self.der_act_fun(z2[k,0]) * W2[k, m]
#             weight for first layer
            for m in range(hidden1_):
```

```
                for v in range(input_):
                    grad_W1[m, v] += delta_2[m, 0] * self.der_act_fun(z1[m,0]) * image[v, 0]
#               bias for first layer
            for m in range(hidden1_):
                grad_b1[m, 0] += delta_2[m, 0] * a1[m, 0] * (1 - z1[m, 0])
        W3 = W3 - (learning_rate * (grad_W3 / batch_size))
        W2 = W2 - (learning_rate * (grad_W2 / batch_size))
        W1 = W1 - (learning_rate * (grad_W1 / batch_size))
        b3 = b3 - (learning_rate * (grad_b3 / batch_size))
        b2 = b2 - (learning_rate * (grad_b2 / batch_size))
        b1 = b1 - (learning_rate * (grad_b1 / batch_size))
#       calculate cost average per epoch  متوسط هزینه را با استفاده از مجموع مربعات خطا بدست آوردم.
    cost = 0
    for train_data in dataset[:200]:
        a0 = train_data[0]
        a1 = self.act_fun(W1.dot(a0) + b1)
        a2 = self.act_fun(W2.dot(a1) + b2)
        a3 = self.act_fun(W3.dot(a2) + b3)
        for j in range(output_):
            cost += np.power((a3[j, 0] - train_data[1][j,  0]), 2)
    cost = cost/ 200
    total_costs.append(cost)
  number_of_correct_estimations = 0
  for train_data in dataset[:200]:
    a0 = train_data[0]
    a1 = self.act_fun(W1.dot(a0) + b1)
    a2 = self.act_fun(W2.dot(a1) + b2)
    a3 = self.act_fun(W3.dot(a2) + b3)
    if a3.argmax() == train_data[1].argmax():  با توجه به مکان های ۱ بودن آرایه خروجی از الگوریتم ما و برابر بودن
آن با مکان ۱ در مقدار داده داده شده دقت را بدست آوردم.
        number_of_correct_estimations += 1
```

print(f"Accuracy: {number_of_correct_estimations / 200}")

دقت و زمان صرف شده در این بخش:

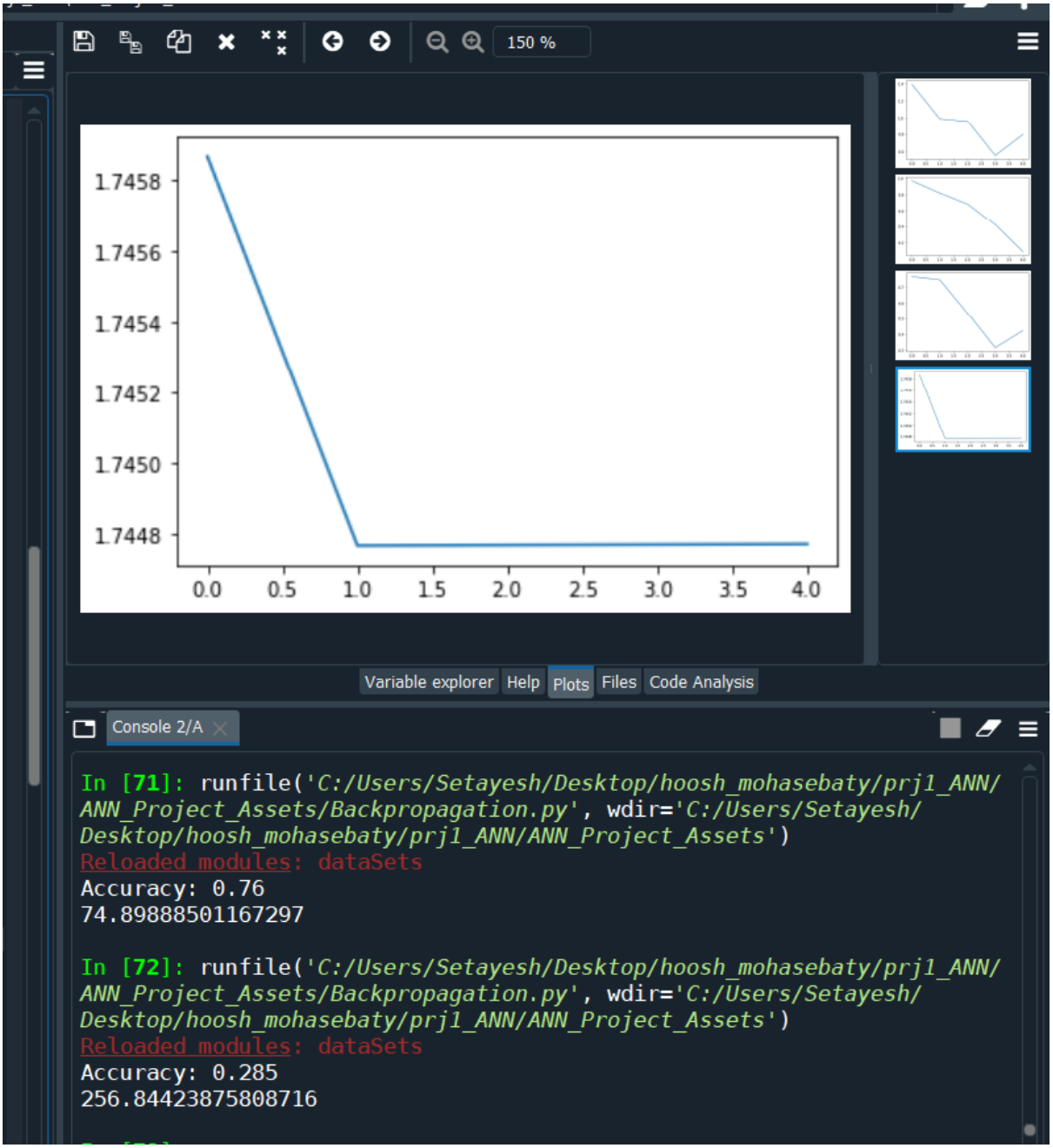
زمان بر حسب میلی ثانیه است و نموداز هم با plot در پایتون نوشته شده که برای ۲۰۰ داده و یک بار اجرا با مقادیر

learning_rate = 1

number_of_epochs = 5

batch_size = 10

مطابق شکل زیر خواهد بود :



که زمان برحسب میلی ثانیه است و دقت به طور میانگین به ۵۰ درصد نزدیک است .همه این اطلاعات به ازای انجام یک بار الگوریتم برای ۲۰۰ داده میباشد.

قدم چهارم : Vectorization:

با توجه به زمانگیر بودن حلقه های تکرار در حالت قبل با استفاده از بردار میتوانیم این زمان را به شدت کاهش دهیم و چون زمان اجرا کاهش یافته میتوان تعداد بیشتری از مجموعه های ورودی به مدل داد و دقت را بهتر محاسبه کرد با توجه به این موارد تغییراتی که در کد نسبت به حالت قبل داشتیم :

```
#======= backpropagation ========
#weight for Last layer
grad_W3 += 2 * (a3 - label) * self.der_act_fun(z3) @ np.transpose(a2)
# bias for Last layer
grad_b3 += 2 * (a3 - label) * self.der_act_fun(z3)
# activation for 3rd layer
delta_3 = np.zeros((hidden2_, 1))
delta_3 += np.transpose(W3) @ (2 *(a3 - label) * self.der_act_fun(z3))
# weight for 2rd layer
grad_W2 += delta_3 * self.der_act_fun(z2) @ np.transpose(a1)
# bias for 2rd layer
grad_b2 += delta_3 * self.der_act_fun(z2)
# activation for 2nd layer
delta_2 = np.zeros((hidden1_, 1))
delta_2 += np.transpose(W2) @ (delta_3 * self.der_act_fun(z2))
# weight for first layer
grad_W1 += delta_2 * self.der_act_fun(z1) @ np.transpose(image)
# bias for first layer
grad_b1 += delta_2 * self.der_act_fun(z1)
```

با توجه به ابعاد باید ضرب برداری و ماتریسی را انجام داد و تعداد تکرار این ۱۰ بار داده ۲۰۰ تایی و میانگین گیری از اطلاعات خروجی آن ها معادل :

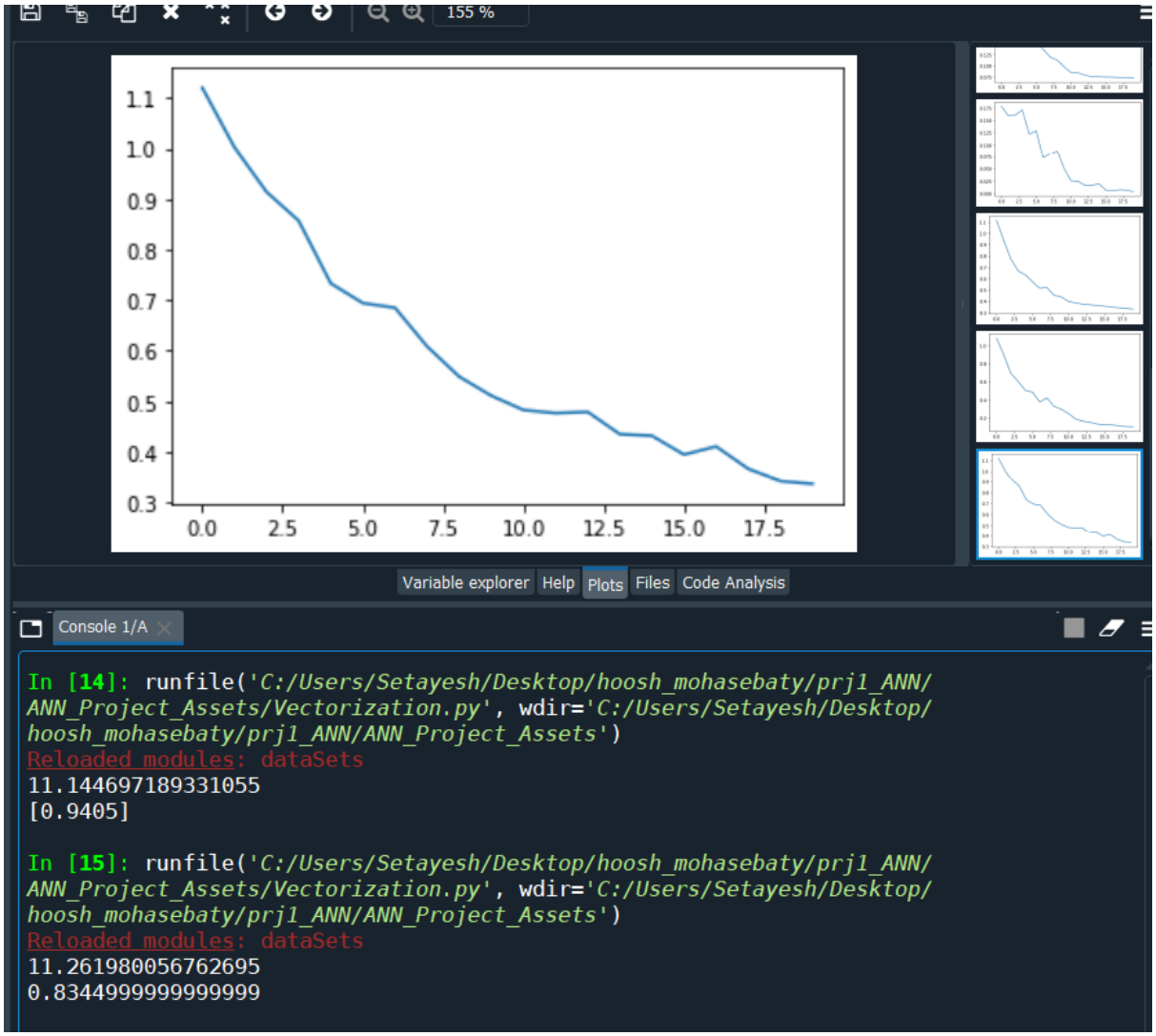
```
total_costs,Accuracy = Backpropagation.StochasticGradientDescent(Backpropagation, 102, 4, 150, 60)
temp = []
for i in range(9):
```

```
    temp.append(Backpropagation.StochasticGradientDescent(Backpropagation, 102, 4, 150, 60))
    total_costs =  np.array(total_costs) + np.array(temp[i][0])
    Accuracy = np.array(Accuracy) + np.array(temp[i][1])
print(end_time - start_time)
avg_total_cost = np.array(total_costs)/10
avg_accuracy = np.array(Accuracy)/10
print(avg_accuracy[0])
```

با توجه به این مقادیر محاسبه شده داریم :

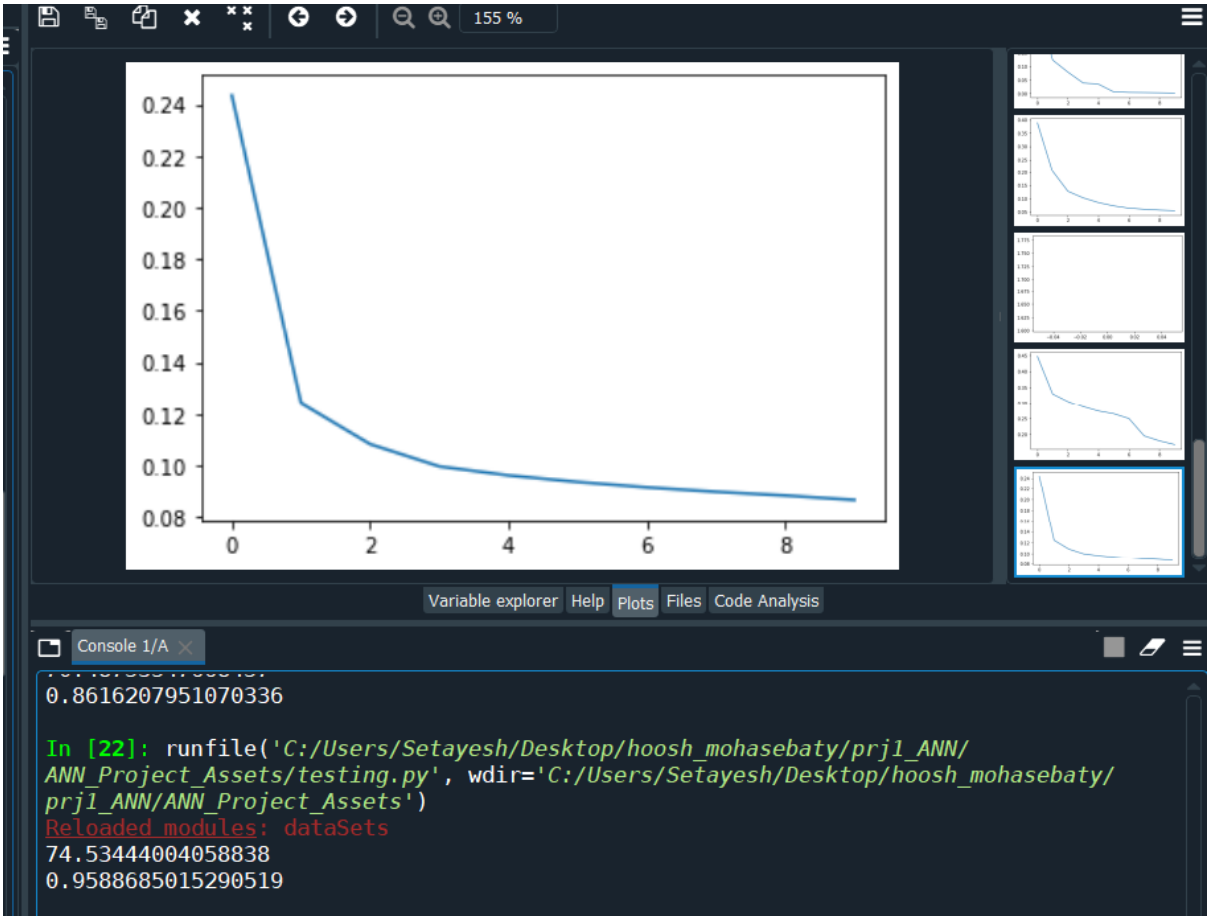
که خط اول زمان و خط دوم دقت است که متوجه میشویم که هم زمان کمتری (حتی برای ۱۰ بار تکرار الگوریتم) و هم دقت بیشتری داریم. دقت به طور متوسط برای داده های ۹۰ درصد میشود.

```
    learning_rate = 1
    number_of_epochs = 20
    batch_size = 10
```

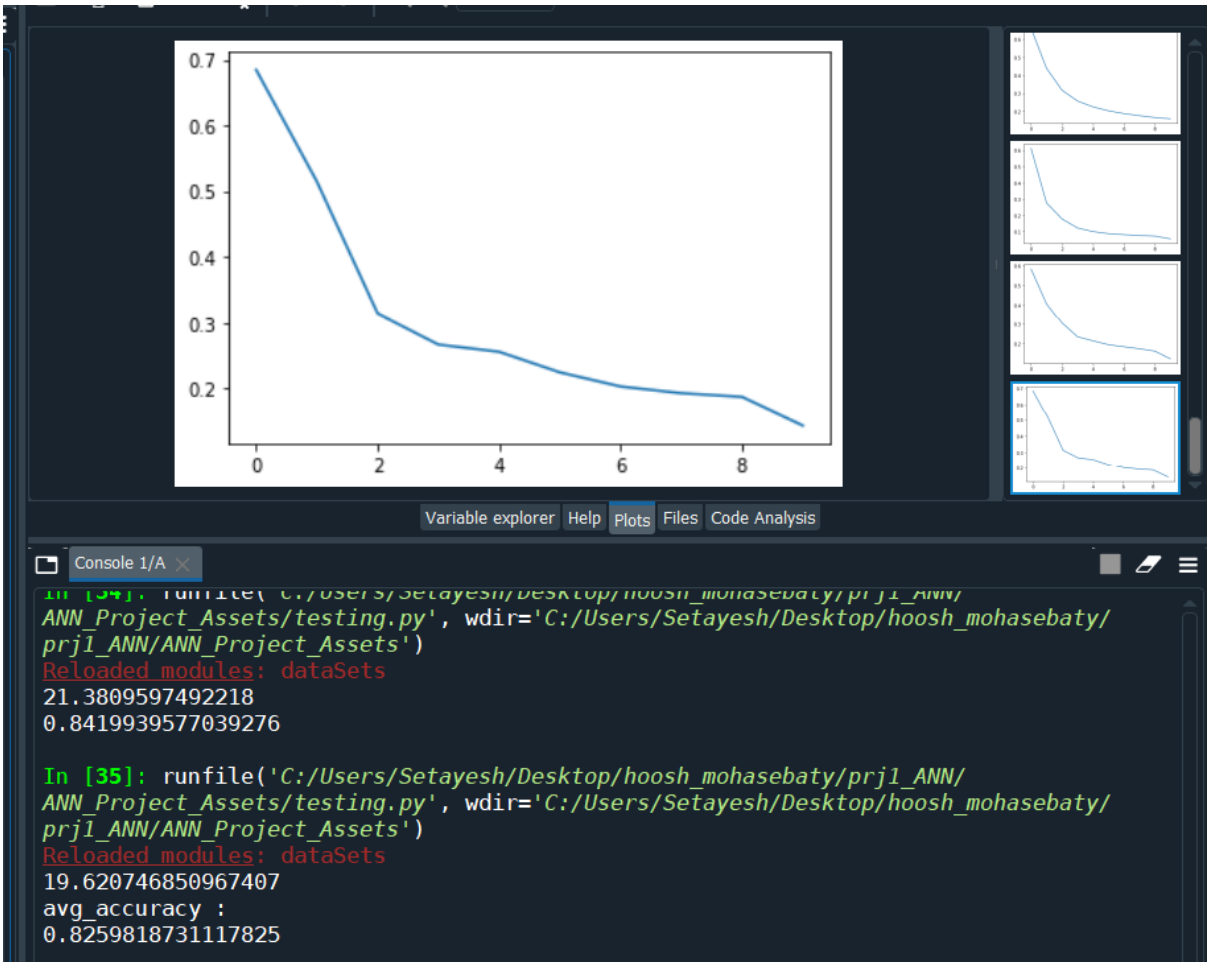


قدم پنجم: تست کردن مدل:

برای داده های train که ۱۹۶۲ عدد داده است دقت به طور متوسز به ازای ۱۰ بار اجرا 90 درصد و زمان حدودا ۹ ثانیه است.(در شکل زیر خط اول زمان برحسب میلی ثانیه و خط بعدی دقت است.



برای داده های تست هم به همین منوال امتحان کردیم و نتیجه مطابق شکل زیر شد که باز هم خط اول زمان و خط دوم دقت است به ازای ۱۰ بار تکرار ۶۶۹ داده تستی که در اختیار داریم. که دقت به طور متوسط حدود ۸۲ درصد است.



امتیازی ها :

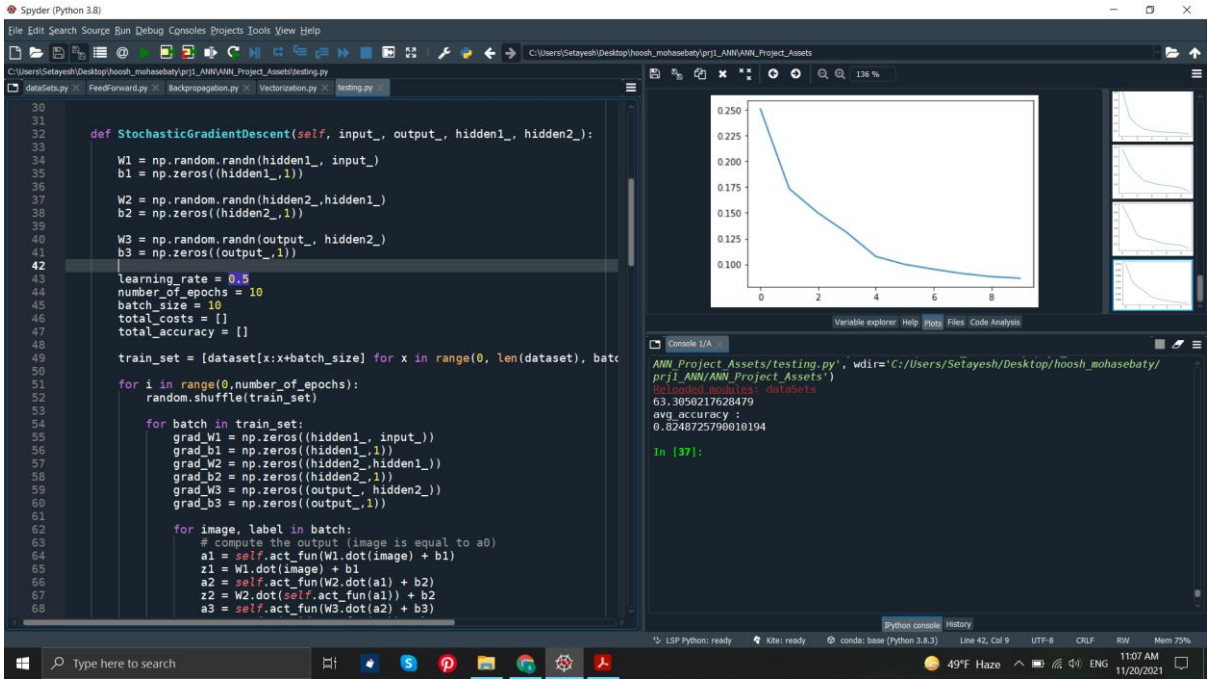
۱. به ازای داده های train که ۱۹۶۲ داده است و ۱۰ بار اجرای الگوریتم مقدار hyperparameter ها را تغییر میدهیم تا به دقت بهتری دست پیدا کنیم.

Learning rate = 0.5

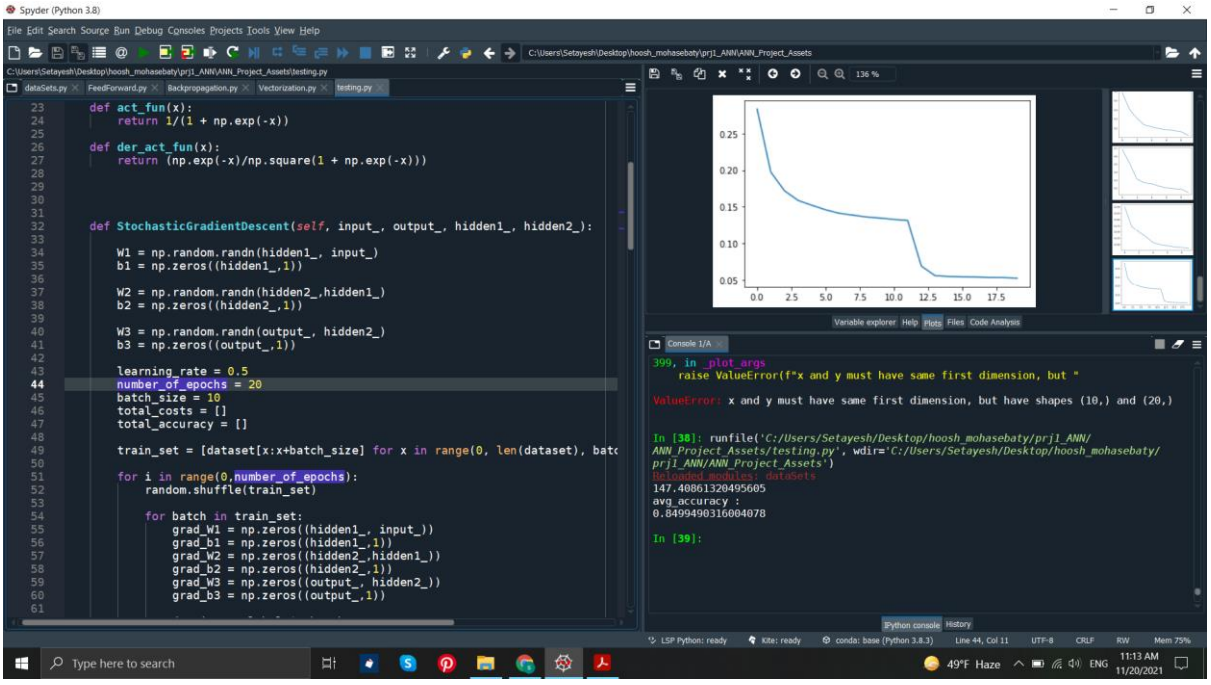
Number of epochs = 10

Batch size = 10

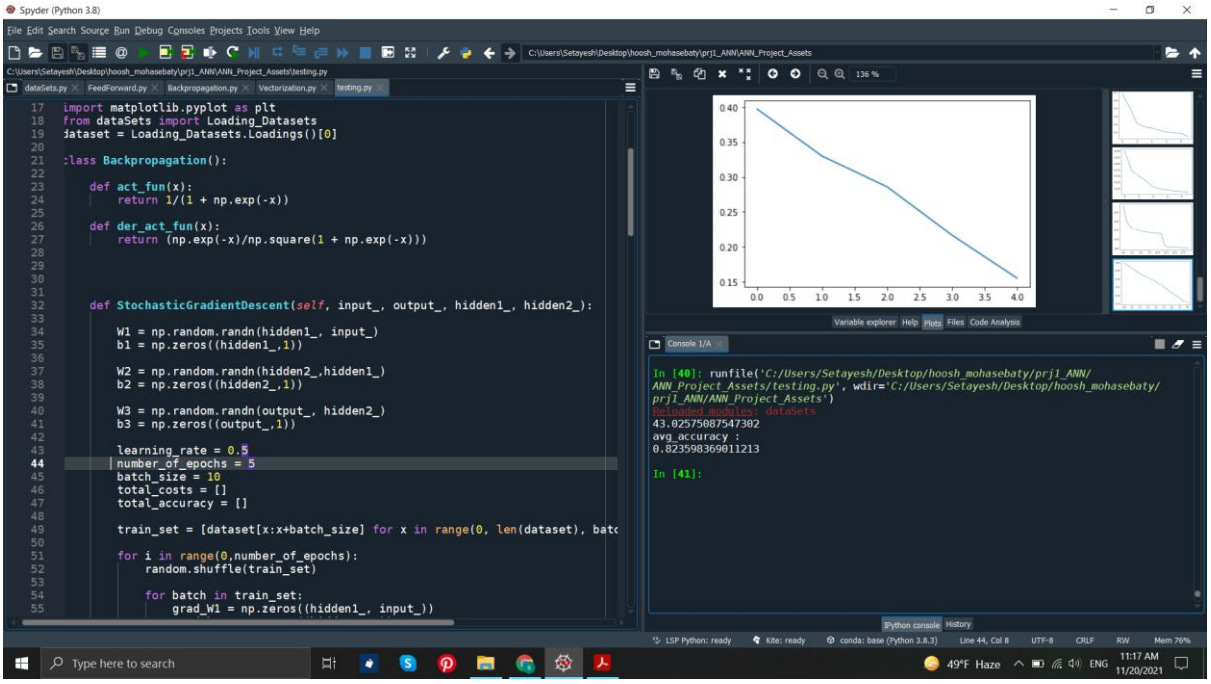
دقت برای این داده ها به طور میانگین ۸۲ درصد با زمان ۶ ثانیه است که میفهمیم دقت کاهش یافته



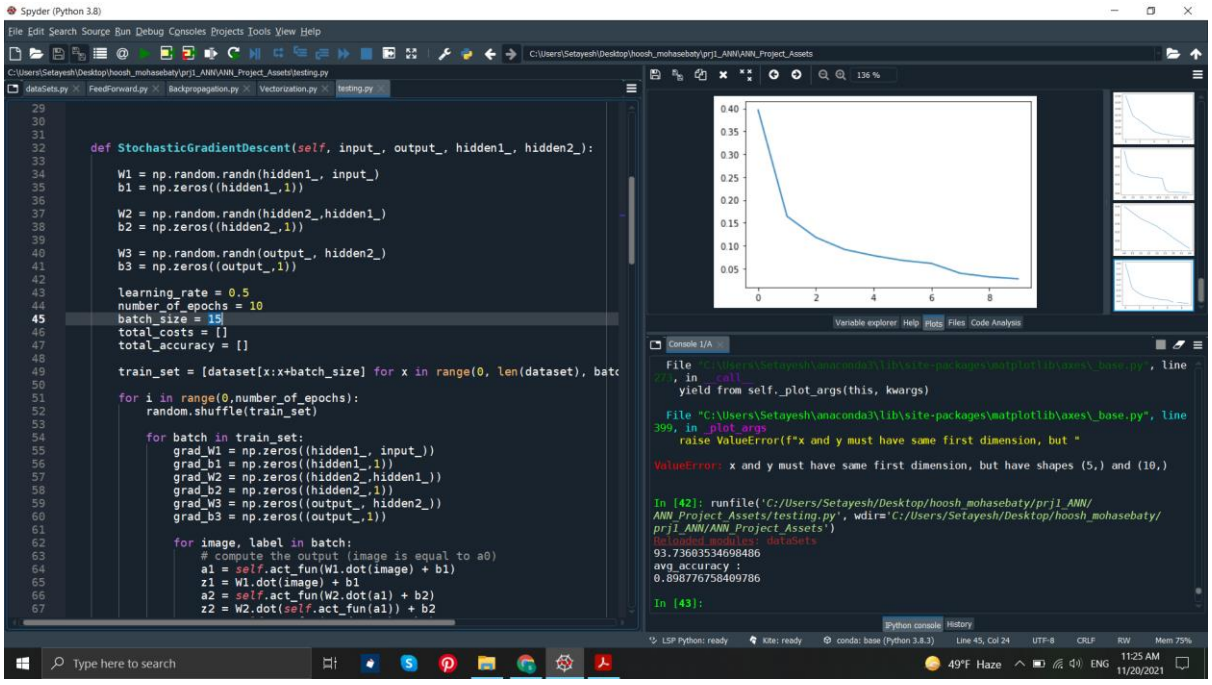
در حالت بعدی با داده هایی که در شکل داریم :



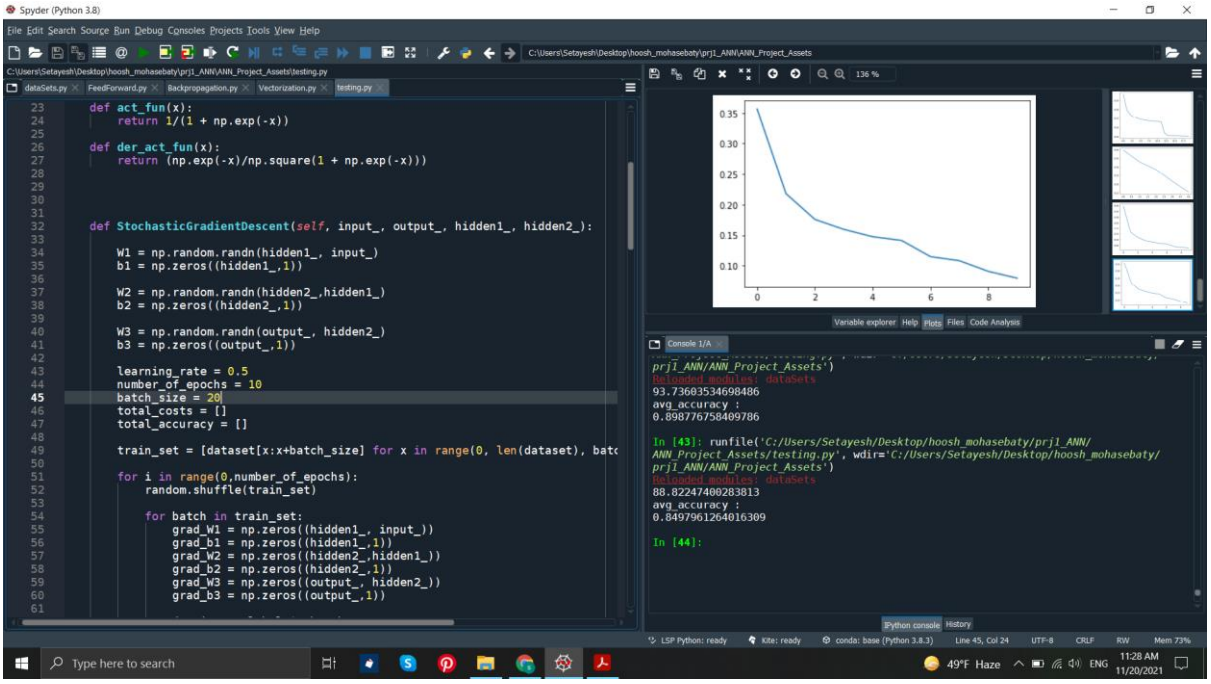
با افزایش تعداد epoch ها زمان اجرا خیلی افزایش داشته در صورتی که دقت افزایش چشم گیری نداشته .



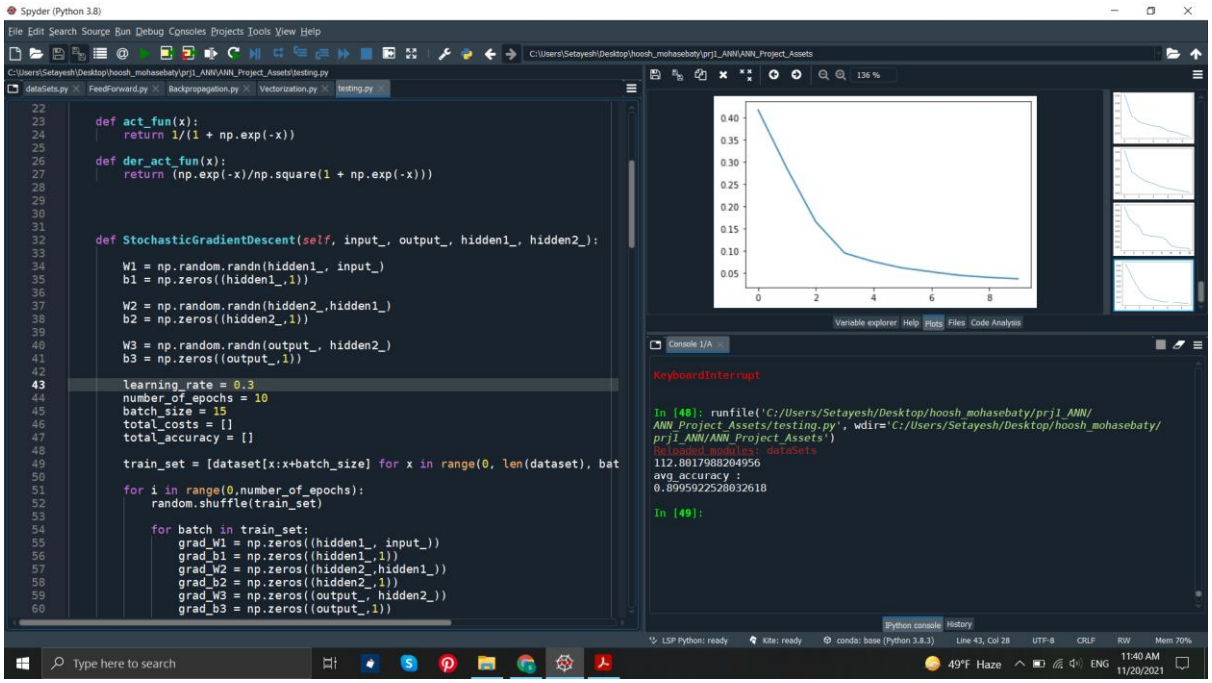
با کاهش بیشتر تعداد epoch ها زمان اجرا کم میشود ولی دقت هم کاهش میابد پس همان مقدار ۱۰ را برای آن درنظر میگیریم.



با افزایش batch size زمان اجرا کمی بیشتر شده ولی از طرف دیگر دقت به شدت افزایش میابد و حدود ۹۰ درصد میشود. ولی با افزایش بیشتر این مقداد باز دقت کاهش میابد (مطابق شکل زیر) پس همین مقدار ۱۵ را برایش درنظر میگیریم



سپس با مقادیر شکل زیر به دقت تقریبا ۹۰ درصد میرسیم که دقت خوبی است :



نهایتا با توجه به اینکه هر چقدر مقدارlearning rate کاهش میابد نمودار دیرتر به همگرایی میرسد و سرعت همگرایی کاهش میابد . البته اگر بیش از حد نیز افزایش یابد ممکن است واگرایی رخ دهد پس بهتر است مقداری بین مقادیر خیلی بزرگ و کوچک انتخاب شود پس مقدار ۰,۳ را برای آن درنظر گرفتم.

برای number of epochs میتوان گفت هر چقدر که افزایش میابد دقت نیز افزایش میابد البته از طرفی سرعت اجرای برنامه پایین میاید و همچنین ممکن است دچار اورفیت شویم .با این حال میتوان گفت که ۱۰ مقدار مناسبی است.

هر چقدر مقدار batch افزایش میابد دقت رو به کاهش میرود و نمودار دیرتر به همگرایی میرسد پس در اینجا میتوان گفت مقدار 15 مقدارمناسبی است.

۲. روش دیگری که میتوانیم برای جلوگیری از مینیمم های محلی در Stochastic Gradient Descent انجام دهیم همین SGD بر مبنای تکانه یا momentum است.

Momento لزوما از مینیمم های محلی صرف نظر نمی کند و به مینیمم های کلی همگرا نمی شود. در عوض، به احتمال زیاد از دست اندازهای کوچک عبور می کند و احتمال بیشتری وجود دارد که زمان خود را در اطراف دست اندازهای بزرگ بگذرانید.

ایده اصلی این استدلال این است که مینیمم های کلی به احتمال زیاد برآمدگی های بزرگ دارند. اما به طور دقیق تر، راه حل های موجود در نقاط بحرانی در اطراف دست اندازهاهای بزرگ به احتمال زیاد تقریباً معادل با مینیمم های کلی هستند تا راه حل در نقاط بحرانی در اطراف دست اندازهای کوچک. برای روشن بودن، توجه داشته باشید که این استدلال مستلزم برخی اظهارات اثبات نشده یعنی اندازه نسبی برآمدگی ها در اطراف نقطه بحرانی است.

$$V_t = \beta V_{t-1} + (1-\beta)\nabla_w L(W, X, y)$$
$$W = W - \alpha V_t$$

L — is loss function, triangular thing

$$V_t = \beta V_{t-1} + \alpha \nabla_w L(W, X, y)$$
$$W = W - V_t$$

که در tenserflow این مقدار :

tf.train.MomentumOptimizer = SGD + momentum