توضیح کد :

ابتدا طبق کد داده شده در **Loading_Datasets.py**، داده ها لود و کاهش سایز داده شدند.

```
# Number of code executions
iteration_number = 1
# Number of data for training
train_number = len(train_set)
epochs = 5
batch_size = 10
learning_rate = 1
# Learning rate method: 0 = divide by batch size , 1 = divide by gradian norm
learning_rate_method = 1
is_vectorized = True
```

سپس برای انجام تمام قدم های ذکرشده در تمرین تعدادی متغیر تعریف شده که با تنظیم آن ها می توان اعمال مختلفی را تست کرد.

متغیر اول، تعداد تکرار کد برای یادگیری مجدد است.

متغیر دوم، تعداد داده برای آموزش است که در ابتدا با ۲۰۰ داده شروع شد و درانتها به کل **train set** رسید.

متغیر یکی مانده به آخر، برای قسمت امتیازی پیاده سازی شده، به این صورت که هنگام اعمال تاثیر گرادیان بر وزن ها می توان به جای تقسیم بر **batch size** بر اندازه گرادیان تقسیم کرد که تاثیر مثبت قابل توجهی بر دقت و سرعت یادگیری داشت که در قسمت مربوطه توضیح داده می شود.

متغیر آخر، برای پیاده سازی دو حالت پیاده سازی با **for** یا پیاده سازی برداری است که با تنظیم آن هر یک از دو حالت اعمال می شود.

```
for iteration in range(iteration_number):
    random.shuffle(train_set)
    random.shuffle(test_set)
    W = [np.random.randn(150, 102), np.random.randn(60, 150), np.random.randn(4, 60)]
    B = [np.zeros((150, 1)), np.zeros((60, 1)), np.zeros((4, 1))]
    A = []

    # first feed forward
    for i in range(train_number):
        A.append([train_set[i][0]])
        for j in range(3):
            A[i].append(sigmoid(W[j] @ A[i][j] + B[j]))

    # before training test
    true = 0
    for i in range(train_number):
        if np.argmax(train_set[i][1]) == np.argmax(A[i][3]):
            true += 1
    before_training_precision.append(true / train_number)

    start_time = time.time()
```

سپس برای هر تکرار کد، یکبار مجموعه داده و تست را **shuffle** می کنیم و آرایه هایی از وزن ها و بایاس ها و خروجی های هر لایه تشکیل می دهیم.

آرایه مربوط به خروجی ۴ مجموعه دارد، مجموعه ابتدایی همان مجموعه داده های آموزشی است و ۳ مجموعه بعد به ترتیب مربوط به خروجی هر یک از لایه های بعدی است.

ابتدا یکبار مرحله **feed forward** انجام می شود و دقت شبکه قبل از یادگیری تست می شود.

```
for epoch in range(epochs):
    for n in range(0, train_number - batch_size + 1, batch_size):
        if is_vectorized:
            G_W = [np.zeros((150, 102)), np.zeros((60, 150)), np.zeros((4, 60))]
            G_B = [np.zeros((150, 1)), np.zeros((60, 1)), np.zeros((4, 1))]
            G_A0 = np.zeros((batch_size, 150, 1))
            G_A1 = np.zeros((batch_size, 60, 1))
            G_A2 = np.zeros((batch_size, 4, 1))
            for i in range(n, n + batch_size):
                for j in range(3):
                    A[i][j + 1] = sigmoid(W[j] @ A[i][j] + B[j])
            # backpropagation
            for k in range(n, n + batch_size):
                G_A2[k - n] = 2 * (A[k][3] - train_set[k][1])
                b_temp = G_A2[k - n] * A[k][3] * (1 - A[k][3])
                G_B[2] += b_temp
                G_W[2] += b_temp @ np.transpose(A[k][2])
            for k in range(n, n + batch_size):
                G_A1[k - n] = np.transpose(W[2]) @ (G_A2[k - n] * A[k][3] * (1 - A[k][3]))
                b_temp = G_A1[k - n] * A[k][2] * (1 - A[k][2])
                G_B[1] += b_temp
                G_W[1] += b_temp @ np.transpose(A[k][1])
            for k in range(n, n + batch_size):
                G_A0[k - n] = np.transpose(W[1]) @ (G_A1[k - n] * A[k][2] * (1 - A[k][2]))
                b_temp = G_A0[k - n] * A[k][1] * (1 - A[k][1])
                G_B[0] += b_temp
                G_W[0] += b_temp @ np.transpose(A[k][0])
```

سپس طبق شبه کد داده شده، در هر **epoch** مجموعه داده ی آموزشی به ازای هر **batch size** به شبکه آموزش داده می شود. دو حالت برداری و با حلقه پیاده سازی شده که ابتدا حالت برداری توضیح داده می شود.

برای گرادیان وزن ها، بایاس ها و خروجی ها ابتدا آرایه هایی تعریف می شود و خروجی شبکه به ازای داده های **batch** حساب می شود.

سپس ابتدا گرادیان خروجی ها در لایه آخر محاسبه می شود و توسط آن گرادیان بایاس و سپس گرادیان وزن ها محاسبه می شود. این کار به ترتیب انجام می شود تا بار محاسباتی به حداقل برسد.

برای لایه های بعدی به همین ترتیب گرادیان ها محاسبه می شود.

```
# update W and B
for i in range(3):
    if learning_rate_method:
        if np.linalg.norm(G_W[i]):
            W[i] -= learning_rate * G_W[i] / np.linalg.norm(G_W[i])
        if np.linalg.norm(G_B[i]):
            B[i] -= learning_rate * G_B[i] / np.linalg.norm(G_B[i])
    else:
        W[i] -= learning_rate * G_W[i] / batch_size
        B[i] -= learning_rate * G_B[i] / batch_size
```

سپس با گرادیان های محاسبه شده وزن ها و بایاس ها را تغییر می دهیم.

دو حالت درنظر گرفته شده، در حالت اول گرادیان ها بر اندازه گرادیان تقسیم می شوند.

در حالت دوم طبق خواسته دستورکار بر **batch size** تقسیم می شود.

```
G_W = [np.zeros((150, 102)), np.zeros((60, 150)), np.zeros((4, 60))]
G_B = [np.zeros((150, 1)), np.zeros((60, 1)), np.zeros((4, 1))]
for i in range(n, n + batch_size):
    for j in range(3):
        A[i][j + 1] = sigmoid(W[j] @ A[i][j] + B[j])
# back propagation
G_A2 = np.zeros((batch_size, len(W[2])))
for i in range(len(W[2])):
    for k in range(n, n + batch_size):
        G_A2[k-n][i] = 2 * (A[k][3][i] - train_set[k][1][i])
        temp = G_A2[k - n][i] * A[k][3][i] * (1 - A[k][3][i])
        G_B[2][i] += temp
        for j in range(len(W[2][0])):
            G_W[2][i][j] += temp * A[k][2][j]

G_A1 = np.zeros((batch_size, len(W[1])))
for i in range(len(W[1])):
    for k in range(n, n + batch_size):
        for m in range(len(W[2])):
            G_A1[k-n][i] += G_A2[k-n][m] * A[k][3][m] * (1 - A[k][3][m]) * W[2][m][i]
        temp = G_A1[k - n][i] * A[k][2][i] * (1 - A[k][2][i])
        G_B[1][i] += temp
        for j in range(len(W[1][0])):
            G_W[1][i][j] += temp * A[k][1][j]
```

کد بالا مربوط به پیاده سازی با حلقه است. طرز کار مانند قسمت قبل است.

```
# shuffle and feed forward and calculate cost
random.shuffle(train_set[0:train_number])
for i in range(train_number):
    A[i][0] = train_set[i][0]
    for j in range(3):
        A[i][j + 1] = sigmoid(W[j] @ A[i][j] + B[j])
cost = 0
for i in range(train_number):
    cost += sum(np.power((A[i][3] - train_set[i][1]), 2))
costs_avg[epoch] += cost / train_number
```

سپس پس از پایان آموزش در هر **epoch**، داده های آموزشی **shuffle** می شوند و دوباره خروجی ها محاسبه می شود. در نهایت هزینه ها محاسبه شده و در آرایه هزینه ها اضافه می شود.

```
training_time.append(end_time - start_time)
# train set precision
true = 0
for i in range(train_number):
    if np.argmax(train_set[i][1]) == np.argmax(A[i][3]):
        true += 1
after_training_train_set_precision.append(true / train_number)
# test set feed forward and precision
C = []
for i in range(len(test_set)):
    C.append([test_set[i][0]])
    for j in range(3):
        C[i].append(sigmoid(W[j] @ C[i][j] + B[j]))
true = 0
for i in range(len(test_set)):
    if np.argmax(test_set[i][1]) == np.argmax(C[i][3]):
        true += 1
after_training_test_set_precision.append(true / len(test_set))
```

پس از پایان تمام **epoch** ها، زمان آموزش به آرایه زمان اضافه می شود.

و دقت شبکه پس از آموزش برای داده های آموزشی سنجیده می شود.

و درنهایت دقت شبکه برای داده های تست سنجیده می شود.

خروجی ها :

خروجی قدم دوم:

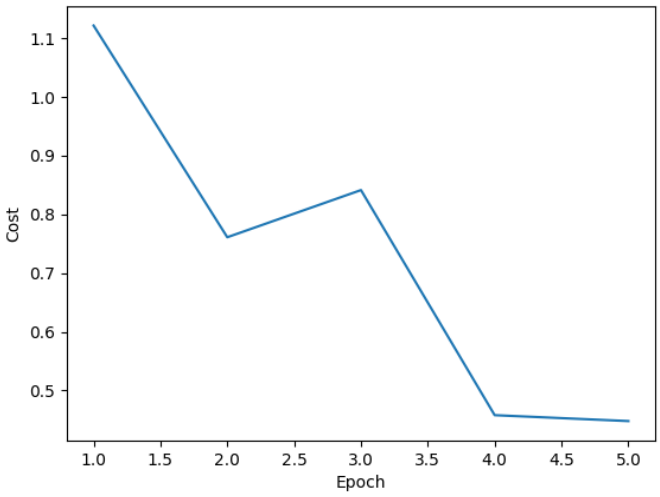
قبل از آموزش با ۲۰۰ داده خروجی ها محاسبه شد که دقت به طور میانگین برابر خروجی زیر است:

```
Before training precision: 25.4 %
```

خروجی قدم سوم:

```
# Number of code executions
iteration_number = 1
# Number of data for training
train_number = 200
epochs = 5
batch_size = 10
learning_rate = 1
# Learning rate method: 0 = divide by batch size , 1 = divide by gradian
learning_rate_method = 0
is_vectorized = False
```

خروجی به ازای مقادیر بالا تست شد که در پایین مشاهده می شود.



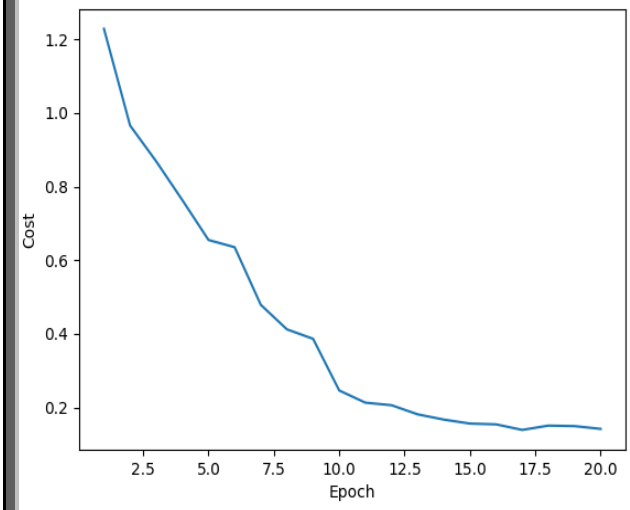

```
Before training precision: 27.0 %
After training precision(train set): 56.5 %
After training precision(test set): 50.15 %
Training time: 204.2 seconds
```

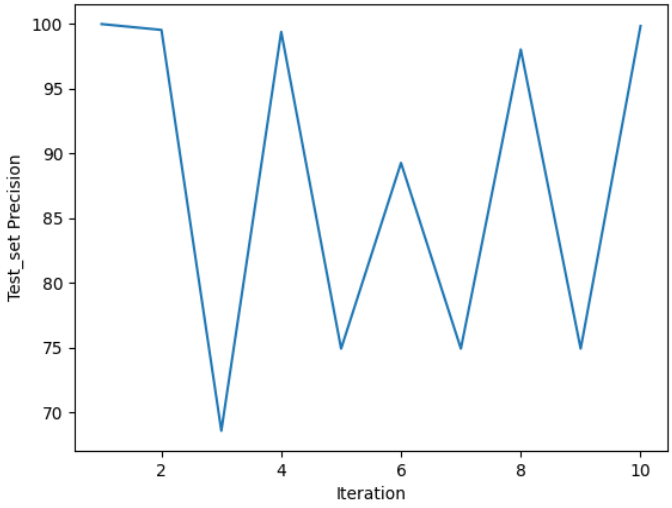
طبق خروجی بالا می بینیم که هزینه ها روند کاهشی دارند. همچنین دقت شبکه در داده های آموزشی ۵۶٪ و در داده های تست ۵۰٪ است.

خروجی قدم چهارم:

```
# Number of code executions
iteration_number = 10
# Number of data for training
train_number = 200
epochs = 20
batch_size = 10
learning_rate = 1
# Learning rate method: 0 = divide by batch size , 1 = divide by gradian
learning_rate_method = 0
is_vectorized = True
```

خروجی به ازای مقادیر بالا تست شد که نتیجه در پایین مشاهده می شود.





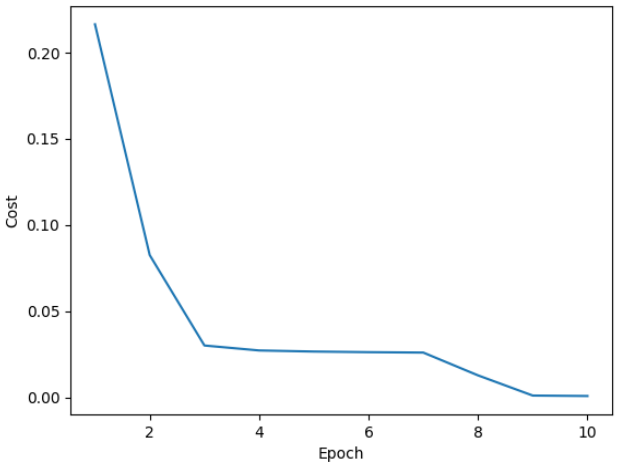

```
Before training precision: 23.95 %
After training precision(train set): 89.95 %
After training precision(test set): 87.95 %
Training time: 1.97 seconds
```

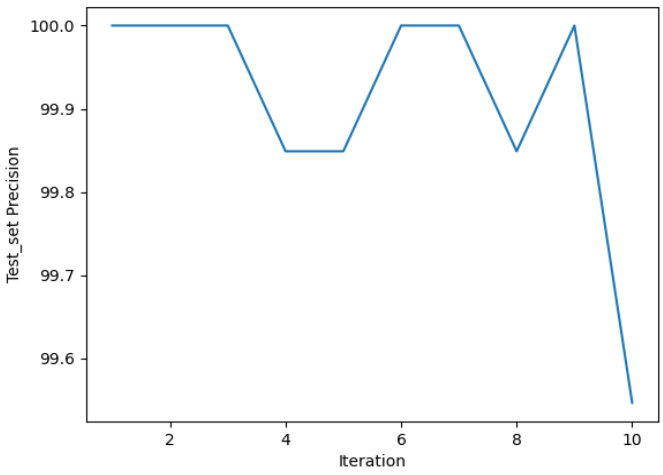
مطابق نتایج بالا، می بینیم هزینه روند کاهشی دارد. نمودار تکرار کد-دقت را نیز می بینیم که میانگین دقت تقریبا ۹۰ درصد است همچنین زمان یادگیری برای هر تکرار به طور میانگین ۲ ثانیه است.

خروجی قدم پنجم:

```
# Number of code executions
iteration_number = 10
# Number of data for training
train_number = len(train_set)
epochs = 10
batch_size = 10
learning_rate = 1
# Learning rate method: 0 = divide by batch size , 1 = divide by gradian
learning_rate_method = 0
is_vectorized = True
```

خروجی به ازای مقادیر بالا تست شد که نتیجه در پایین مشاهده می شود.






```
Before training precision: 23.26 %
After training precision(train set): 100.0 %
After training precision(test set): 99.91 %
Training time: 9.52 seconds
```

طبق نتایج بالا، می بینیم با آموزش روی کل داده ها دقت به شدت افزایش یافته است و تقریبا به ٪۱۰۰ رسیده است. ولی همچنان امکان گیرکردن در مینیمم محلی وجود دارد همانند مورد بالا که از **epoch** ۳ تا ۷ در مینیمم محلی بود که در **epoch** ۸ به سمت مینیمم کمتر می رود.

میانگین زمان آموزش برای هر بار اجرای کد ۹.۵ ثانیه است.

امتیازی ها :

امتیازی ۱:

مقادیر مختلف برای **learning rate, epoch, batch size** تست شد که نتیجه به صورت زیر است:

- به طور کلی زیاد شدن **epoch** تاثیر مثبتی در دقت و کاهش هزینه دارد و در اینجا زیاد بودن آن باعث کم شدن دقت در داده های تست و **overfit** نمی شود. علت آن شاید زیاد نبودن تعداد لایه ها و پرسپترون ها و کم بودن تعداد خروجی باشد.
- **batch size** به ازای مقادیر مختلف از ۱ تا ۱۰۰ تست شد که در اینجا چون تعداد کل داده ها ۱۹۶۲ است **batch size** بین ۲ تا ۲۰ بهترین نتیجه را از نظر دقت و کاهش هزینه دارد. به طور کلی هرچه قدر **batch size** کم تر می شود امکان گیرکردن در مینیمم محلی کمتر می شود.
- **learning rate** به ازای مقادیر ۰.۰۵ تا ۵ تست شد که به طور کلی **learning rate** بین ۰.۵ تا ۱.۱ نتیجه بهتری داشت و ۱ انتخاب بهتری نسبت به بقیه بود.

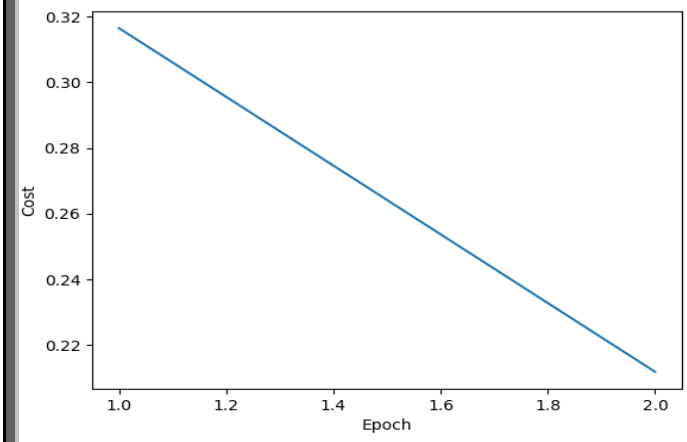
همچنین **learning rate** پویا نیز تست شد به طوری که بعد از هر **epoch** اگر **cost** بیشتر می شد مقدار **learning rate** در ۰.۹ ضرب می شد و در هر **epoch** که **cost** کمتر می شد مقدار **learning rate** برابر ۱ می شد.

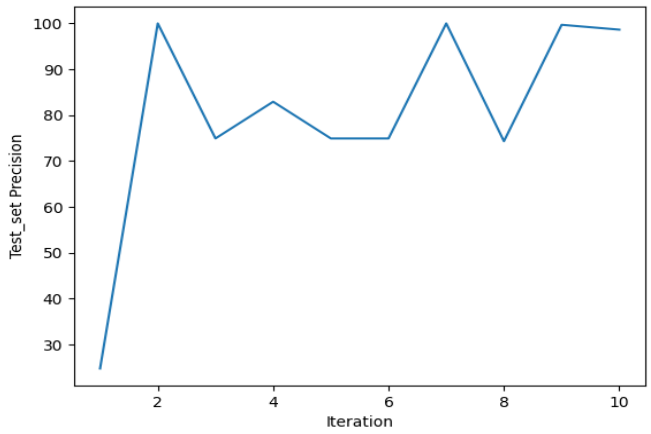
این کار نتیجه مثبتی داشت و میانگین دقت و کاهش هزینه بهتر می شد به طوری که امکان گیرکردن در مینیمم محلی کمتر می شد ولی هنوز احتمال گیرکردن وجود داشت.

امتیازی ۲:

یک تغییری که در شبه کد می توان داد این است که مقدار گرادیان وزن ها و بایاس به جای تقسیم شدن بر **batch size** ، بر اندازه گرادیان تقسیم شوند این کار تاثیر مثبت بسیار زیادی در افزایش میانگین دقت و سرعت بسیار زیاد در یادگیری و همچنین کاهش احتمال گیرکردن در مینیمم محلی دارد. به طوری که با انجام اینکار در **epoch 2** به میانگین دقت ۱۰۰٪ رسیدیم.

این مورد را با متغیر **learning_rate_method** پیاده سازی کردیم که نتیجه استفاده کردن و نکردن را در زیر می بینیم:



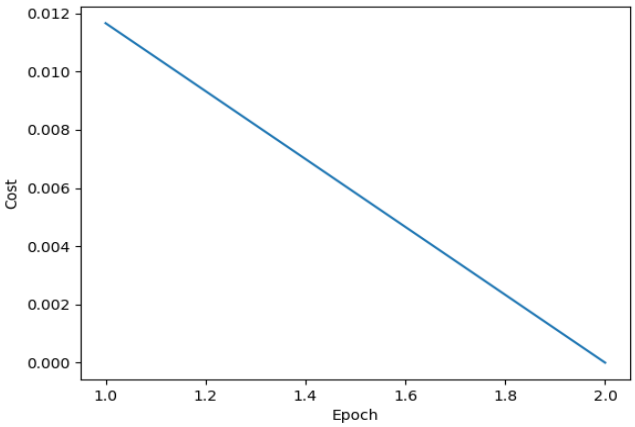


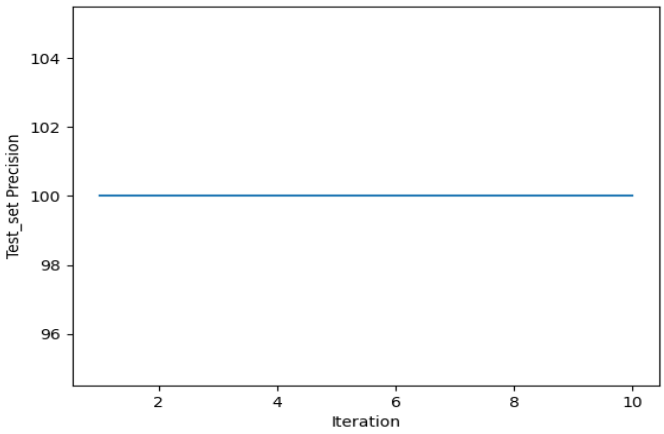

```
Before training precision: 25.0 %
After training precision(train set): 81.1 %
After training precision(test set): 80.51 %
Training time: 1.92 seconds
```

نتایج بالا مربوط به حالت عادی با متغیر های زیر است.(**epoch=2**)

```
# Number of code executions
iteration_number = 10
# Number of data for training
train_number = len(train_set)
epochs = 2
batch_size = 10
learning_rate = 1
# Learning rate method: 0 = divide by batch size , 1 = divide by gradian norm
learning_rate_method = 0
is_vectorized = True
```

تنها با تغییر **learning_rate_method** به نتایج زیر می رسیم:



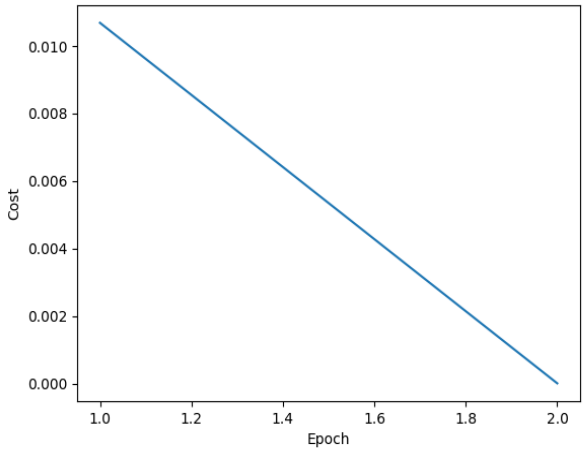


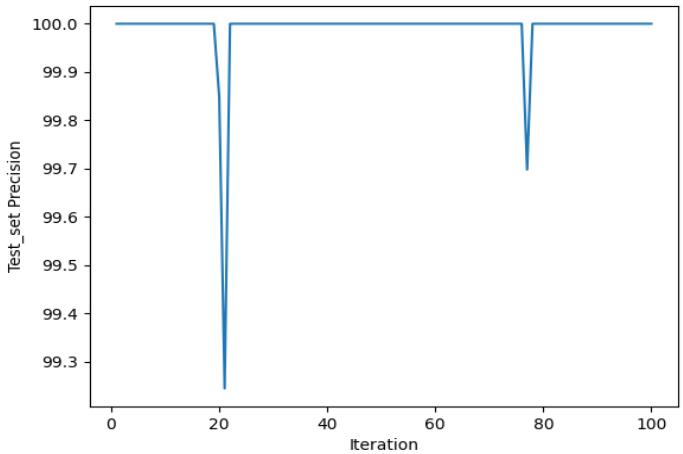

```
Before training precision: 26.18 %
After training precision(train set): 100.0 %
After training precision(test set): 100.0 %
Training time: 2.03 seconds
```

همانطور که می بینیم نتیجه به طور قابل توجهی بهبود می یابد به طوری که در **epoch** ۲ به میانگین دقت ۱۰۰ درصد می رسیم.

همچنین احتمال گیرکردن در مینیمم محلی را با اجرای ۱۰۰ بار کد با **epoch=2** بررسی می کنیم که نتیجه را در زیر می بینیم:

```
Before training precision: 24.09 %
After training precision(train set): 100.0 %
After training precision(test set): 99.99 %
Training time: 2.02 seconds
```





همانطور که می بینیم با **epoch=2** که خیلی کم است، نتیجه بسیار عالی است و از ۱۰۰ بار تکرار تنها ۲ بار دقت زیر ۱۰۰٪ است که آن هم بالای ۹۹ درصد است. که با افزایش **epoch** این موارد نیز برطرف می شود.