به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق

تمرین اول

نام و نام خانوادگی	مهسا همت‌پناه – محمد هادی بابالو
شماره دانشجویی	۸۱۰۹۹۵۸۴ - ۸۱۰۱۹۹۳۸۰
تاریخ ارسال گزارش	۱۴۰۲,۰۸,۱۷

فهرست

شکل‌ها

جدول‌ها

پاسخ ۱. شبکه عصبی Mcculloch-Pitts

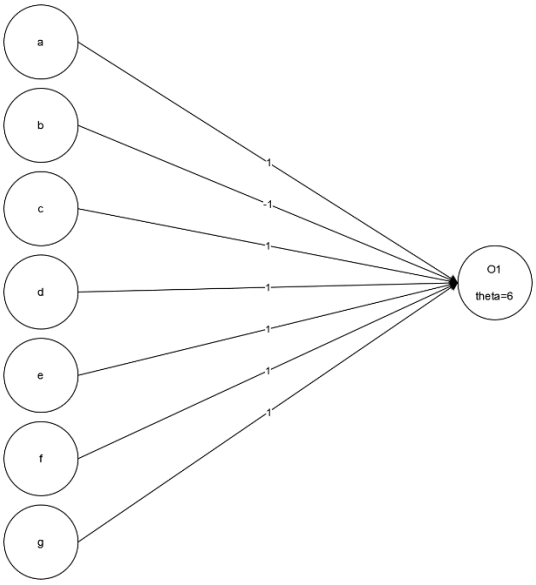
۱-۱. نمایشگر 7-segment

۱-۲. شبکه عصبی یک لایه

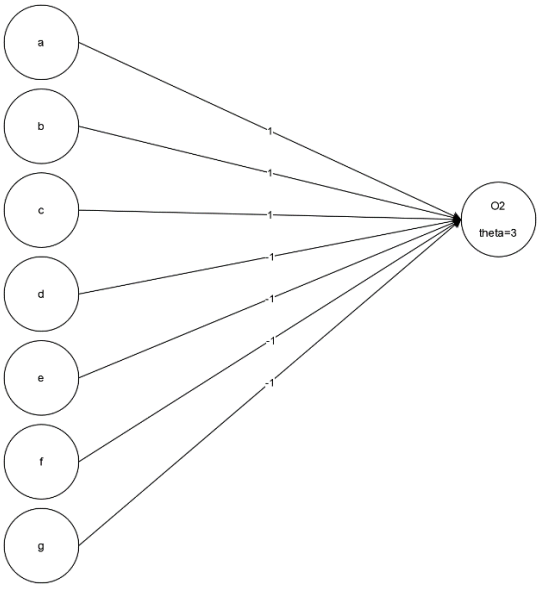
a	b	c	d	e	f	g	O_1	O_2	O_3	O_4
1	0	1	1	1	1	1	1	0	0	0
1	1	1	0	0	0	0	0	1	0	0
1	1	1	1	1	1	1	0	0	1	0
1	1	0	1	1	1	1	0	0	0	1
Any other combination							0	0	0	0

شکل ۱-۱. عملکرد شبکه به ازای هر ورودی

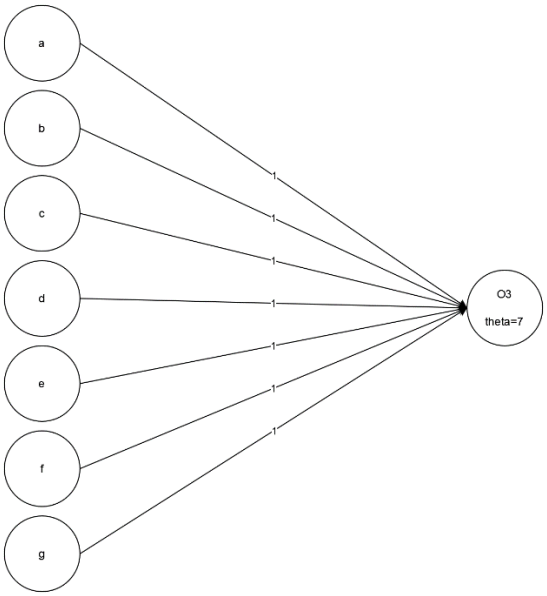
برای تعیین وزن‌های این شبکه بر اساس معماری این شبکه که در شکل ۱-۲ مشخص است، به این صورت عمل می‌کنیم که به ازای ۴ سطر اول جدول درستی شکل ۱-۱ وزن‌های متناظر با هر نورون خروجی را تعیین می‌کنیم. این کار با استفاده از پیاده‌سازی توابع AND و NOT می‌کنیم. به این صورت که به ازای هر سطر جدول، وزن یال متصل‌کننده ورودی به خروجی در صورت ۱ بودن خانه جدول برابر ۱ و در صورت ۰ بودن آن، وزن برابر منفی ۱ قرار می‌گیرد. همچنین threshold نورون خروجی هم برابر با تعداد یال‌های با وزن ۱ متصل به این نورون خروجی قرار می‌گیرد. مثلا برای پیاده‌سازی سطر اول جدول برای نورون خروجی O1، وزن همه یال‌های متصل به آن غیر از یال دوم که ۱- است، برابر ۱ است و تتا هم برابر با ۶ خواهد بود. وزن‌های کامل این شبکه در شکل‌های زیر قابل مشاهده هستند.



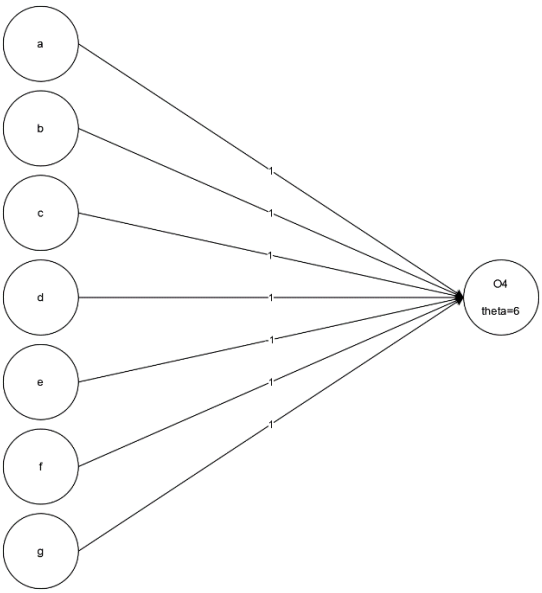
شکل ۱-۲. وزن‌های شبکه مربوط به O1



شکل ۱-۳. وزن‌های شبکه مربوط به O2



شکل ۱-۴. وزن‌های شبکه مربوط به O3



شکل ۱-۵. وزن‌های شبکه مربوط به O4

همچنین برای آزمایش عملکرد این مدل هم از پیاده‌سازی زیر استفاده شده است.

```
oneLayerNet.linear1.weight = nn.Parameter(torch.tensor([[1., -1., 1., 1., 1., 1., 1.],
                                                        [1., 1., 1., -1., -1., -1., -1.],
                                                        [1., 1., 1., 1., 1., 1., 1.],
                                                        [1., 1., 1., 1., -1., 1., 1.]]))

oneLayerNet.linear1.bias = nn.Parameter(torch.tensor([-6., -3., -7., -6.]))
```

شکل ۱-۶. تخصیص وزن‌های شبکه

```
def eval_model(model, data_loader):

    model.eval()

    num_preds = 0
    true_preds = 0

    with torch.no_grad():
        for data, label in data_loader:
            preds = model(data)
            pred_labels = (preds)
            num_preds += len(preds) * 4
            true_preds += (pred_labels == label).sum().item()

            # print wrong predictions
            for i in range(len(preds)):
                if not torch.all(pred_labels[i] == label[i]):
                    print(f'Input: {data[i]}')
                    print(f'Predicted: {pred_labels[i]} | Actual: {label[i]}')
                    print()

    return true_preds, num_preds
```

```
oneLayerNet_true_preds, oneLayerNet_num_preds = eval_model(oneLayerNet, data_loader)

print(f'Accuracy: {(oneLayerNet_true_preds/oneLayerNet_num_preds)*100:.2f}')
```

```
Accuracy: 100.00
```

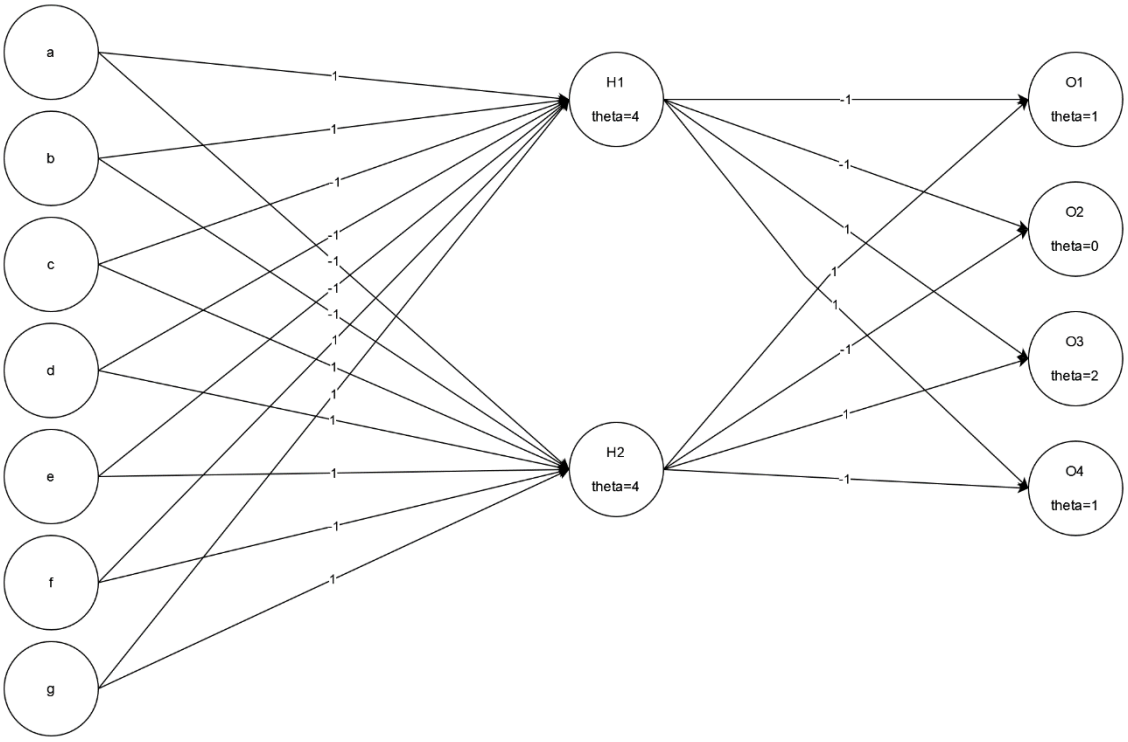
شکل ۱-۷. ارزیابی عملکرد مدل

همانطور که در شکل ۱-۷ هم دیده می‌شود، مدل مطابق انتظار ما به ازای همه ۱۲۸ حالت ممکن ورودی، پاسخ صحیح را تولید می‌کند.

۱-۳. شبکه عصبی دو لایه

الف)

برای تعیین وزن‌های این شبکه، شبکه را به دو بخش چپ و راست تقسیم می‌کنیم و در هر بخش از روشی مثل روش استفاده شده در بخش قبلی برای تعیین وزن‌ها استفاده می‌کنیم. وزن‌های نهایی ما در نهایت مطابق با شکل ۱-۸ خواهد بود.



شکل ۱-۸. وزن‌های نهایی شبکه عصبی دو لایه

عملکرد این مدل هم در شکل‌های زیر قابل مشاهده است:

```
twoLayerNet.linear1.weight = nn.Parameter(torch.tensor([[1., 1., 0., 0., 0., 1., 1.],
                                                        [0., 0., 1., 1., 1., 0., 1.]]))

twoLayerNet.linear1.bias = nn.Parameter(torch.tensor([-4., -4.]))
```

```
twoLayerNet.linear2.weight = nn.Parameter(torch.tensor([[-1., 1.],
                                                        [-1., -1.],
                                                        [1., 1.],
                                                        [1., -1.]]))

twoLayerNet.linear2.bias = nn.Parameter(torch.tensor([-1., 0., -2., -1.]))
```

شکل ۱-۹. تخصیص وزن‌های شبکه دو لایه

```
digits = [torch.tensor([1, 0, 1, 1, 1, 1, 1], dtype=torch.float32),
            torch.tensor([1, 1, 1, 0, 0, 0, 0], dtype=torch.float32),
            torch.tensor([1, 1, 1, 1, 1, 1, 1], dtype=torch.float32),
            torch.tensor([1, 1, 1, 1, 0, 1, 1], dtype=torch.float32)]
```

```
labels = [torch.tensor([1, 0, 0, 0], dtype=torch.float32),
            torch.tensor([0, 1, 0, 0], dtype=torch.float32),
            torch.tensor([0, 0, 1, 0], dtype=torch.float32),
            torch.tensor([0, 0, 0, 1], dtype=torch.float32)]
```

```
num_preds = 0
true_preds = 0

for i in range(len(digits)):
    pred = twoLayerNet(digits[i])

    num_preds += 4
    true_preds += (pred == labels[i]).sum().item()

    if not torch.all(pred == labels[i]):
        print(f'Digit: {i+6}')
        print(f'Input: {digits[i]}')
        print(f'Predicted: {pred} | Actual: {labels[i]}')
        print()

print(f'Accuracy: {(true_preds/num_preds)*100:.2f}')
```

```
Accuracy: 100.00
```

شکل ۱-۱۰. ارزیابی عملکرد مدل دو لایه روی ورودی‌های مورد نظر

ب)

نورون ابتدایی لایه پنهان، مسئول تشخیص الگوی مربع روشن در ۴ سگمنت بالایی است که برای اعداد ۸ و ۹ فعال خواهد شد. نورون بعدی لایه پنهان الگوی مربع روشن در ۴ سگمنت پایینی را تشخیص می‌دهد و با اعداد ۶ و ۸ فعال می‌شود. از روی این نورون‌ها و با دادن وزن‌های مناسب می‌توان به خروجی‌های مناسب دست پیدا کرد. چونکه در عدد ۸ هر دو نورون میانی فعال می‌شوند، در عدد ۷ هیچ‌کدام فعال می‌شوند، در عدد ۶ صرفا نورون پایینی فعال می‌شود و در عدد ۹ هم صرفا نورون میانی بالایی فعال می‌شود.

ج)

برای شبکه اول ۷ نورون ورودی و ۴ نورون خروجی داریم که هر نورون ورودی به تمام نورون‌های خروجی متصل شده است. همچنین ۴ ترشهولد برای نورون‌های خارجی هم نیاز داریم. در نتیجه تعداد کل پارامترهای مدل برابر است با:

$$7 \times 4 + 4 = 28 + 4 = 32$$

برای شبکه دوم هم به طور مشابه محاسبات را بخش چپ و راست انجام داده و با هم جمع می‌کنیم:

$$Left: 7 \times 2 + 2 = 14 + 2 = 16$$

$$Right: 2 \times 4 + 4 = 8 + 4 = 12$$

$$Total: 16 + 12 = 28$$

همانطور که از محاسبات مشخص است تعداد پارامترهای مدل اول بیشتر است و همچنین قدرت پردازشی بیشتری هم دارد و می‌تواند برای تمام ۱۲۸ حالت ممکن درست خروجی تولید کند در صورتی که مدل دوم اینطور نیست و صرفا برای این ۴ ورودی تضمین می‌شود که خروجی صحیح را تولید کند.

پاسخ ۲ – آموزش شبکه های Adaline وMadaline

۲-۱. Adaline

الف)

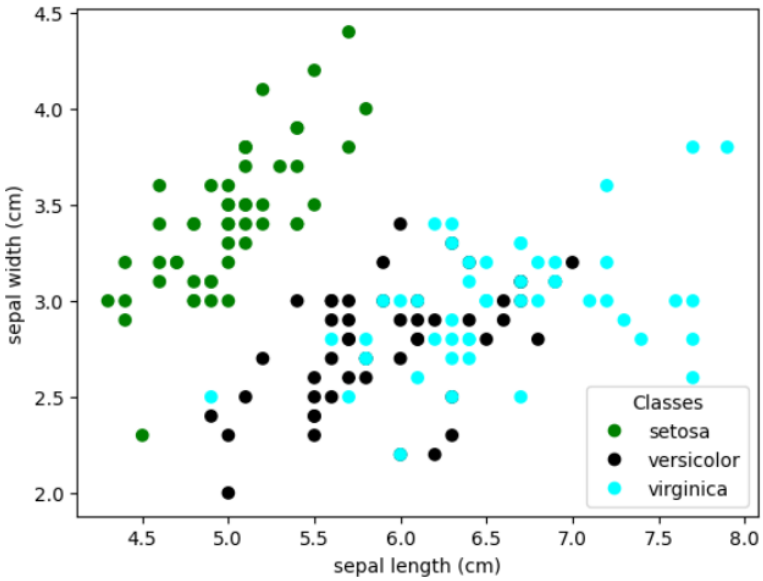
ابتدا مجموعه داده Iris (که از ۳ نوع عنبیه مختلف Setosa،Versicolour و Virginica تشکیل شده) را با استفاده از پکیج sklearn. Datasets لود می کنیم و برای قسمت الف، نوع Setosa را از سایر دسته ها جدا می کنیم تا دو دسته Setosa و Non_ Setosa داشته باشیم.

```
iris = datasets.load_iris(return_X_y=False)
data = pd.DataFrame(data=iris.data, columns=iris.feature_names)
data['target'] = iris.target
```

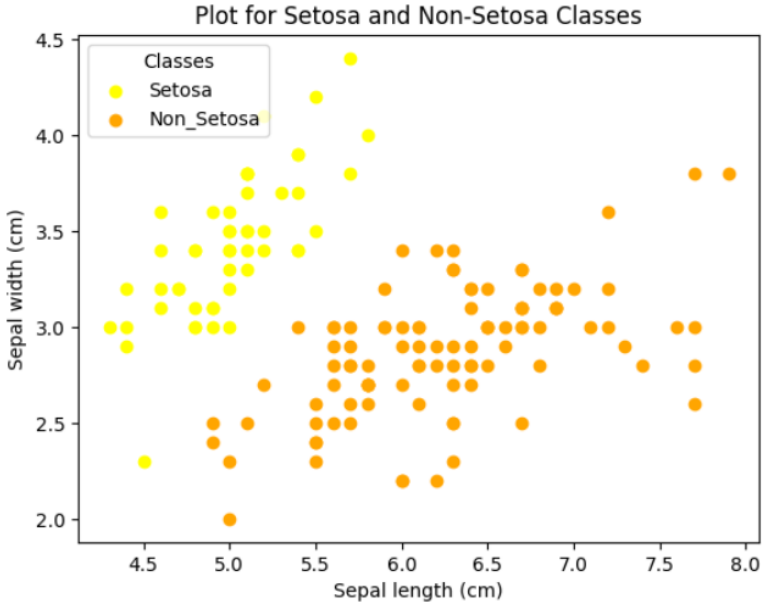
```
# make a copy of iris dataset in which targets are setosa and non-setosa
iris_setosa = data.copy()
iris_setosa['target'] = np.where(iris_setosa['target'] == 0, -1, 1)

# Select only two features: Sepal length (0) and Sepal width (1)
X_setosa = iris_setosa.iloc[:, [0, 1]].values
y_setosa = iris_setosa['target'].values
```

شکل ۲-۱. لود کردن و جداسازی داده ها

نمودار پراکندگی داده ها با استفاده از کتابخونه matplotlib رسم شده است. شکل ۲-۲ نشان دهنده نمودار پراکندگی داده ها در دو بعد Sepal-length و Sepal-width است و شکل ۳-۲ نشان دهنده نمودار پراکندگی داده های جدا شده در دو دسته Setosa و Non_ Setosa در دو بعد Sepal-length و Sepal-width است.



شکل ۲-۲. نمودار پراکندگی داده ها



شکل ۲-۳. نمودار پراکندگی داده ها در دو کلاس Setosa و Non_ Setosa

در ادامه یک شبکه Adaline را که با توجه به الگوریتمی که در کتاب رفرنس آمده است پیاده سازی کرده ایم و شبکه را روی این داده ها آموزش دادیم.

در ادامه توضیح کوتاهی از نحوه پیاده سازی آورده شده است.

- در تابع fit ابتدا وزن ها (w) مقدار دهی می شوند و در مرحله forward وزن ها در ورودی های (x) شبکه طبق معادله زیر، ضرب می شوند.

$$net = \sum_{i=1}^{n} w_i x_i + w_0$$

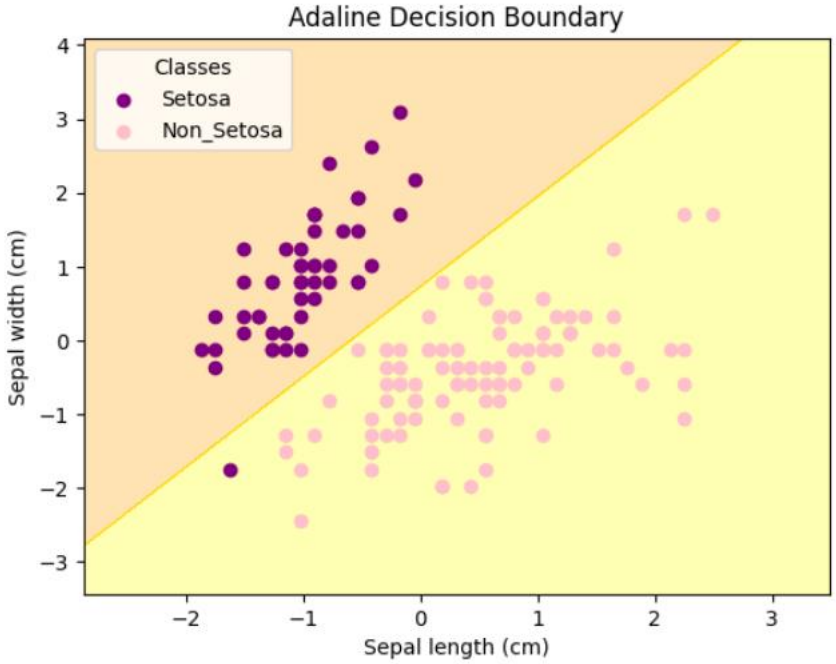
- برای تابع فعال ساز از تابع sign که تابع فعال ساز در الگوریتم اصلی است، استفاده شده است.
- مقدار تابع هزینه که طبق معادله زیر طراحی شده است نیز در ارایه cost ذخیره می شود.

$$cost = \frac{1}{2}(t - net)^2$$

- وزن ها طبق فرمول زیر با توجه به داده هایی که اشتباه دسته بندی شده اند، اپدیت می شوند.

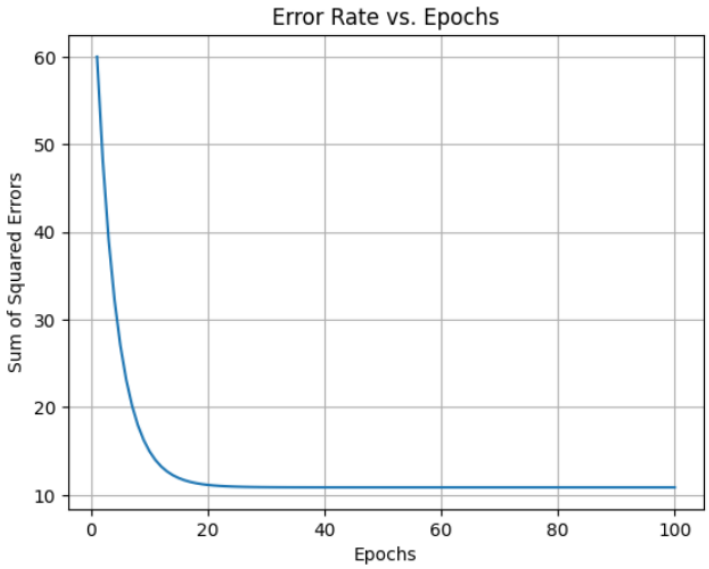
$$w_i(new) = w_i(old) + \alpha(t - net)x_i$$

شکل ۲-۴ خط جدا کننده محاسبه شده توسط شبکه Aadaline را نشان می دهد. همان طور که در شکل قابل مشاهده است، چون کلاس Setosa به صورت خطی از دو دسته دیگر جدایی پذیر است پس با مدل خطی Aadaline می توان آنها را جدا کرد.



شکل ۲-۴. نمودار پراکندگی همراه با خط جدا کننده بخش الف

شکل ۲-۵ نشان دهنده نمودار تغییرات تابع هزینه در طول ایپاک های مختلف است. با تعداد محدودی ایپاک، خطا کم شده است پس یعنی داده ها به راحتی قابل تفکیک هستند و نیاز به مدل پیچیده ندارند.

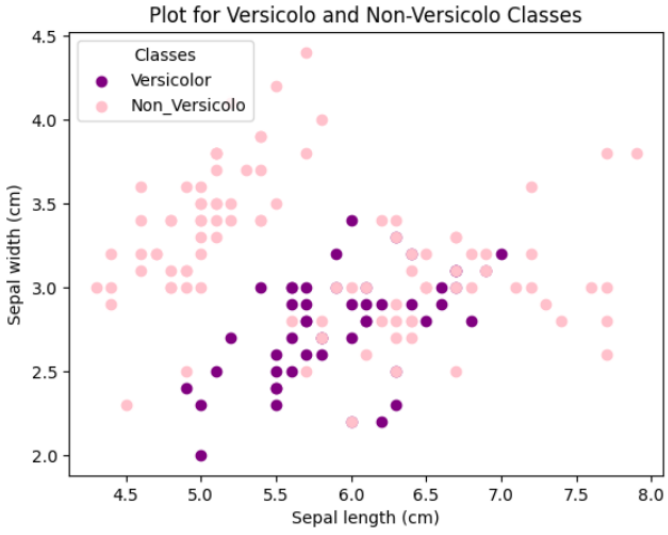


شکل ۲-۵. نمودار تابع هزینه درگذر ایپاک مدل بخش الف

جدول ۲-۱. دقت مدل برای داده تست و ترین بخش الف

	Train dataset	Test dataset
Accuracy	0.9916666666666667	1.0

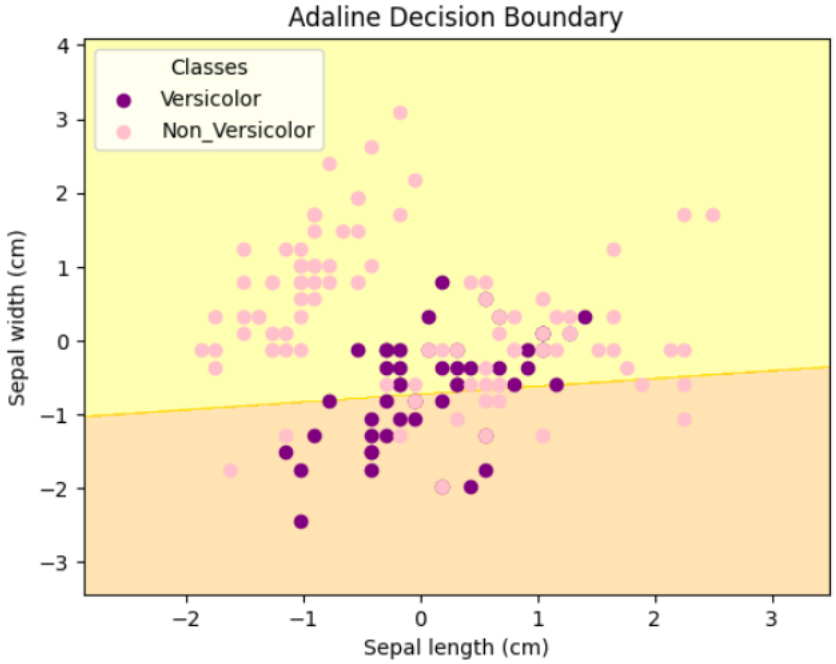
ب)



شکل ۲-۶. نمودار پراکندگی داده ها در دو کلاس Versicolour و Non_ Versicolour

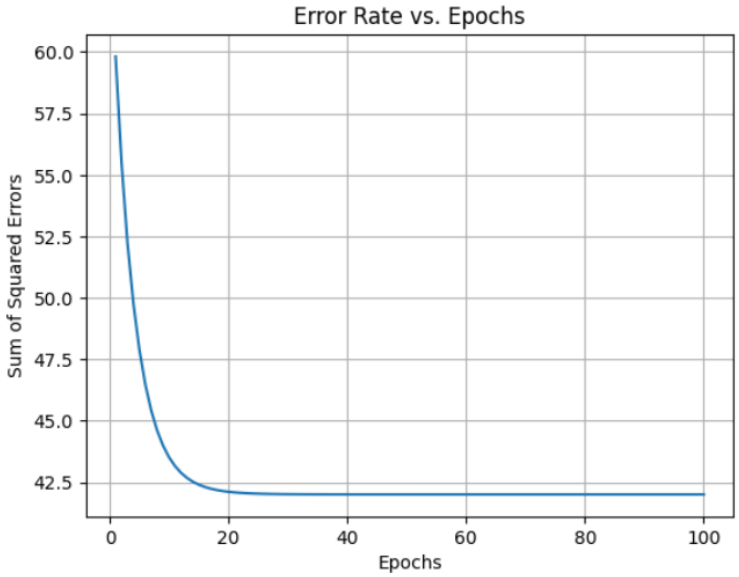
برای قسمت ب داده ها را به دو کلاس Non-Versicolour , Versicolour تقسیم می کنیم و آموزش را بر روی این داده ها انجام می دهیم.

در این حالت چون کلاس Versicolour بر خلاف کلاس Setosa در قسمت الف، به صورت خطی از دو دسته دیگر کاملا جدایی پذیر نیست، تعداد قابل توجهی از داده ها اشتباه دسته بندی شده اند(شکل ۲-۷). با توجه به این موضوع با مدل خطی و ساده ای مثل Adaline نمی توان آن ها را کاملا جدا کرد و نیاز به مدل پیچیده تری است. اما در کل این مدل توانسته به دقت خوبی برسد.(جدول ۲-۲)



شکل ۲-۷. نمودار پراکندگی همراه با خط جدا کننده بخش ب

شکل ۲-۸ نشان دهنده نمودار تغییرات تابع هزینه در طول ایپاک است.



شکل ۲-۸. نمودار تابع هزینه درگذر ایپاک مدل بخش ب

جدول ۲-۲. دقت مدل برای داده تست و ترین بخش ب

	Train dataset	Test dataset
Accuracy	0.7083333333333334	0.7333333333333333

Madaline .۲-۲

الف)

MRI: این شبکه دو لایه دارد یک لایه پنهان (تشکیل شده از یونیت های Adaline) و یک لایه خروجی. در این شبکه وزنهای لایه خروجی ثابت هستند و عموما لایه خروجی منطق OR را دارد. وزنها و بایاس این لایه به طوری تنظیم می شوند که OR خروجی نورون های لایه پنهان را بسازند. البته می توان به جای OR منطق AND هم استفاده کرد. نسخه های پیشرفته تر با تعداد لایه های پنهان بیشتر هم وجود دارد که به آن در کتاب پرداخته نشده است. در طراحی این شبکه با تعداد نورون های متفاوت معمولا وزن های لایه خروجی همه ۱ قرار داده می شود و مقدار بایاس این لایه نیز برابر با $1 - \#\text{hidden neurons}$ قرار داده می شود تا تنها زمانی خروجی مدل ۱- شود که همه نورون ها مقدار ۱- داشته باشند. با این کار منطق OR به درستی پیاده سازی می شود.

الگوریتم MRI به شرح زیر است:

مرحله 0 . ابتدا وزنها و بایاس رو با مقادیر کوچک تصادفی، مقداردهی اولیه می کنیم.

مرحله۱ . حلقه می زنیم و تا زمانی که شرط خاتمه true نباشد مراحل ۲ تا ۸ را انجام می دهیم:

مرحله ۲. برای هر جفت ورودی s:t (s فیچر ها و t لیبل داده است) مراحل ۳ تا ۷ را تکرار می کنیم.

مرحله ۳. مقدار ورودی ها را ست می کنیم.

$$x_i = s_i$$

مرحله ۴. مقدار ورودی هر نورون لایه مخفی را محاسبه می کنیم.

$$z_{in} = b + \sum_{i=1}^{n} w_i x_i$$

مرحله ۵. مقدار خروجی را برای هر نورون لایه مخفی بدست میاوریم.

$$z = f(z_{in})$$

. مرحله ۶. خروجی شبکه را بدست می آوریم . برای این کارخروجی هر نورون را به تابع فعال ساز Sign میدهیم که اگر خروجی مثبت باشد یک و اگر خروجی منفی باشد منفی یک را برمی گرداند.V وزن های ثابت لایه خروجی هستند.

$$y_{in} = b_{out} + \sum_{i=1}^{n} z_i v_i$$

مرحله ۷. خطا را محاسبه می کنیم و وزن ها را آپدیت می کنیم.

- اگر درست لیبل زده باشم نیازی به آپدیت وزن نیست در غیر اینصورت:

- اگر لیبل درست ۱ باشد و ما ۱- لیبل زده باشیم یعنب تمام خروجی های نورونهای لایه مخفی منفی هستند پس اگر یکی را مثبت کنیم خط جدا ساز بهتر میشود. برای همین نزدیکترین وزن به صفر را آپدیت میکنیم.

$$w_{i_J}(new) = w_{i_J}(old) + \alpha(1 - z_{in_J})x_i$$

$$b_J(new) = b_J(old) + \alpha(1 - z_{in_J})$$

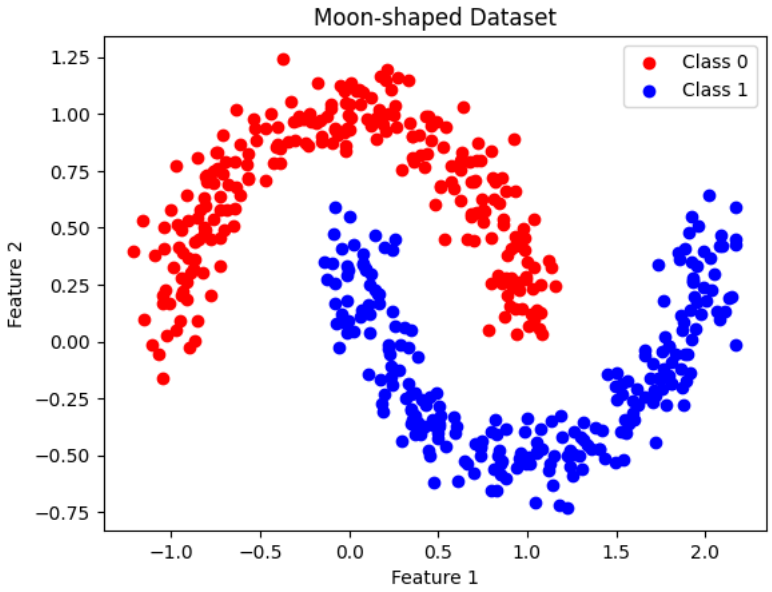
- اگر لیبل درست ۱- باشد یعنی لیبل ما ۱ است پس برای درست کردن باید تمام وزن های مثبت را کم کنیم تا منفی یا نزدیک منفی شوند.

$$w_{i_K}(new) = w_{i_K}(old) + \alpha(-1 - z_{in_K})x_i$$

$$b_K(new) = b_K(old) + \alpha(-1 - z_{in_K})$$

مرحله ۸. چک کردن شرط خاتمه. اگر تغییر وزن متوقف شود یا به حداکثر تعداد ایپاک ها برسیم یا به ارور قابل قبولی برسیم، متوقف می شویم.

ب)



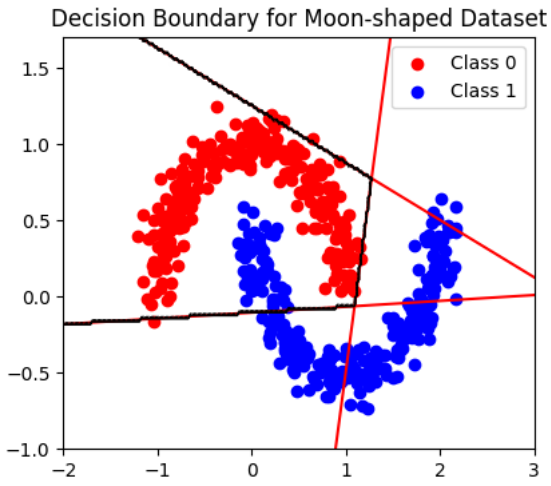
شکل ۲-۹. توزیع داده moon shaped

ابتدا با استفاده از دستور های داده شده در دستور کار داده های moon shaped را load می کنیم. شکل ۲-۹ پراکندگی داده ها را نشان می دهد. همانطور که در شکل مشخص است داده ها به صورت خطی و قابل جدا سازی نیستند. این پیچیدگی استفاده از Madeline را برای دسته بندی این داده توجیه می کند.

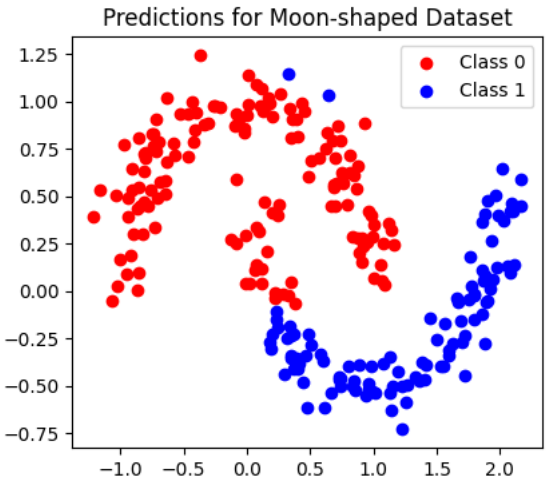
در همه حالات شبکه با مقادیر مختلف hyperparameters (نرخ یادگیری و ایپاک) آموزش داده شده است و بهترین نتیجه در ادامه آورده شده است

- ۳ نورون

شکل ۲-۱۰ نشان دهنده ۳ خط جداساز و شکل ۲-۱۱ نشان دهنده دسته بندی پیش بینی شده شبکه Madeline با ۳ نورون و ۵۰۰ ایپاک و نرخ یادگیری 0.01 است. دقت بدست آمده در این حالت برابر 0.9 است.



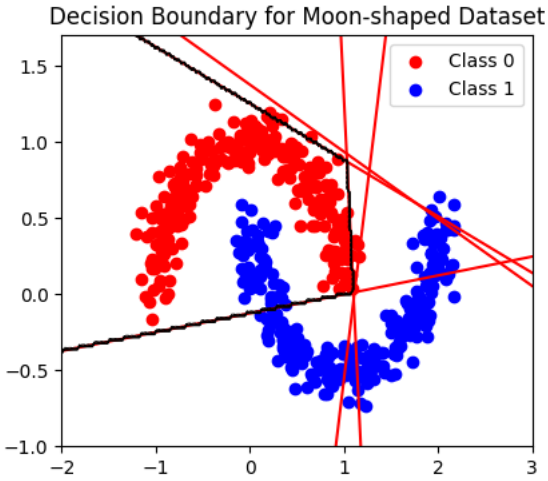
شکل ۲-۱۰. خط های جداساز برای شبکه **Madaline** با ۳ نورون



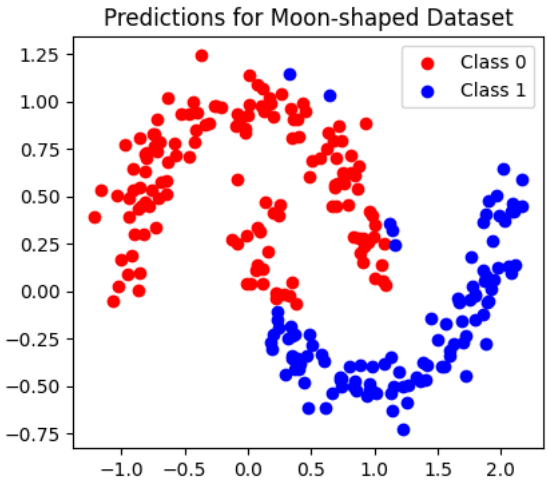
شکل ۲-۱۱. دسته بندی بدست آمده از شبکه **Madaline** با ۳ نورون

- ۵ نورون

شکل ۲-۱۲ نشان دهنده ۳ خط جداساز و شکل ۲-۱۳ نشان دهنده دسته بندی پیش بینی شده شبکه Madeline با ۳ نورون و ۵۰۰ ایپاک و نرخ یادگیری 0.01 است. دقت بدست آمده در این حالت برابر 0.888 است.



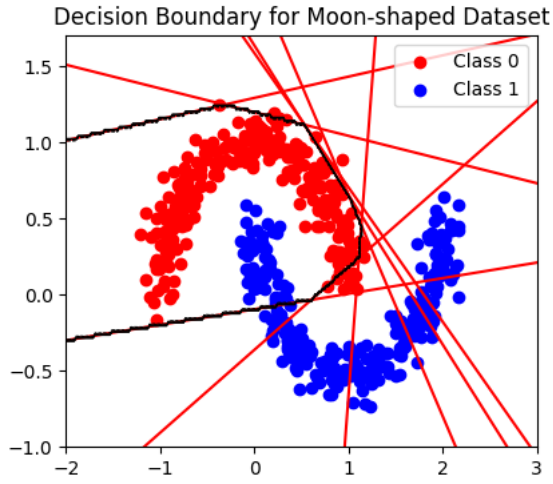
شکل ۲-۱۲. خط های جداساز برای شبکه **Madaline** با ۵ نورون



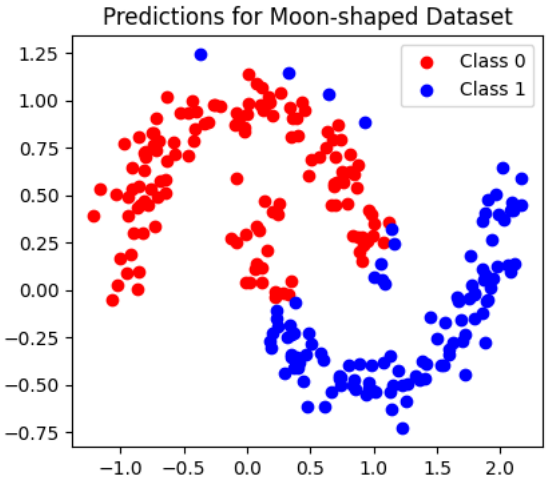
شکل ۲-۱۳. دسته بندی بدست آمده از شبکه **Madaline** با ۵ نورون

- ۸ نورون

شکل ۲-۱۴ نشان دهنده ۳ خط جداساز و شکل ۲-۱۵ نشان دهنده دسته بندی پیش بینی شده شبکه Madeline با ۳ نورون و ۵۰۰ ایپاک و نرخ یادگیری 0.01 است. دقت بدست آمده در این حالت برابر 0.872 است.



شکل ۲-۱۴. خط های جداساز برای شبکه Madaline با ۸ نورون



شکل ۲-۱۵. دسته بندی بدست آمده از شبکه Madaline با ۸ نورون

ج)

از بین ۳ حالت تست شده بیشترین دقت مدل در حالت۳ نورون است و کمترین یا ۸ نورون.

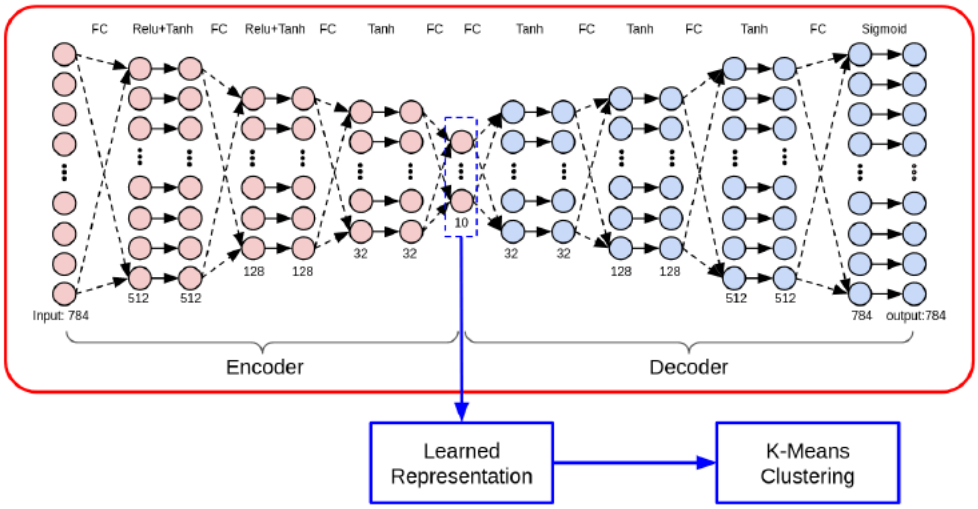
به طور کلی افزایش ایپاک به بهبود عملکرد مدل کمک می کند. اما از طرفی تعداد ایپاک خیلی بزرگ نیز باعث overfitting مدل می شود.

پاسخ ۳ – خوشه بندی با استفاده از Autoencoder

۳-۱. پیاده سازی Deep Autoencoder برای کاهش ابعاد داده ها

۱.

در این بخش مدل طراحی شده در مقاله " DAC: Deep Autoencoder-based Clustering, a General Deep Learing Framework of Representation Learning" را پیاده سازی می کنیم.



شکل ۳-۱. طراحی Autoencoder

Encoder: دارای ۸ لایه است که شامل لایه ورودی و لایه خروجی نمایش آموخته است. لایه ها fully connected هستند. همچنین دو نوع لایه تابع فعال سازی دارد، Relu و Tanh. لایه های Relu می تواند غیرخطی بودن مدل را معرفی کند و آن را در برابر داده های ورودی غیر خطی قوی تر کند. از طرف دیگر، لایه Tanh می تواند داده ها را در رنج (0, 1) قرار دهد.

Decoder: دارای ۹ لایه است که شامل لایه ورودی (خروجی رمزگذار) و لایه خروجی نهایی است. هر لایه کوچکتر به طور کامل (fully connected) به لایه بزرگتر بعدی و سپس یک لایه فعال سازی Tanh متصل می شود. علاوه بر این، لایه خروجی، لایه فعال سازی sigmoid است تا مقادیر خروجی را در محدوده (0, 1) اعمال کند.

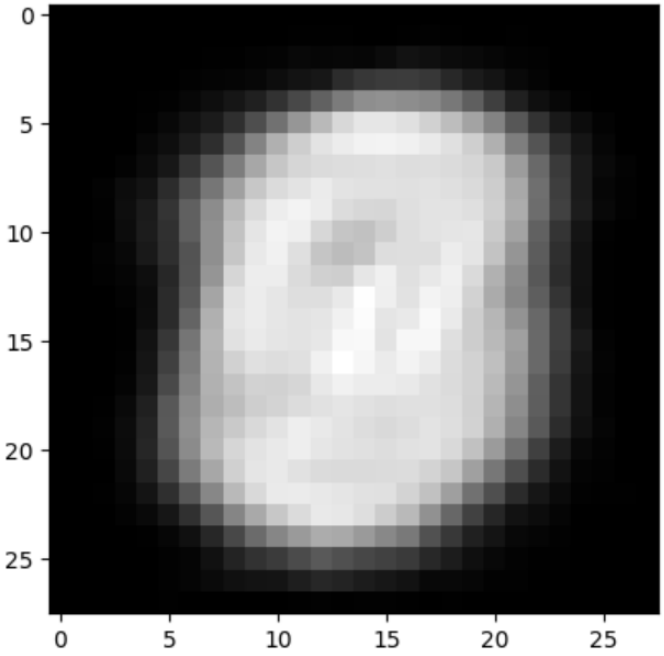
.۲

در ادامه با استفاده از فرمول زیر تابع CustomWeights را تعریف می کنیم. طبق فرض مقاله، ۱۰۰۰ داده را از بین داده های آموزش انتخاب می کنیم و تفاوت درون دسته ای و بین دسته ای را برای هر فیچر در این داده ها بدست می آوریم. حاصل وزنی برای هر فیچر است تا در فرایند آموزش به فیچر هایی که در شناسایی اعداد مهم تر هستند، بیشتر توجه بشود. برای مثال پیکسل هایی که در اطراف عکس ها هستند، در همه عکس ها مشابه به هم هستند و کمکی به شناسایی عدد موجود در عکس نمی کنند پس وزن نزدیک به صفر می گیرند و کمتر در جریان آپدیت شدن وزن ها در autoencoder مورد اهمیت قرار داده می شوند.

$$w_i = \frac{\sum_{l_p=l_q} e^{-(x_{ip}-x_{iq})^2}}{\sum_{l_p=l_q} 1} \cdot \frac{\sum_{l_p\neq l_q} (1-e^{-(x_{ip}-x_{iq})^2})}{\sum_{l_p\neq l_q} 1}$$

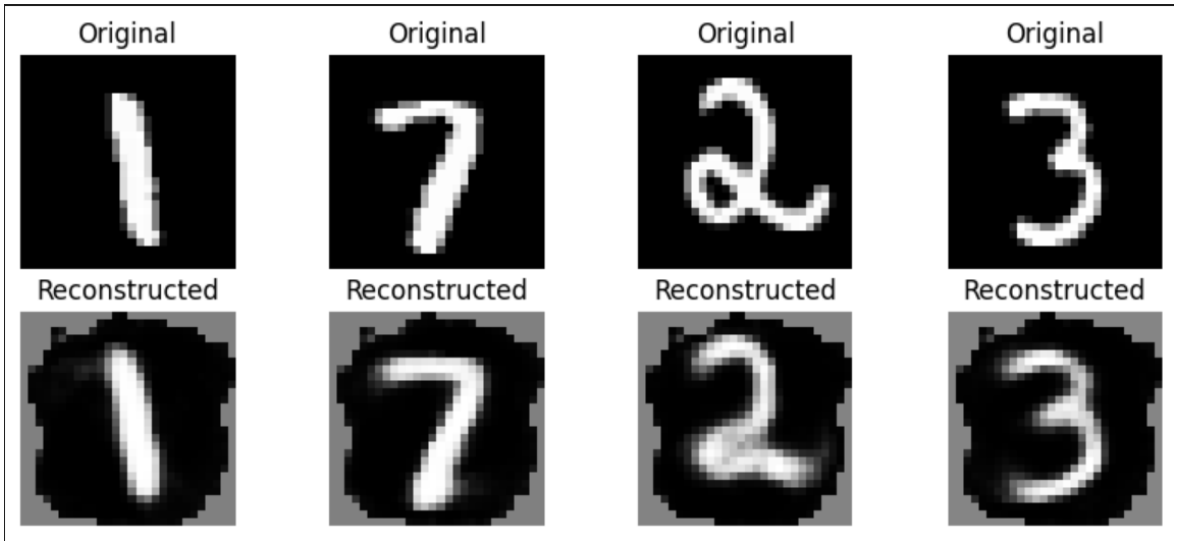
شکل ۳-۲ نشان دهنده عکس بدست آمده از محاسبه وزن پیکسل های عکس ها است .سپس طبق معادله داده شده برای Clustering-weighted MSE Loss تابع CustomWeightedMSELoss را تعریف کردیم.

$$L_{cmse} = \frac{\sum_{i=1}^{n} w_i(y_i - \hat{y}_i)^2}{n} \qquad L = L_{cmse} + \beta \dot{L}_r$$



شکل ۳-۲. Map بدست آمده از وزن های محاسبه شده از ۱۰۰۰ عکس

در شکل ۳-۳ عکس های اصلی و بازسازی شده توسط autoencoder آورده شده اند. همانطور که مشخص است بیتشر به بازسازی پیکسل هایی که در مرکز هستند، توجه شده است.



شکل ۳-۳. مقایسه عکس های تولید شده توسط autoencoder و عکس های اصلی

.۳

در این بخش پس از آموزش اتوانکودر، از لایه نهان حاصل از اتوانکودر به عنوان ورودی برای الکوریتم K-Means و برای انجام تسک clustering استفاده کردیم. با این کار تعداد فیچر های داده های ورودی

K-means را از ۷۸۴ به ۱۰ کاهش می دهیم. این کار عملکرد K-means را بهبود می بخشد زیرا در بعد های پایین تر این الگوریتم بهتر عمل میکند.

.۴

جدول ۳-۱ مقادیر بدست آمده از مدل ما را با مقادیر گزارش شده در مقاله را نشان می دهد. همان طور که در مقاله هم ذکر شده استفاده از این روش وزن دهی به فیچر ها باعث بهبود عملکرد دسته بندی در الگوریتم k-means شده است.

جدول ۳-۱. مقایسه ARI در روش ها مختلف

	K-Mesn	DAC
ARI from Our Impplemetation	0.3831	0.6194
ARI from Paper	0.3477	0.6624

پاسخ ۴ – شبکه Multi-Layer Perceptron

۴-۱. آشنایی و کار با مجموعه دادگان (پیش‌پردازش)

الف)

```
transform = transforms.Compose([transforms.ToTensor(), transforms.Lambda(lambda x: (x - x.min()) /
(x.max() - x.min()))])

trainset = datasets.MNIST('./data', download=True, train=True, transform=transform)
testset = datasets.MNIST('./data', download=True, train=False, transform=transform)
```

شکل ۴-۱. فراخوانی مجموعه دادگان MNIST

در اینجا از transform برای اعمال فیلتر min-max که در بخش‌های جلوتر نیاز به انجام آن داریم، در هنگام لود کردن داده استفاده شده است. می‌توان این کار را در اینجا انجام نداده و به صورت دستی جلوتر این بخش پردازش روی داده‌ها انجام شود.

```
train_shape = trainset.data.shape
print(f"Train data size: {train_shape[0]}")
print(f"Train data shape: {train_shape[1:]}")
```

```
Train data size: 60000
Train data shape: torch.Size([28, 28])
```

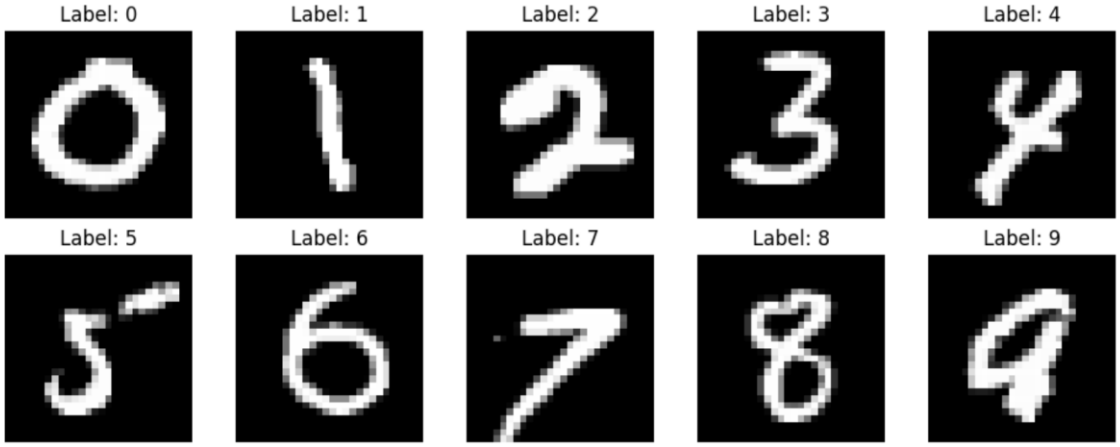
```
test_shape = testset.data.shape
print(f"Test data size: {test_shape[0]}")
print(f"Test data shape: {test_shape[1:]}")
```

```
Test data size: 10000
Test data shape: torch.Size([28, 28])
```

شکل ۴-۲. ابعاد و سایز مجموعه دادگان

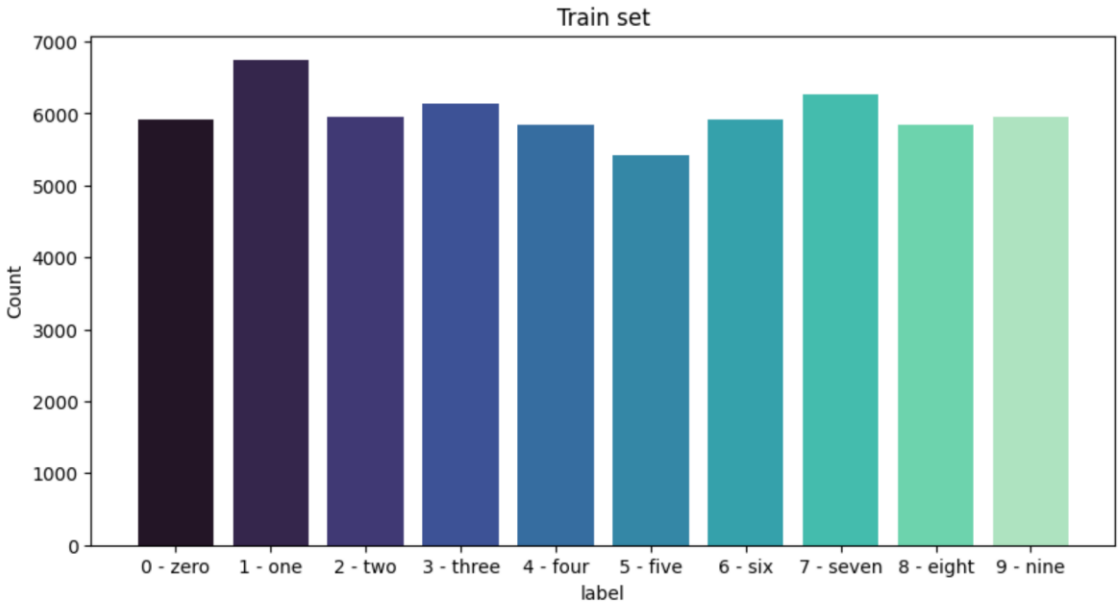
همانطور که در شکل ۴-۲ دیده می‌شود، مجموعه آموزش ما شامل ۶۰۰۰۰ سمپل و مجموعه ارزیابی ما شامل ۱۰۰۰۰ سمپل است. همچنین هر سمپل یک ماتریس با ابعاد ۲۸ در ۲۸ است.

ب)



شکل ۴-۳. نمایش یک سمپل از هر کلاس

ج)



شکل ۴-۴. رسم نمودار هیستوگرام تعداد نمونه‌های هر کلاس

د)

این بخش در هنگام لود کردن دیتاست انجام شده است. در صورت لزوم با استفاده از قطعه کد زیر می‌تواند این بخش را پس از لود کردن دیتا به صورت دستی نیز انجام داد.

```
# trainset.data = trainset.data.type(torch.float32)
# trainset.data = (trainset.data - trainset.data.min()) / (trainset.data.max() - trainset.data.min())
```

```
# testset.data = testset.data.type(torch.float32)
# testset.data = (testset.data - testset.data.min()) / (testset.data.max() - testset.data.min())
```

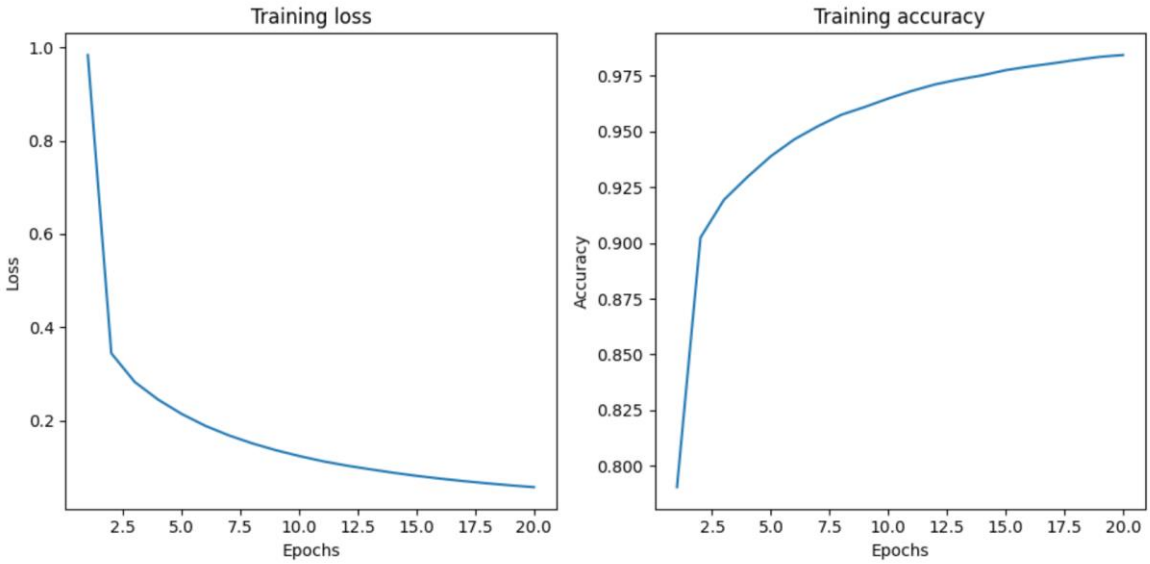
شکل ۴-۵. Min-max normalization

۴-۲. شبکه Teacher

```
class Classifier(nn.Module):

    def __init__(self, input_size, hidden1_size, hidden2_size, output_size):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear1 = nn.Linear(input_size, hidden1_size)
        self.act_fn1 = nn.ReLU()
        self.linear2 = nn.Linear(hidden1_size, hidden2_size)
        self.act_fn2 = nn.ReLU()
        self.linear3 = nn.Linear(hidden2_size, output_size)

    def forward(self, x):
        x = self.flatten(x)
        x = self.linear1(x)
        x = self.act_fn1(x)
        x = self.linear2(x)
        x = self.act_fn2(x)
        x = self.linear3(x)
        return x
```

```
teacher_model = Classifier(28*28, 1024, 512, 10)
teacher_model.to(device)
```

شکل ۴-۶. تعریف شبکه Teacher



شکل ۴-۷. نمودار دقت و خطا در حین آموزش شبکه Teacher

برای یافتن کلاس پیش‌بینی‌شده توسط شبکه از روی logitهای خروجی داده شده توسط شبکه، صرفا کافی است که اندیس نورون خروجی حامل بیشترین مقدار را پیدا کرده و به عنوان کلاس خروجی در نظر بگیریم. یعنی صرفا کافی است که روی لایه آخر شبکه یک argmax اعمال کنیم. این کار در واقع در راستای استفاده ما از loss function و به نوعی انتخاب نورون با بیشترین احتمال است.

در پایان دقت شبکه روی داده تست در حدود ۹۸ درصد بود. همچنین از ۱۰۰۰۰ نمونه تست، نتیجه پیش‌بینی شده توسط مدل تنها در ۲۳۳ مورد اشتباه بود.

```
teacher_num_preds_eval, teacher_true_preds_eval = eval_model(teacher_model, testloader)

print(f"Accuracy: {(teacher_true_preds_eval/teacher_num_preds_eval)*100:.2f}%")
print(f"Number of correct predictions: {teacher_true_preds_eval}/{teacher_num_preds_eval}")
print(f"Number of misclassifications: {teacher_num_preds_eval - teacher_true_preds_eval}")
```

```
Accuracy: 97.67%
Number of correct predictions: 9767/10000
Number of misclassifications: 233
```

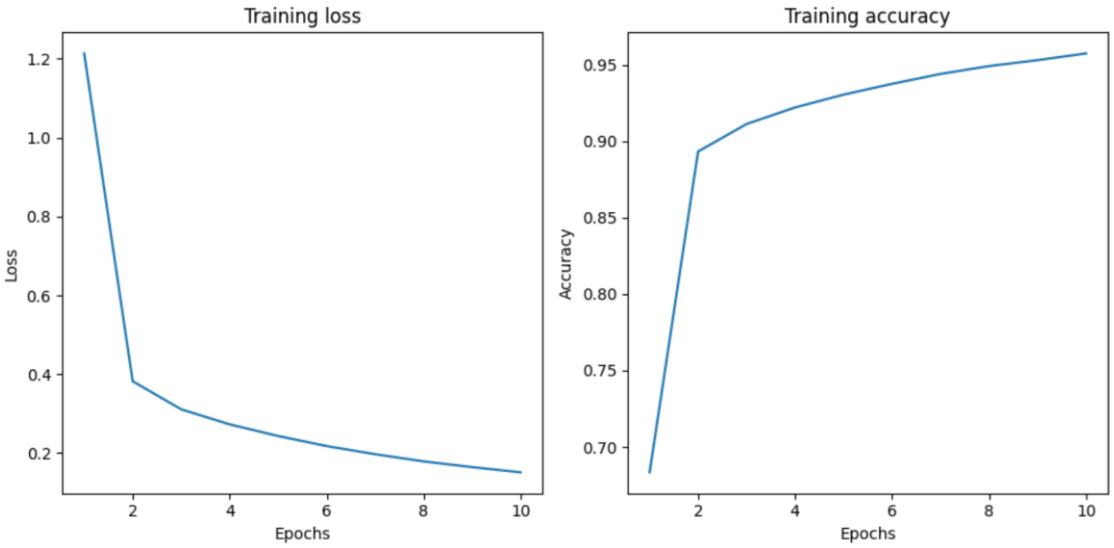
شکل ۴-۸. دقت و تعداد misclassificationهای مدل teacher

۴-۳. شبکه Student

```
student_model = Classifier(28*28, 128, 64, 10)
student_model.to(device)
```

```
Classifier(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear1): Linear(in_features=784, out_features=128, bias=True)
  (act_fn1): ReLU()
  (linear2): Linear(in_features=128, out_features=64, bias=True)
  (act_fn2): ReLU()
  (linear3): Linear(in_features=64, out_features=10, bias=True)
)
```

شکل ۴-۹. تعریف شبکه Student



شکل ۴-۱۰. نمودار دقت و خطای مدل Student در هنگام آموزش

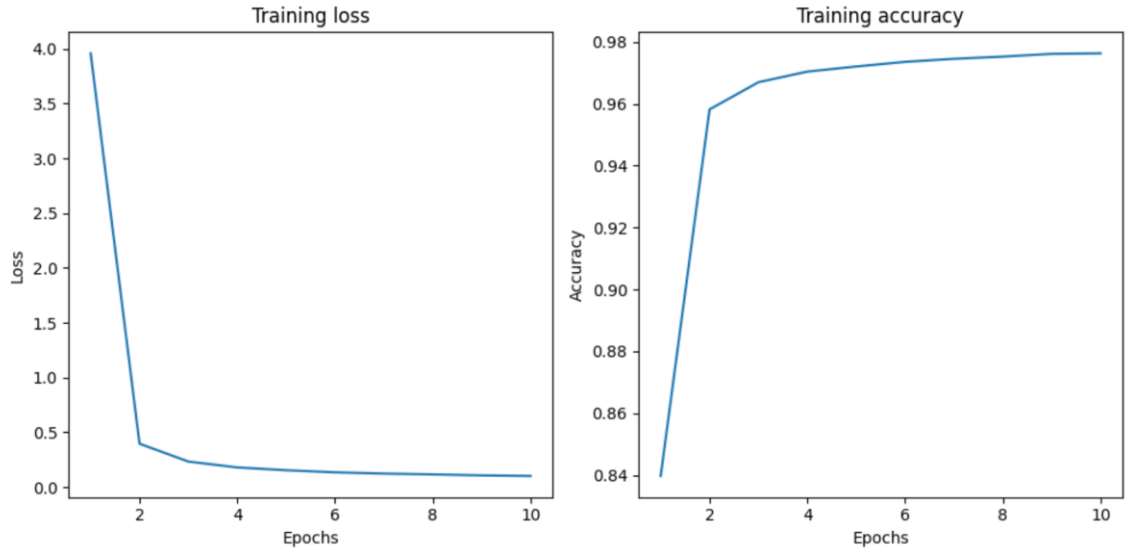
این مدل که پیچیدگی کمتری نسبت به مدل قبلی دارد، پس از آموزش توانست به دقت حدود ۹۵ درصد روی مجموعه داده تست برسد.

```
student1_num_preds_eval, student1_true_preds_eval = eval_model(student_model, testloader)

print(f"Accuracy: {(student1_true_preds_eval/student1_num_preds_eval)*100:.2f}%")
print(f"Number of correct predictions: {student1_true_preds_eval}/{student1_num_preds_eval}")
print(f"Number of misclassifications: {student1_num_preds_eval - student1_true_preds_eval}")
```

```
Accuracy: 95.63%
Number of correct predictions: 9563/10000
Number of misclassifications: 437
```

شکل ۴-۱۱. دقت و misclassification شبکه Student

۴-۴. Knowledge Distillation



شکل ۴-۱۲. نمودار دقت و خطا در حین آموزش

در پایان و پس از اتمام آموزش این شبکه، با ارزیابی آن روی مجموعه داده تست به دقت حدود ۹۷ درصد می‌رسیم که نسبت به مدل Student در بخش قبلی بهبود دارد. از آنجا که معماری مدل دقیقا مثل قبل است و پیچیدگی‌ای به آن افزوده نشده است، این بهبود قابل توجه است.

```
student2_num_preds_eval, student2_true_preds_eval = eval_model(student_model_with_logits, testloade
r)

print(f"Accuracy: {(student2_true_preds_eval/student2_num_preds_eval)*100:.2f}%")
print(f"Number of correct predictions: {student2_true_preds_eval}/{student2_num_preds_eval}")
print(f"Number of misclassifications: {student2_num_preds_eval - student2_true_preds_eval}")
```

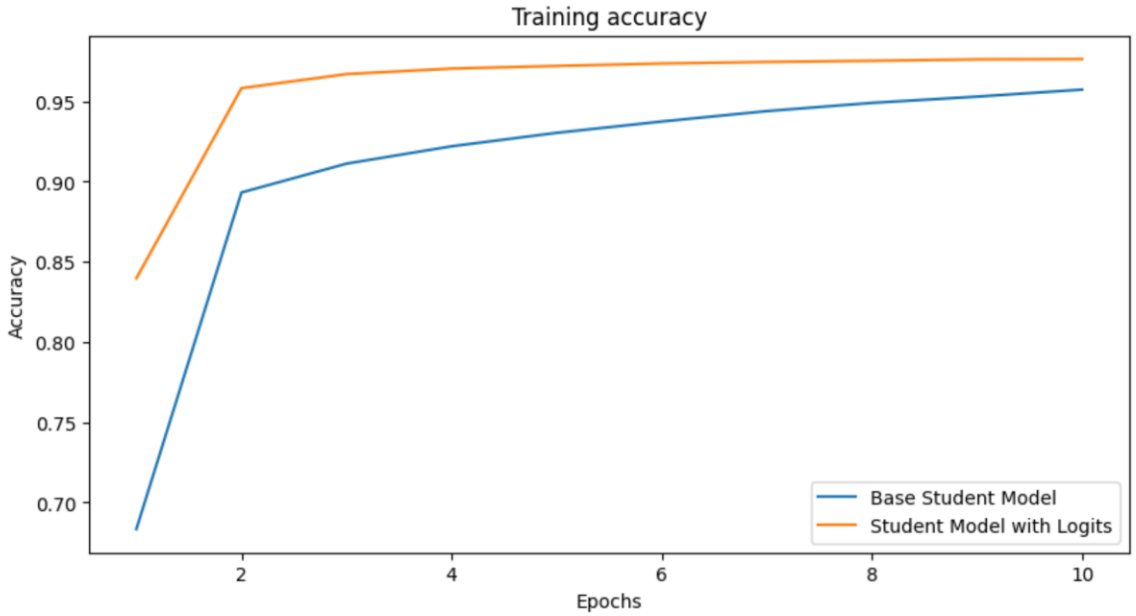
```
Accuracy: 97.19%
Number of correct predictions: 9719/10000
Number of misclassifications: 281
```

شکل ۴-۱۳. دقت و تعداد misclassificationها

الف) مقایسه تعداد پیش‌بینی‌های غلط

همانطور که در نتایج مشخص است، در حالت اول و با آموزش مدل روی داده‌های آموزش اصلی، ۴۳۷ پیش‌بینی اشتباه داشتیم که این عدد در حالت دوم با آموزش همان مدل روی logitها به ۲۸۱ پیش‌بینی اشتباه کاهش یافت که این بهبود نسبی حدود ۳,۷ درصدی را نشان می‌دهد و نتیجه قابل قبولی است.

ب) مقایسه دقت مدل‌ها حین آموزش



شکل ۴-۱۴. نمودار دقت دو مدل حین آموزش

همانطور که در نمودار هم دیده می‌شود، استفاده از knowledge Distillation منجر به همگرایی سریع‌تر و بهتر مدل می‌شود که این به دلیل استفاده مدل از دانش مدل بزرگ‌تر و انتقال آن به مدل با پیچیدگی کمتر است. این اثر همگرایی سریع‌تر به خصوص در epochهای ابتدایی قابل مشاهده است. همچنین این کار در کل می‌تواند به generalization بهتر مدل و بهتر عمل کردن آن به خصوص در نمونه‌های چالشی‌تر منجر شود.