به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

# درس شبکه‌های عصبی و یادگیری عمیق

# تمرین اول

| | |
|---|---|
| **نام و نام خانوادگی** | محمدرضا نعمتی |
| **شماره دانشجویی** | 810100226 |
| **نام و نام خانوادگی** | محمدامین یوسفی |
| **شماره دانشجویی** | 810100236 |
| **مهلت ارسال پاسخ** | **1403/08/15** |

# فهرست

# شکل‌ها و جدول‌ها

# پرسش 1. تحلیل و طراحی شبکه‌های عصبی چندلایه(MLP)

## ۱-۱. طراحی MLP

مدل ایجاد شده برای این بخش:

```python
class FashionMNISTNet(nn.Module):
    def __init__(self):
        super(FashionMNISTNet, self).__init__()
        self.fc1 = nn.Linear(28 * 28, 100)
        self.dropout = nn.Dropout(0.3)
        self.fc2 = nn.Linear(100, 10)

    def forward(self, x):
        x = x.view(-1, 28 * 28)
        x = torch.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

در این مدل، همه نکات گفته شده در صورت سوال ذکر شده است. این مدل را با استفاده از تابع هزینه CrossEntropy و بهینه ساز SGD و با مقادیر Hyperparameterهای زیر آموزش می‌دهیم:

جدول 1-1. مقادیر Hyperparameterها در بخش 1-1

| | |
|---|---|
| **Learning rate** | 0.01 |
| **Lambda** | 0.0001 |
| **Dropout rate** | 0.3 |
| **Epochs** | 40 |
| **Batch size** | 32 |

حال پس از آموزش مدل، به سراغ تحلیل نتایج آن می‌رویم. گزارش کلی طبقه‌بندی:

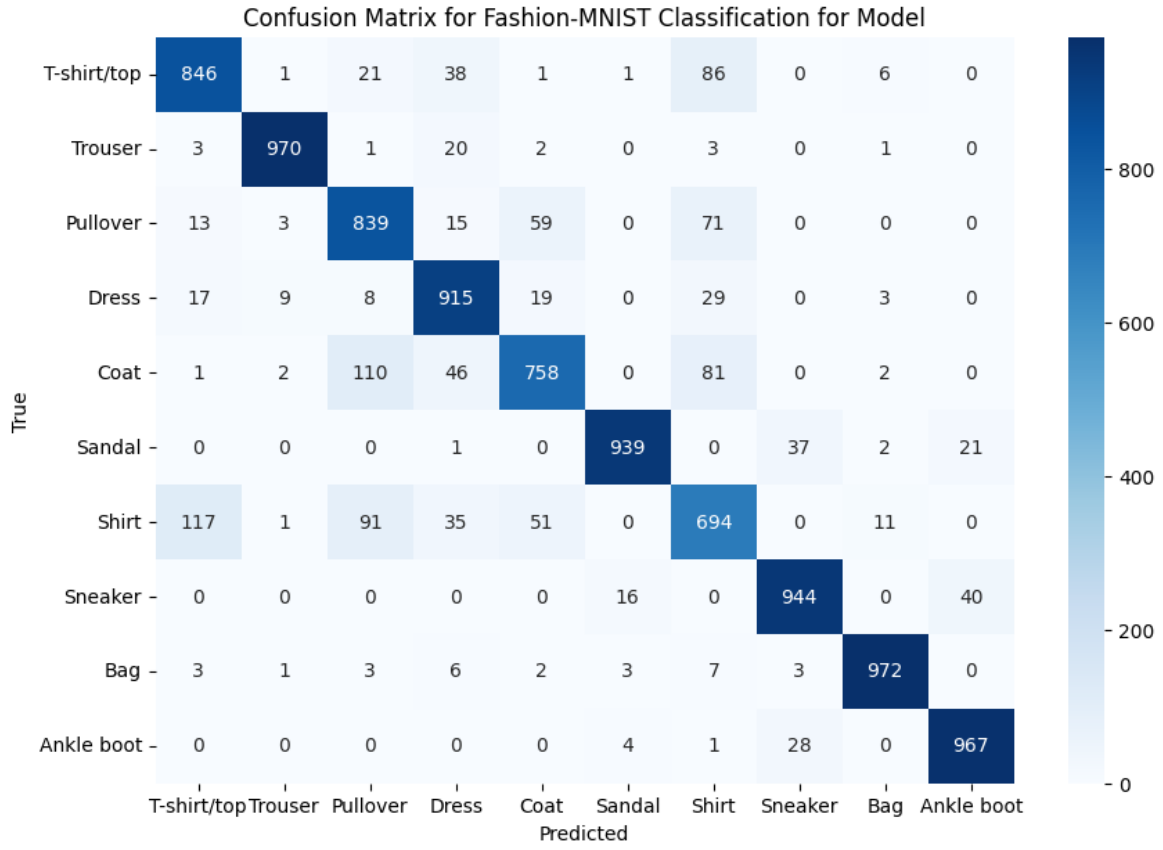
| | precision | recall | f1-score | support |
|---|---|---|---|---|
| T-shirt/top | 0.85 | 0.85 | 0.85 | 1000 |
| Trouser | 0.98 | 0.97 | 0.98 | 1000 |
| Pullover | 0.78 | 0.84 | 0.81 | 1000 |
| Dress | 0.85 | 0.92 | 0.88 | 1000 |
| Coat | 0.85 | 0.76 | 0.80 | 1000 |
| Sandal | 0.98 | 0.94 | 0.96 | 1000 |
| Shirt | 0.71 | 0.69 | 0.70 | 1000 |
| Sneaker | 0.93 | 0.94 | 0.94 | 1000 |
| Bag | 0.97 | 0.97 | 0.97 | 1000 |
| Ankle boot | 0.94 | 0.97 | 0.95 | 1000 |
| accuracy | | | 0.88 | 10000 |
| macro avg | 0.88 | 0.88 | 0.88 | 10000 |
| weighted avg | 0.88 | 0.88 | 0.88 | 10000 |

شکل 1- 1. گزارش کلی طبقه‌بندی

این نتایج نشان می‌دهند که مدل شما در طبقه‌بندی لباس‌ها عملکرد قابل قبولی دارد. مدل در کل ۸۸ درصد از نمونه‌ها را به درستی طبقه‌بندی کرده است که برای یک مدل یادگیری عمیق نتیجه‌ی بدی نیست، اما ایده‌آل هم نیست. میانگین دقت، بازیابی و F1-score در کل کلاس‌ها هم حدود ۰.۸۸ است، که یعنی مدل تقریباً برای همه کلاس‌ها به یک اندازه خوب یا ضعیف عمل کرده.

مدل در تشخیص بعضی لباس‌ها، مثل Trouser، Sandal، Sneaker و Bag، خیلی خوب عمل کرده و دقت بالای ۰.۹۴ دارد. یعنی برای این دسته‌ها اشتباه کمی داشته و به خوبی آنها را شناسایی کرده.

اما برای بعضی از کلاس‌ها، مثل Shirt و Pullover، مدل عملکرد ضعیف‌تری دارد. F1-score پیراهن فقط ۰.۷۰ است و برای ژاکت هم ۰.۸۱ است. به احتمال زیاد دلیلش این است که این دو نوع لباس از لحاظ ظاهری شباهت بیشتری به هم دارند و باعث سردرگمی مدل شده‌اند.




شکل 1- 2. ماتریس آشفتگی طبقه‌بندی

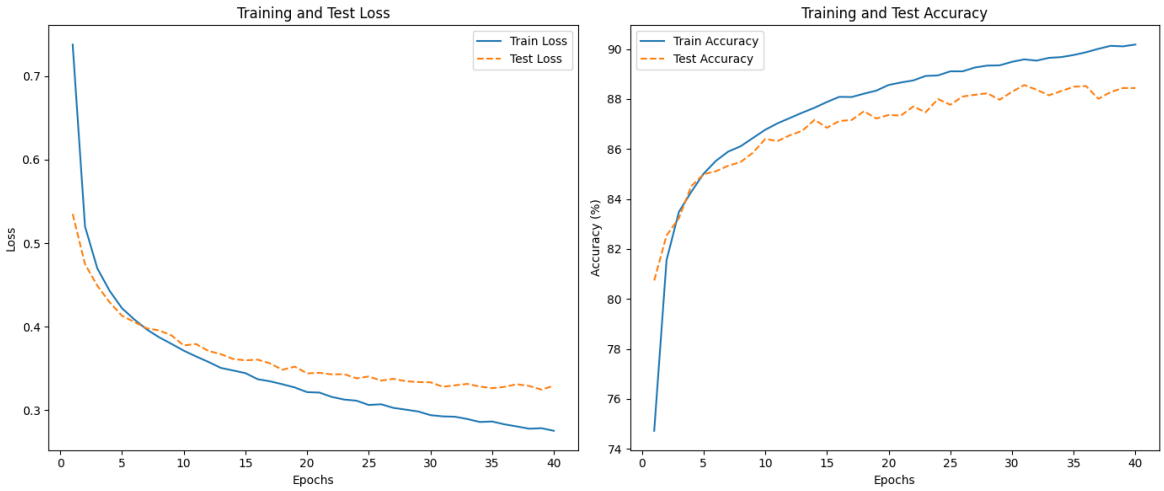
تصویر 1-2، ماتریس آشفتگی نتیجه طبقه‌بندی را نشان می‌دهد. مطابق این ماتریس، دو کلاسی که به طور کلی بیش از سایر کلاس‌ها با هم اشتباه گرفته می‌شوند، Shirt و T-shirt/top و در رتبه بعد از آنها، Coat و Pullover هستند.

در تصویر 1-3 ، برای هر کلاس، کلاسی که بیشتر با آن اشتباه گرفته می‌شود را مشاهده می‌کنید:

```
Class 'T-shirt/top' is most confused with class 'Shirt'.
Class 'Trouser' is most confused with class 'Dress'.
Class 'Pullover' is most confused with class 'Shirt'.
Class 'Dress' is most confused with class 'Shirt'.
Class 'Coat' is most confused with class 'Pullover'.
Class 'Sandal' is most confused with class 'Sneaker'.
Class 'Shirt' is most confused with class 'T-shirt/top'.
Class 'Sneaker' is most confused with class 'Ankle boot'.
Class 'Bag' is most confused with class 'Shirt'.
Class 'Ankle boot' is most confused with class 'Sneaker'.
```

شکل 1- 3. کلاس‌هایی که بیش از کلاس‌های دیگر، با هم اشتباه گرفته می‌شوند

تصویر زیر، سرعت و نحوه همگرایی هزینه و دقت طبقه‌بندی را نمایش می‌دهد. همانطور که مشاهده می‌کنید، مدل پس از epoch شماره 30، شروع به overfit می‌کند.




شکل 1- 4. سرعت و نحوه همگرایی هزینه و دقت طبقه‌بندی

**پرسش:** چگونه افزایش پیچیدگی مدل با استفاده از تعداد بیشتر لایه‌های مخفی یا نورون‌ها می‌تواند بهبود عملکرد را در پی داشته باشد؟

افزایش پیچیدگی مدل با افزودن لایه‌های مخفی بیشتر یا افزایش تعداد نورون‌ها در هر لایه، ظرفیت مدل را برای یادگیری و تشخیص الگوهای پیچیده‌تر بهبود می‌بخشد. این امر به مدل اجازه می‌دهد تا روابط غیرخطی و وابستگی‌های پیچیده بین ورودی‌ها و خروجی‌ها را بهتر شناسایی کند.

با این حال، افزایش بیش از حد پیچیدگی مدل می‌تواند منجر به بیش‌برازش شود، به این معنا که مدل به جای یادگیری الگوهای کلی، جزئیات و نویز موجود در داده‌های آموزشی را حفظ می‌کند. این امر باعث می‌شود که مدل روی داده‌های آموزشی عملکرد بسیار خوبی داشته باشد اما روی داده‌های جدید و دیده‌نشده ضعیف عمل کند. برای مقابله با این مشکل، تکنیک‌های منظم‌سازی مانند Dropout, L2 Regularization و Early Stopping به کار می‌روند. بنابراین، افزایش تعداد لایه‌ها و نورون‌ها باید به دقت و با استفاده از تکنیک‌های منظم‌سازی همراه باشد تا مدل بتواند تعادلی بین دقت و تعمیم‌پذیری ایجاد کند و بهترین عملکرد ممکن را در داده‌های واقعی ارائه دهد

**پرسش:** چه معیارهایی برای انتخاب بهترین پیکربندی وجود دارد؟

برای انتخاب بهترین پیکربندی شبکه عصبی، معیارهای اصلی شامل دقت مدل روی داده‌های اعتبارسنجی و دیگر معیارهای ارزیابی مانند Precision، Recall، F1 score، و AUC-ROC می‌شوند. این معیارها بر اساس نوع مسئله انتخاب می‌شوند و به سنجش عملکرد مدل کمک می‌کنند. جلوگیری از بیش‌برازش با استفاده از تکنیک‌هایی مثل Dropout و Regularization، همچنین استفاده از Early Stopping به بهبود تعمیم‌دهی مدل کمک می‌کند. زمان محاسباتی و پیچیدگی مدل نیز از دیگر معیارهای مهم هستند، زیرا مدل‌های پیچیده‌تر زمان بیشتری برای آموزش نیاز دارند و در کاربردهایی که پاسخ سریع ضروری است، ممکن است مدل‌های کم عمق و بهینه‌تر انتخاب بهتری باشند.

انتخاب معماری مناسب نیز بر اساس نوع داده‌ها و نیازهای خاص مسئله اهمیت دارد. برای مثال، برای داده‌های پیچیده مثل تصاویر و زبان، مدل‌های عمیق‌تر مانند CNN و RNN مناسب‌ترند، اما در مسائل ساده‌تر، مدل‌های کم‌عمق کفایت می‌کنند. انتخاب تابع فعال‌ساز مناسب نیز بر اساس پیچیدگی داده‌ها اهمیت دارد تا مدل بتواند ویژگی‌های مورد نیاز را به خوبی یاد بگیرد. محدودیت‌های سخت‌افزاری و نیاز نهایی پروژه نیز در انتخاب مدل مؤثرند، زیرا باید مدلی انتخاب شود که هم عملکرد خوبی داشته باشد و هم از نظر زمان اجرا و منابع محاسباتی کارآمد باشد.

## ۱-۲. آموزش دو مدل متفاوت

دو مدل متفاوت مطابق خواسته صورت سوال ایجاد می‌کنیم. مدل شماره 1 دارای یک لایه مخفی با 128 نود و بدون منظم کننده و Dropout می‌باشد اما در طرف مقابل، مدل دوم دارای یک لایه مخفی با 48 نود، منظم کننده با مقدار lambda برابر 0.0001 و Dropout با نرخ 0.2 می‌باشد. در قطعه کد زیر، می‌توانید مدل‌ها را مشاهده نمایید.

```
class MLPWithoutDropout(nn.Module):
    def __init__(self):
        super(MLPWithoutDropout, self).__init__()
        self.fc1 = nn.Linear(28*28, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(-1, 28*28)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

class MLPWithDropout(nn.Module):
    def __init__(self):
```

```
        super(MLPWithDropout, self).__init__()
        self.fc1 = nn.Linear(28*28, 48)
        self.dropout = nn.Dropout(0.2)
        self.fc2 = nn.Linear(48, 10)

    def forward(self, x):
        x = x.view(-1, 28*28)
        x = torch.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

Hyperparameterهای آموزش این مدل‌ها را می‌توانید در جدول زیر مشاهده نمایید.

جدول 1-2. مقادیر Hyperparameterها در بخش 1-2

| | |
|---|---|
| **Learning rate** | 0.01 |
| **Lambda** | 0.0001 |
| **Dropout rate** | 0.2 |
| **Epochs** | 40 |
| **Batch size** | 32 |

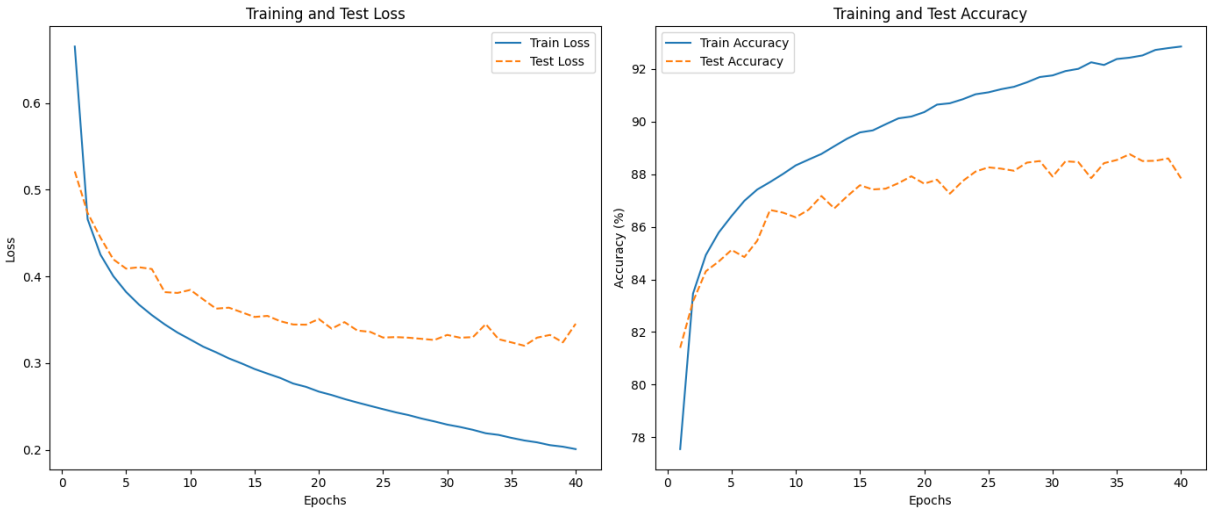
| | precision | recall | f1-score | support |
|---|---|---|---|---|
| T-shirt/top | 0.81 | 0.87 | 0.84 | 1000 |
| Trouser | 0.99 | 0.95 | 0.97 | 1000 |
| Pullover | 0.68 | 0.90 | 0.78 | 1000 |
| Dress | 0.86 | 0.91 | 0.89 | 1000 |
| Coat | 0.86 | 0.70 | 0.77 | 1000 |
| Sandal | 0.97 | 0.94 | 0.96 | 1000 |
| Shirt | 0.79 | 0.62 | 0.70 | 1000 |
| Sneaker | 0.93 | 0.96 | 0.95 | 1000 |
| Bag | 0.97 | 0.96 | 0.96 | 1000 |
| Ankle boot | 0.95 | 0.96 | 0.96 | 1000 |
| accuracy | | | 0.88 | 10000 |
| macro avg | 0.88 | 0.88 | 0.88 | 10000 |
| weighted avg | 0.88 | 0.88 | 0.88 | 10000 |

شکل 1-5. گزارش کلی طبقه‌بندی با مدل اول

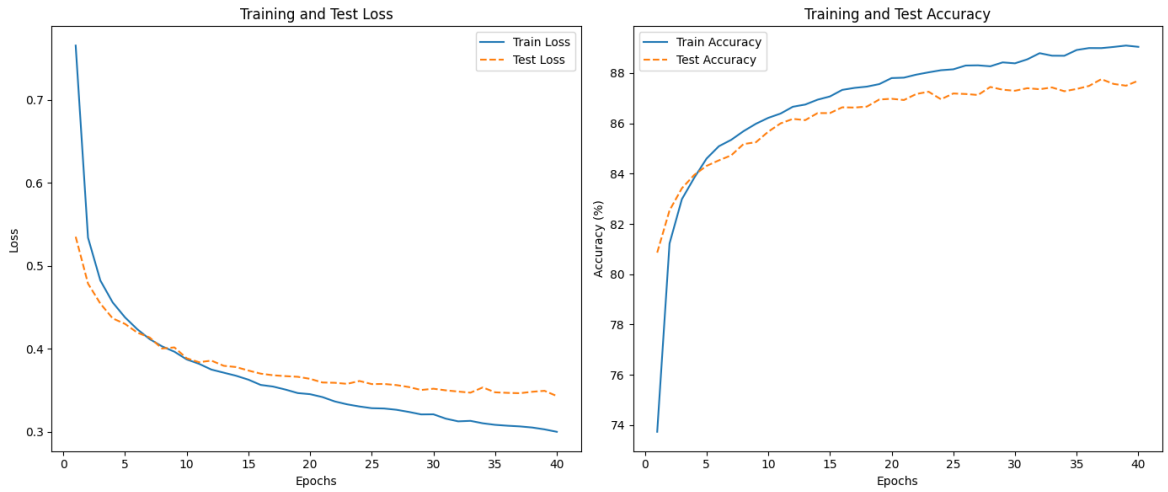
| | precision | recall | f1-score | support |
|---|---|---|---|---|
| T-shirt/top | 0.80 | 0.86 | 0.83 | 1000 |
| Trouser | 0.99 | 0.96 | 0.98 | 1000 |
| Pullover | 0.82 | 0.77 | 0.79 | 1000 |
| Dress | 0.88 | 0.88 | 0.88 | 1000 |
| Coat | 0.78 | 0.84 | 0.81 | 1000 |
| Sandal | 0.96 | 0.94 | 0.95 | 1000 |
| Shirt | 0.71 | 0.63 | 0.67 | 1000 |
| Sneaker | 0.91 | 0.97 | 0.94 | 1000 |
| Bag | 0.95 | 0.97 | 0.96 | 1000 |
| Ankle boot | 0.97 | 0.94 | 0.95 | 1000 |
| accuracy | | | 0.88 | 10000 |
| macro avg | 0.88 | 0.88 | 0.88 | 10000 |
| weighted avg | 0.88 | 0.88 | 0.88 | 10000 |

شکل 1-6. گزارش کلی طبقه‌بندی با مدل دوم

تصویر 1-5 و 1-6 نشان دهنده نتیجه طبقه‌بندی داده‌ها با استفاده از مدل اول و دوم می‌باشند. همانطور که مشاهده می‌کنید، دقت طبقه‌بندی این دو مدل تفاوت آشکاری با یکدیگر ندارد بلکه تفاوت در جای دیگری می‌باشد. تصویر 1-7 نشان دهنده سرعت و نحوه همگرایی میزان هزینه و دقت داده‌های آموزش و تست با استفاده از مدل اول می‌باشد. همانطور که مشاهده می‌کنید، این مدل دچار Overfitting شده است اما تصویر 1-8 که متعلق به مدل دوم است، به دلیل داشتن منظم‌کننده و Dropout، درگیر Overfitting نشده است.

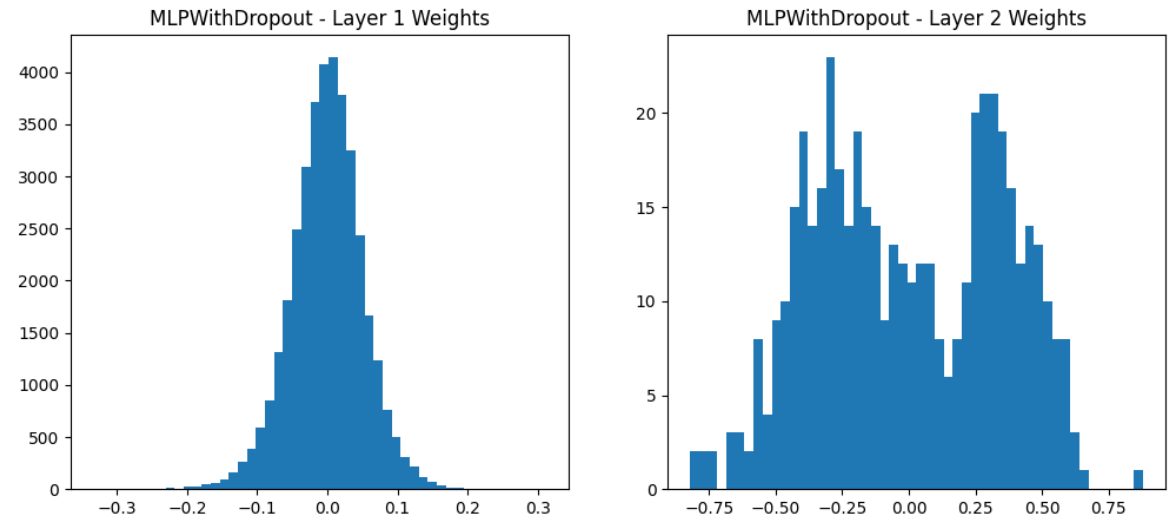



شکل 1-7. سرعت و نحوه همگرایی هزینه و دقت طبقه‌بندی با مدل اول

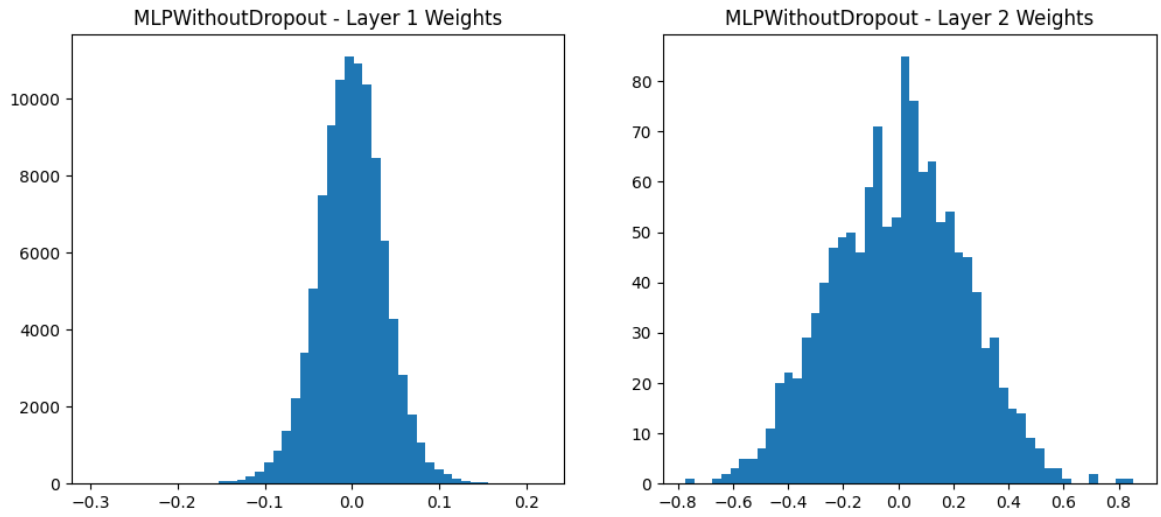



شکل 1-8. سرعت و نحوه همگرایی هزینه و دقت طبقه‌بندی با مدل دوم

تفاوت کلیدی بین این دو شبکه در استفاده از لایه Dropout و منظم‌سازی L2 در شبکه دوم است که در شبکه اول وجود ندارد. به‌کارگیری L2 Regularization در شبکه دوم باعث می‌شود نورون‌های لایه اول بیشتر روی وزن‌های کوچکتر و نزدیک به صفر تمرکز کنند. همان‌طور که در تصاویر 1-9 و 1-10 مشاهده می‌شود، در شبکه دوم وزن‌های این لایه بیشتر حول مقادیر کوچکتر توزیع شده‌اند، در حالی که در شبکه اول این توزیع گسترده‌تر و شامل وزن‌های بزرگتر است. همچنین، وجود لایه Dropout در شبکه دوم کمک می‌کند تا شبکه برای پیش‌بینی به وزن‌ها و نورون‌های خاص وابسته نباشد، بلکه تمام نورون‌ها به‌طور یکنواخت آموزش دیده و در فرایند تصمیم‌گیری نهایی نقش داشته باشند.




شکل 1-9. اوزان مدل اول




شکل 1-10. اوزان مدل دوم

بهینه‌ساز Adam و SGD هر دو الگوریتم‌هایی برای بهینه‌سازی شبکه‌های عصبی هستند، اما تفاوت‌های کلیدی دارند. در حالی که SGD با استفاده از یک نرخ یادگیری ثابت و محاسبه گرادیان در هر مرحله، پارامترها را به سمت مینیمم محلی حرکت می‌دهد، Adam با تطبیق نرخ یادگیری برای هر پارامتر بر اساس میانگین وزنی گرادیان‌ها و مربع آنها عمل می‌کند. این روش به Adam اجازه می‌دهد تا در تنظیم نرخ یادگیری در هر مرحله هوشمندانه‌تر و با سرعت بیشتری به همگرایی برسد. به‌ویژه در مسائل پیچیده‌تر یا داده‌های noisy، Adam با ترکیب مزایای SGD و RMSprop پایدارتر و کارآمدتر عمل می‌کند، هرچند که ممکن است در برخی موارد دقت نهایی پایین‌تری نسبت به SGD داشته باشد.

## ۱-۳. الگوریتم بازگشت به عقب

- **بهینه‌ساز Adam**

بهینه‌ساز Adam یک الگوریتم پیشرفته برای آموزش شبکه‌های عصبی است که ترکیبی از دو روش Momentum و RMSProp می‌باشد. این الگوریتم از میانگین نمایی گرادیان‌ها و مربعات آن‌ها استفاده می‌کند تا نرخ یادگیری را به صورت تطبیقی برای هر پارامتر تنظیم کند، به این ترتیب سرعت و پایداری همگرایی افزایش می‌یابد. Adam همچنین با اعمال **تصحیح بایاس**، تخمین‌های اولیه را بهبود می‌بخشد.

مقادیر اولیه پارامترها:

- نرخ یادگیری: 0.001
- بتا1: 0.9

- بتا2: 0.999
- اپسیلون: 0.000000001

برخی مراحل اصلی:

1. مقداردهی اولیه:

$$m_0 = 0$$
$$v_0 = 0$$

2. به روزرسانی میانگین و واریانس:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

3. تصحیح Bias:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

4. به روزرسانی پارامترها:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

- **بهینه‌ساز Nadam**

بهینه‌ساز Nadam نسخه بهبودیافته‌ای از Adam است که از تکنیک Nesterov Accelerated Gradient استفاده می‌کند. Nadam، مانند Adam، از میانگین متحرک گرادیان‌ها و مربعات آن‌ها برای تنظیم نرخ یادگیری به صورت تطبیقی بهره می‌برد، اما با افزودن Nesterov Accelerated Gradient، به‌روزرسانی‌ها را با استفاده از اطلاعات آینده‌ای که به دست می‌آورد، بهینه‌تر می‌کند. به طور خلاصه، باعث می‌شود مدل سریع‌تر به همگرایی برسد و عملکرد بهتری در تنظیمات مختلف داشته باشد. پارامترهای پیش‌فرض آن مشابه Adam و به دلیل اصلاحات در به‌روزرسانی پارامترها، به ویژه در معماری‌های پیچیده مانند شبکه‌های عصبی عمیق، عملکرد بهتری نسبت به Adam نشان می‌دهد.

برخی مراحل اصلی:

1. مقداردهی اولیه:

$m_0 = 0$
$v_0 = 0$

2. به روزرسانی میانگین و واریانس:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

3. تصحیح Bias:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

4. به روزرسانی پارامترها:

$$\text{Nesterov term} = (1 - \beta_1) g_t + \beta_1 \hat{m}_{t-1}$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} (\beta_1 \hat{m}_t + \text{Nesterov term})$$

- **بهینه‌ساز RMSprop**

بهینه‌ساز RMSprop یک الگوریتم بهینه‌سازی است که به منظور حل مشکل نوسانات شدید در نرخ یادگیری در طول آموزش شبکه‌های عصبی طراحی شده است. این روش برای تنظیم نرخ یادگیری به صورت تطبیقی عمل می‌کند و به خصوص در شبکه‌های عمیق و مسائل سری‌های زمانی مانند RNNها عملکرد خوبی دارد. این بهینه‌ساز، از میانگین نمایی مربعات گرادیان‌های گذشته استفاده می‌کند تا نرخ یادگیری را برای هر پارامتر بهینه‌سازی کند. در زمینه سرعت همگرایی می‌توان گفت Nadam نسبت به Adam سرعت همگرایی بالاتری دارد (به دلیل پیش‌بینی گرادیان) و هردو این بهینه‌سازها نسبت به RMSprop در مدل‌های بزرگ سرعت همگرایی بالاتری دارند. اگرچه سرعت همگرایی RMSprop نسبت به SGD ساده بیشتر است. در زمینه دقت کلی مدل، Nadam در مدل‌های بزرگ‌تر دقت نسبتا بالاتری نسبت به Adam ارائه می‌دهد. بهینه‌ساز RMSprop در مدل‌های RNN عملکرد مناسبی نشان می‌دهد، ولی در حالت‌های کلی ممکن است به خوبی Adam و Nadam عمل نکند.

مراحل این الگوریتم به این شکل است:

1. مقداردهی اولیه:

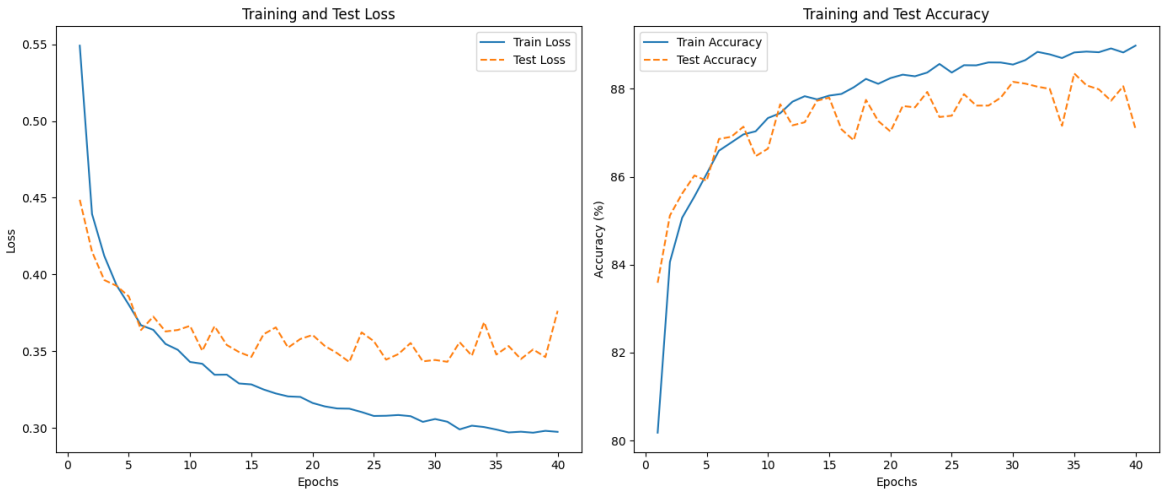
$$E[g^2]_0 = 0$$

2. به روزرسانی میانگین متحرک مربع گرادیان‌ها:

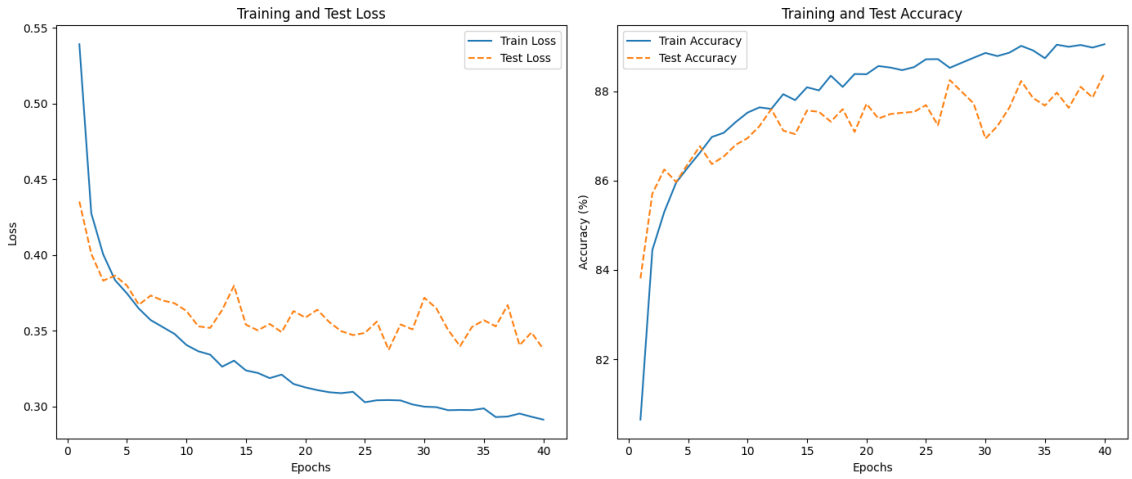
$$E[g^2]_t = \beta E[g^2]_{t-1} + (1-\beta) g_t^2$$

3. به روزرسانی پارامترها:

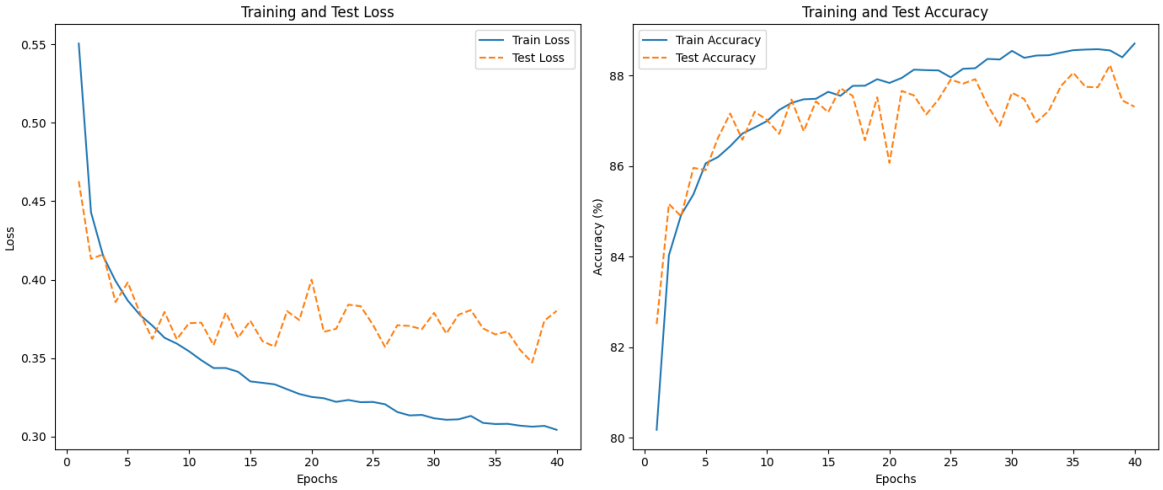
$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{E[g^2]_t + \epsilon}} g_t$$



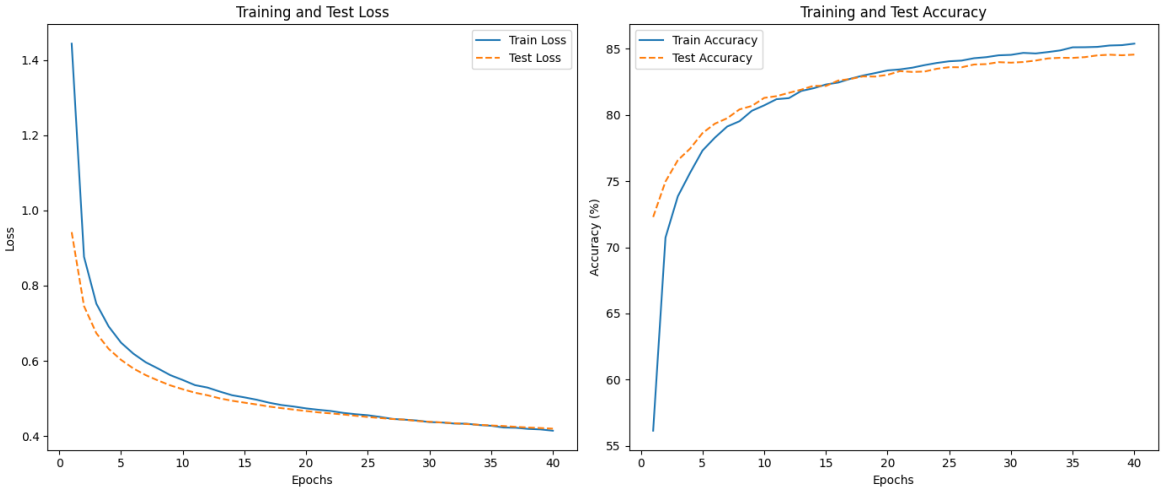

شکل 1-11. سرعت و نحوه همگرایی هزینه و دقت طبقه‌بندی با بهینه‌ساز Adam



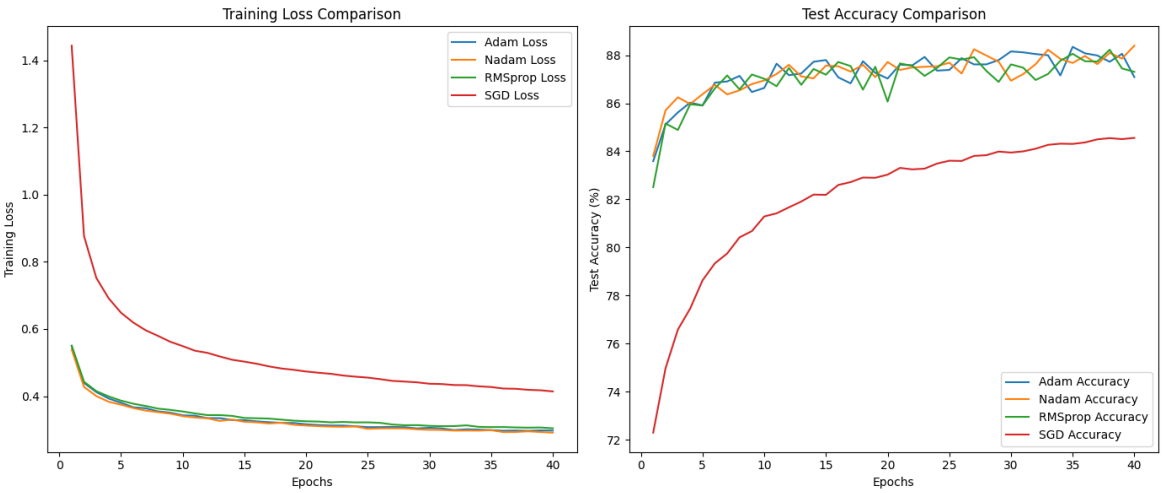

شکل 1-12. سرعت و نحوه همگرایی هزینه و دقت طبقه‌بندی با بهینه‌ساز Nadam




شکل 1-12. سرعت و نحوه همگرایی هزینه و دقت طبقه‌بندی با بهینه‌ساز RMSprop




شکل 1-13. سرعت و نحوه همگرایی هزینه و دقت طبقه‌بندی با بهینه‌ساز SGD

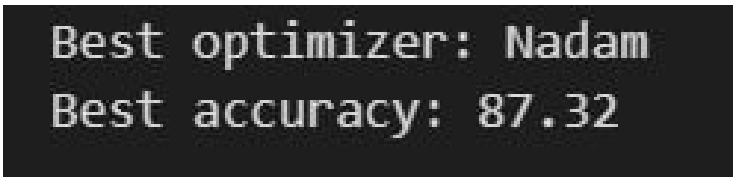


شکل 1-14. مقایسه سرعت و نحوه همگرایی هزینه و دقت طبقه‌بندی با هر 4 بهینه‌ساز

مدل استفاده شده در این بخش، همان مدل پیاده سازی شده در بخش 1-1 است. تنها تفاوت بخش آموزش این بخش با بخش 1-1، میزان نرخ یادگیری است که در اینجا برابر 0.001 می‌باشد.

به طور کلی، همانطور که در تصویر 1-14 مشاهده می‌کنید، دقت و سرعت همگرایی Nadam از باقی بهینه‌ها بیشتر است (با اختلاف بسیار کم نسبت به Adam) که این توضیحات داده شده را تصدیق می‌نماید.

جستجوی بیزی برای یافتن بهترین هایپرپارامترها یک روش بهینه‌سازی آماری است که به جای جستجوی تصادفی یا شبکه‌ای، با مدل‌سازی احتمالاتی از نتایج هایپرپارامترها، بهینه‌ترین مقادیر را پیدا می‌کند. این روش با تعریف تابع هدف و استفاده از تابع کسب (Acquisition Function)، به‌طور هوشمندانه ترکیبات جدیدی از هایپرپارامترها را آزمایش می‌کند و با هر تکرار، مدل احتمالاتی را به‌روزرسانی می‌کند تا به بهترین نتیجه برسد. جستجوی بیزی به دلیل کاهش تعداد آزمایش‌ها و بهره‌گیری از نتایج قبلی، برای بهینه‌سازی هایپرپارامترهای مدل‌های پیچیده و ارزیابی‌های زمان‌بر بسیار مفید است.



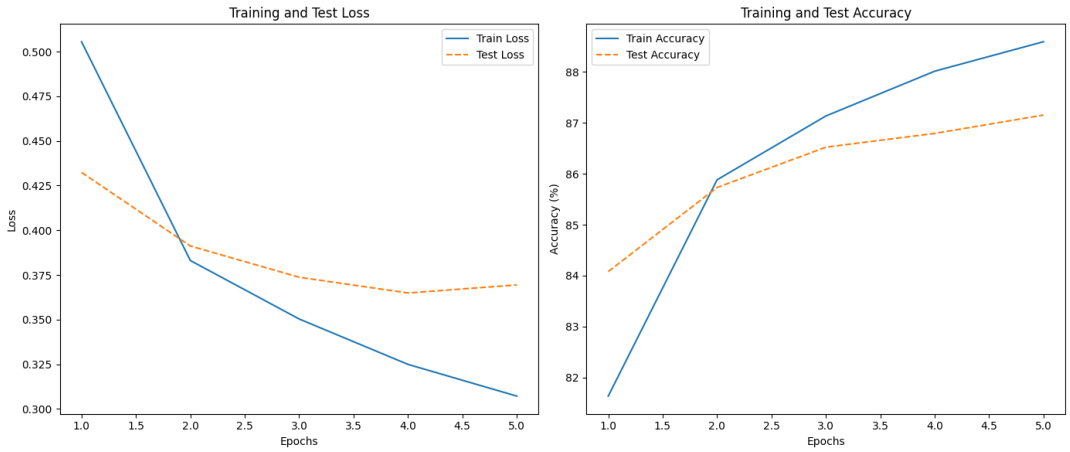

شکل 1-15. جستجوی بیزی روی بهینه‌ساز

## ۱-۴. بررسی هایپرپارامترهای مختلف

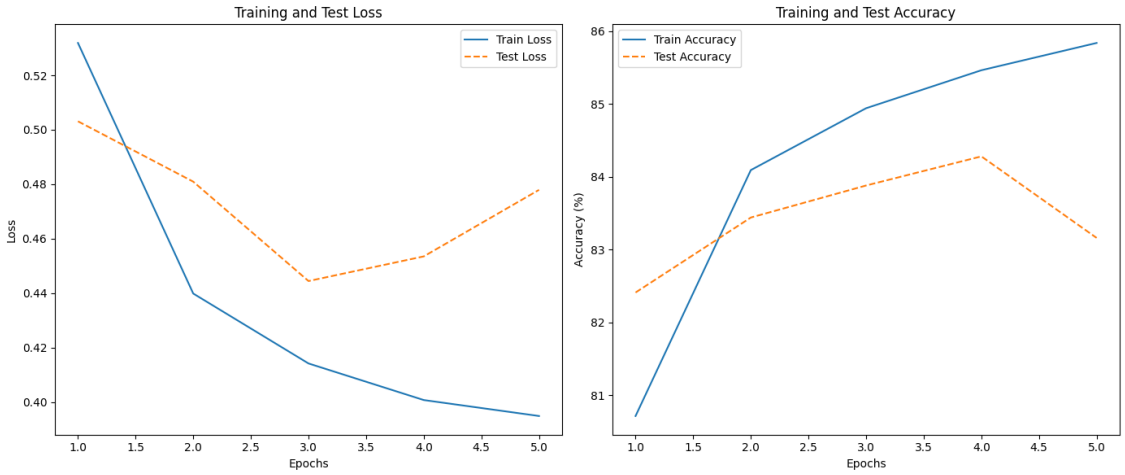
هایپرپارامترها در شبکه‌های عصبی پارامترهای مهمی هستند که پیش از شروع آموزش تنظیم می‌شوند و تأثیر زیادی بر یادگیری مدل دارند. این پارامترها شامل مواردی مانند نرخ یادگیری (learning rate) ، که سرعت به‌روزرسانی وزن‌ها را مشخص می‌کند؛ تعداد لایه‌ها و نورون‌ها، که معماری شبکه را تعیین می‌کنند؛ اندازه دسته (batch size) ، که تعداد نمونه‌ها در هر گام آموزشی را مشخص می‌کند؛ و تعداد epochs، که تعداد دفعات مشاهده کل داده‌ها توسط مدل را تعیین می‌کند، می‌شوند. تنظیم دقیق این هایپرپارامترها می‌تواند دقت، سرعت همگرایی و توانایی تعمیم‌دهی مدل را بهبود بخشد. معمولاً بهترین مقادیر هایپرپارامترها با آزمون و خطا یا روش‌های خودکار مانند grid search و random search پیدا می‌شوند، و تنظیم درست آنها به مدل کمک می‌کند تا عملکرد بهینه‌ای داشته باشد.

حال در این بخش، 3 هایپرپارامتر را تغییر می‌دهیم و نتایج آن را گزارش می‌کنیم:

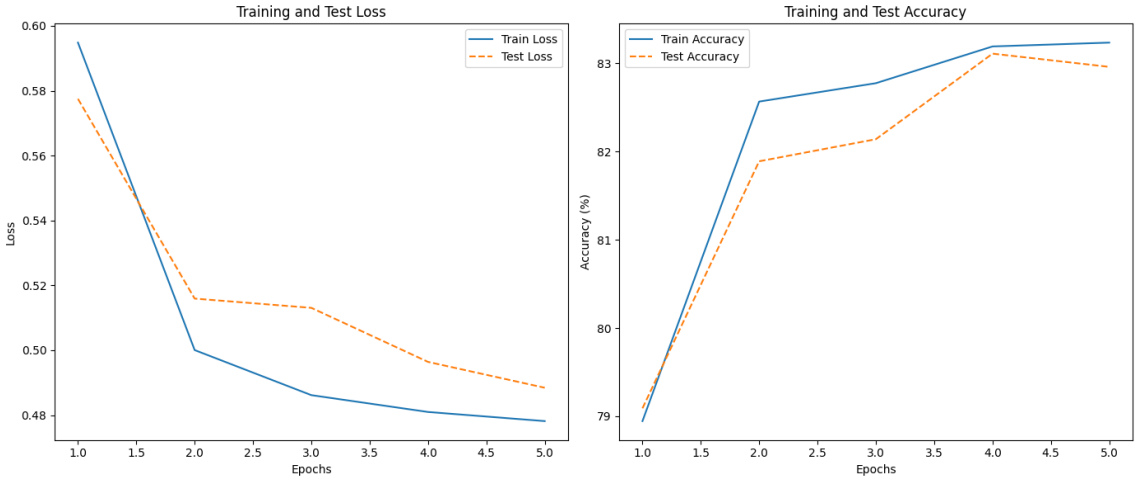
- **نرخ یادگیری (Learning rate):**




شکل 1-16. سرعت و نحوه همگرایی هزینه و دقت طبقه‌بندی با نرخ یادگیری 0.001




شکل 1-17. سرعت و نحوه همگرایی هزینه و دقت طبقه‌بندی با نرخ یادگیری 0.005



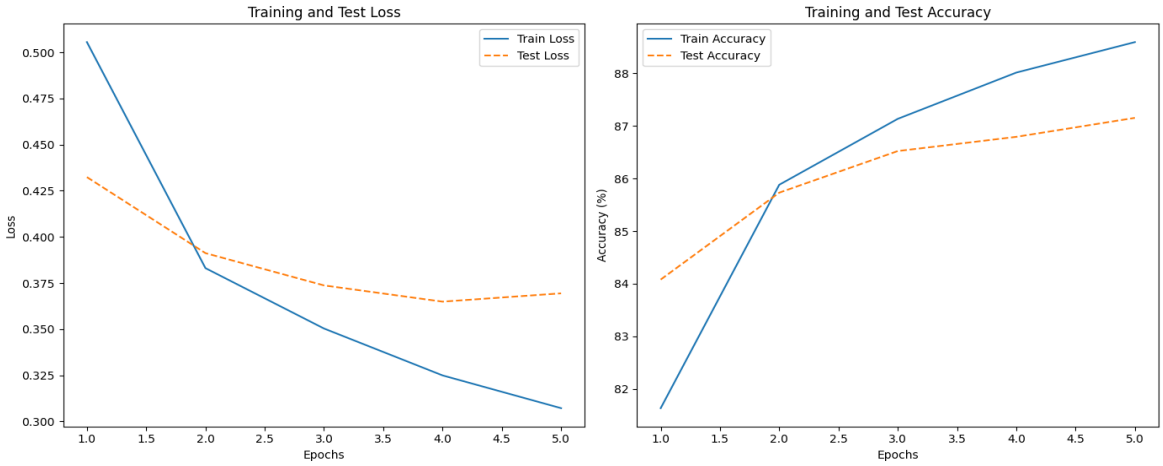

شکل 1-18. سرعت و نحوه همگرایی هزینه و دقت طبقه‌بندی با نرخ یادگیری 0.01

نرخ یادگیری یک پارامتر حساس در آموزش شبکه‌های عصبی است که سرعت به‌روزرسانی وزن‌ها در هر گام آموزشی را تعیین می‌کند. نرخ یادگیری بیش از حد پایین باعث می‌شود مدل به‌آرامی یاد بگیرد و

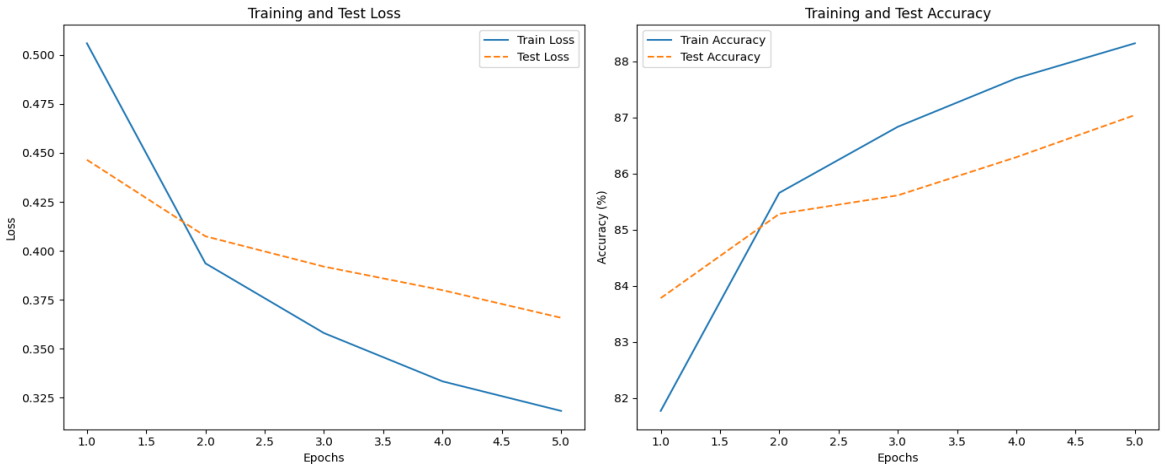
ممکن است به زمان زیادی برای همگرا شدن نیاز داشته باشد؛ همچنین، ممکن است مدل در یک مینیمم محلی گیر کند و بهینه‌ترین نتیجه را به دست نیاورد. از طرف دیگر، نرخ یادگیری بیش از حد بالا می‌تواند باعث شود مدل به سرعت از مینیمم‌های بهینه عبور کند و حتی به جای همگرایی، نوسانات بزرگی در به‌روزرسانی‌ها ایجاد شود و در نهایت از رسیدن به پاسخ بهینه باز بماند.

با توجه به نتایج، از نرخ یادگیری 0.001 استفاده می‌کنیم.

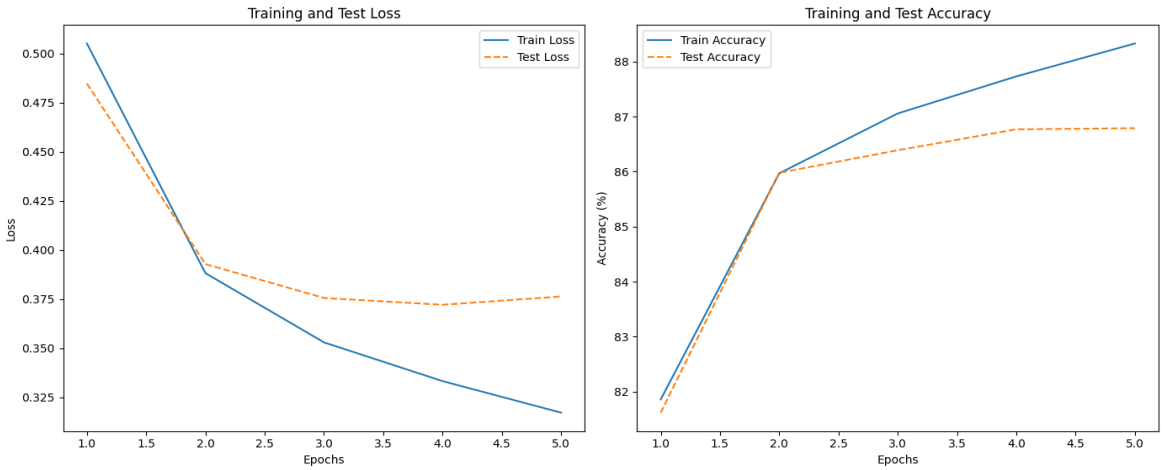
- **سایز دسته (Batch size):**




شکل 1-18. سرعت و نحوه همگرایی هزینه و دقت طبقه‌بندی با اندازه دسته 16




شکل 1-19. سرعت و نحوه همگرایی هزینه و دقت طبقه‌بندی با اندازه دسته 32

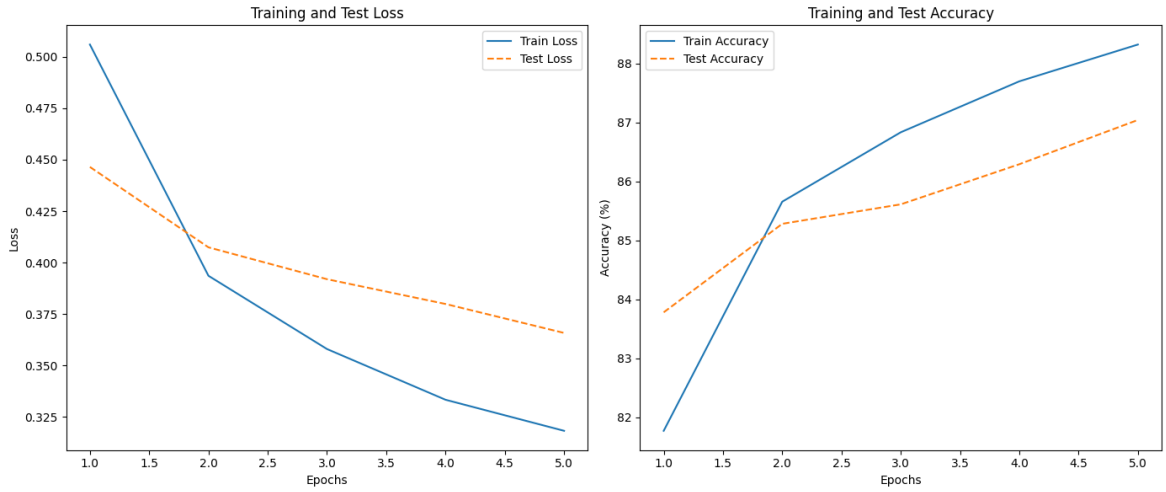



شکل 1-19. سرعت و نحوه همگرایی هزینه و دقت طبقه‌بندی با اندازه دسته 64

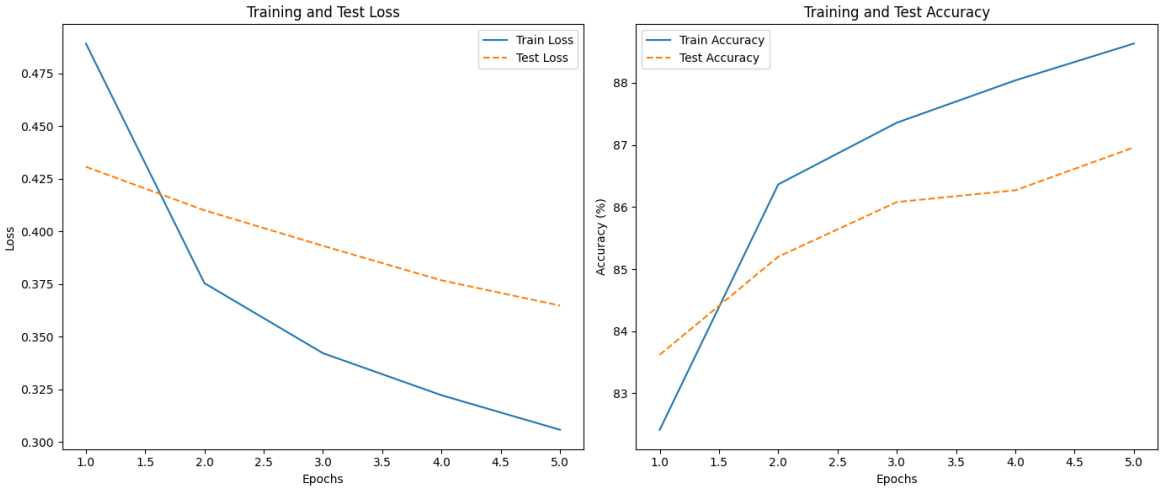
اندازه دسته (Batch size) در آموزش شبکه‌های عصبی تعداد نمونه‌ها در هر به‌روزرسانی مدل را تعیین می‌کند و تأثیر زیادی بر یادگیری دارد. دسته‌های کوچک به‌روزرسانی‌های سریع‌تر و متنوع‌تری فراهم می‌کنند و به مدل کمک می‌کنند تا از مینیمم‌های محلی عبور کند، اما این روش می‌تواند ناپایداری بیشتری ایجاد کند و زمان بیشتری برای آموزش نیاز دارد. در مقابل، دسته‌های بزرگ‌تر به‌روزرسانی‌های پایدارتر و با واریانس کمتر دارند که همگرایی مدل را باثبات‌تر می‌کند، ولی ممکن است به حافظه بیشتری نیاز داشته باشد و همگرایی را کندتر کند. انتخاب اندازه دسته مناسب، تعادلی بین سرعت، پایداری و دقت مدل ایجاد می‌کند.

با توجه به نتایج، اندازه دسته 32 انتخاب می‌شود.

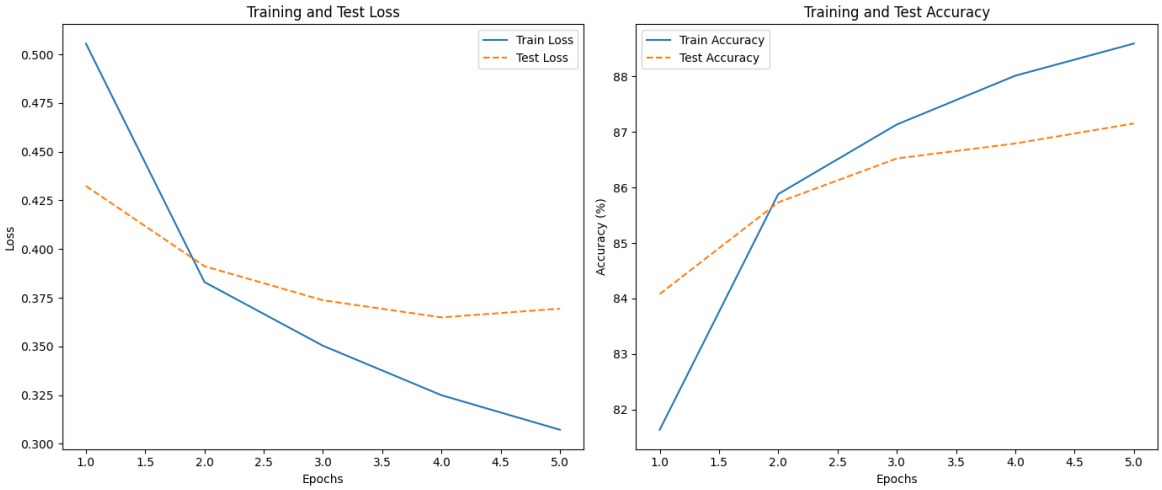
- **معماری مدل (تعداد لایه‌های مخفی و نورون‌ها):**




شکل 1-20. سرعت و نحوه همگرایی هزینه و دقت طبقه‌بندی با یک لایه مخفی با 64 نورون




شکل 1-21. سرعت و نحوه همگرایی هزینه و دقت طبقه‌بندی با یک لایه مخفی با 128 نورون




شکل 1-22. سرعت و نحوه همگرایی هزینه و دقت طبقه‌بندی با دو لایه مخفی با 64 و 128 نورون

با توجه به نتایج، معماری با دو لایه مخفی با 64 و 128 نورون انتخاب می‌شود.

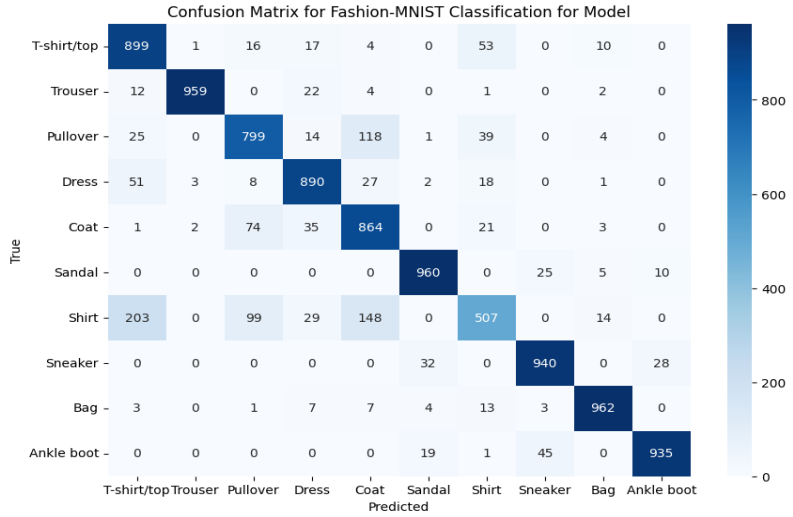
**پرسش:** توضیح دهید چگونه روش های بهینه سازی هایپرپارامتر مانند جستجوی تصادفی می توانند به انتخاب بهترین ترکیب ها کمک کنند؟

روش‌های بهینه‌سازی هایپرپارامتر مانند جستجوی تصادفی می‌توانند سرعت یافتن ترکیب‌های بهینه هایپرپارامترها را افزایش دهند و به یافتن بهترین تنظیمات برای مدل کمک کنند. در جستجوی تصادفی، به جای بررسی تمام ترکیبات ممکن (مانند جستجوی شبکه‌ای)، مقادیر مختلف برای هایپرپارامترها به صورت تصادفی انتخاب و آزمایش می‌شوند. این روش به دلیل انتخاب تصادفی، امکان تست ترکیب‌های متنوع‌تری را فراهم می‌کند و احتمال یافتن ترکیب بهینه در زمانی کمتر نسبت به جستجوی شبکه‌ای را

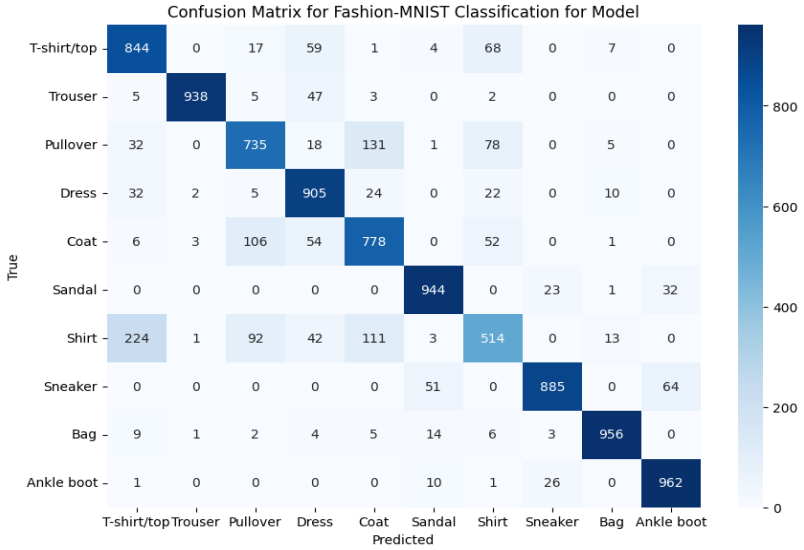
افزایش می‌دهد. از آنجایی که در جستجوی شبکه‌ای فقط مقادیر خاص و از پیش‌تعریف‌شده‌ای از هایپرپارامترها بررسی می‌شوند، برخی ترکیب‌های بهینه ممکن است نادیده گرفته شوند. اما در جستجوی تصادفی، فضای هایپرپارامترها گسترده‌تر و متنوع‌تر پوشش داده می‌شود.

**پرسش:** از نتایج ماتریس آشفتگی برای بررسی دقیق تر کلاس هایی که بیشتر اشتباه گرفته میشوند، استفاده کنید و تحلیل کنید تغییر هر کدام از هایپرپارامترها چه تغییری روی کلاس هایی که باهم اشتباه گرفته میشوند دارد؟ چرا؟

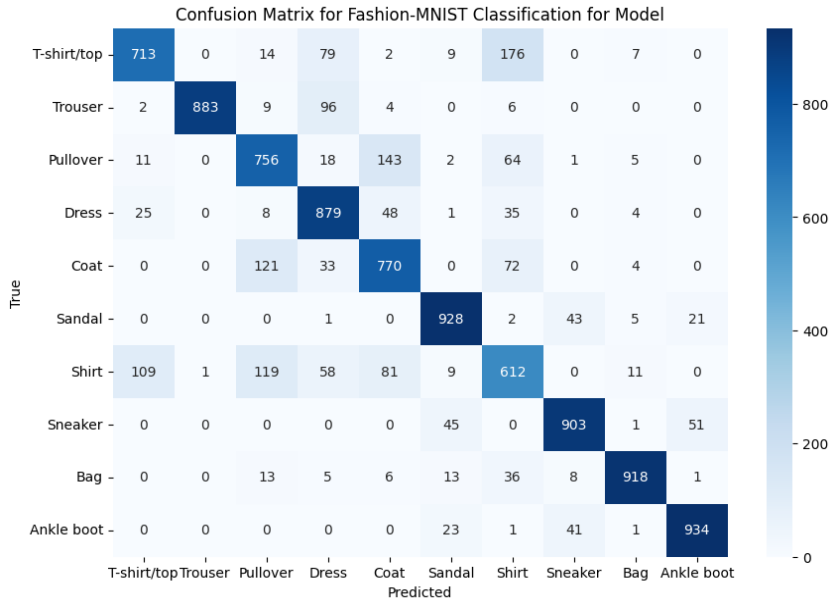
- **نرخ یادگیری (Learning rate):**




شکل 1-23. ماتریس آشفتگی طبقه بندی با نرخ یادگیری 0.001



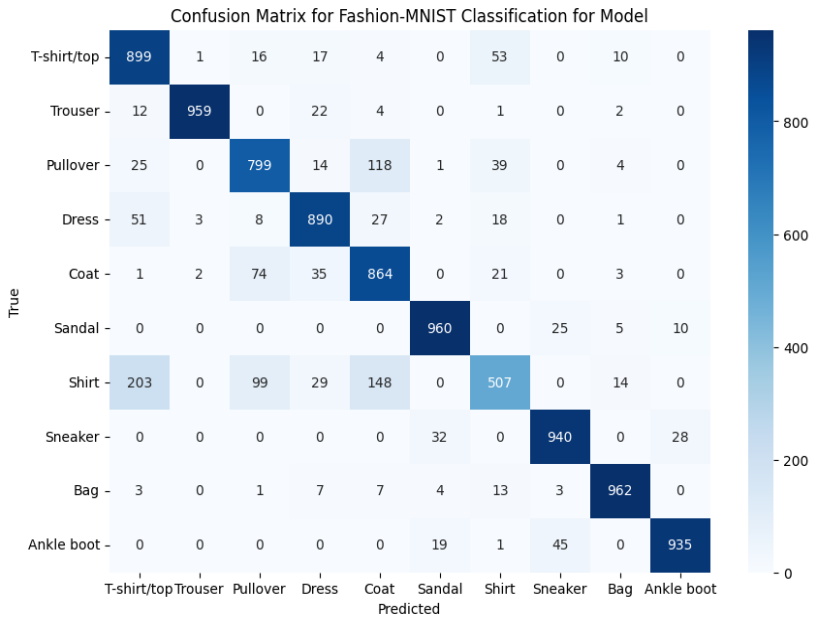

شکل 1-24. ماتریس آشفتگی طبقه بندی با نرخ یادگیری 0.005



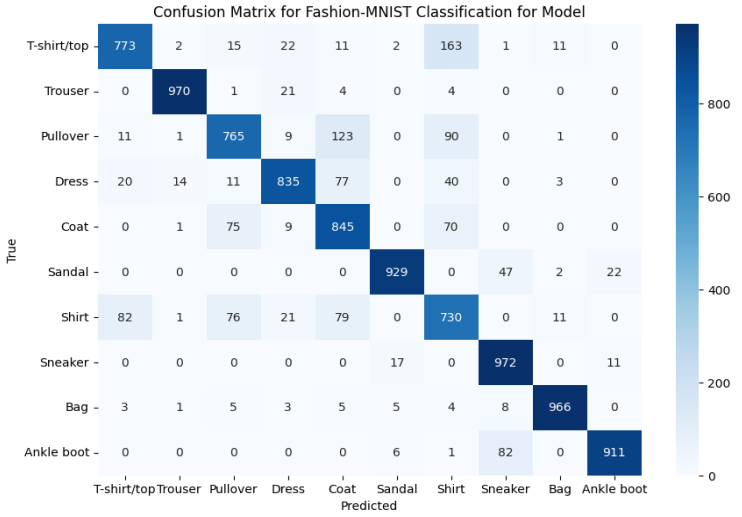

شکل 1-25. ماتریس آشفتگی طبقه بندی با نرخ یادگیری 0.01

با توجه به نتایج، با افزایش نرخ یادگیری، میزان اشتباهات به دلیل کاهش دقت، افزایش می‌یابد و دو کلاس Shirt و T-shirt/top بیش از پیش با هم اشتباه گرفته می‌شوند. این به علت واریانس بالای ناشی از overfitting می‌باشد.

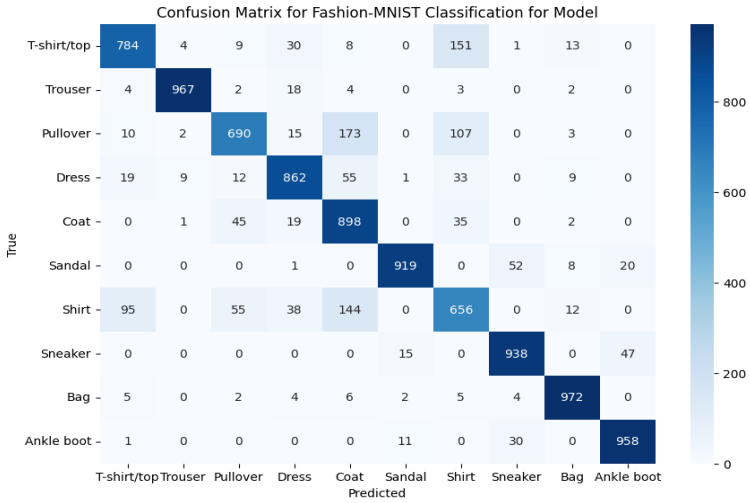
- **سایز دسته (Batch size):**




شکل 1-26. ماتریس آشفتگی طبقه بندی با اندازه دسته 16




شکل 1-27. ماتریس آشفتگی طبقه بندی با اندازه دسته 32



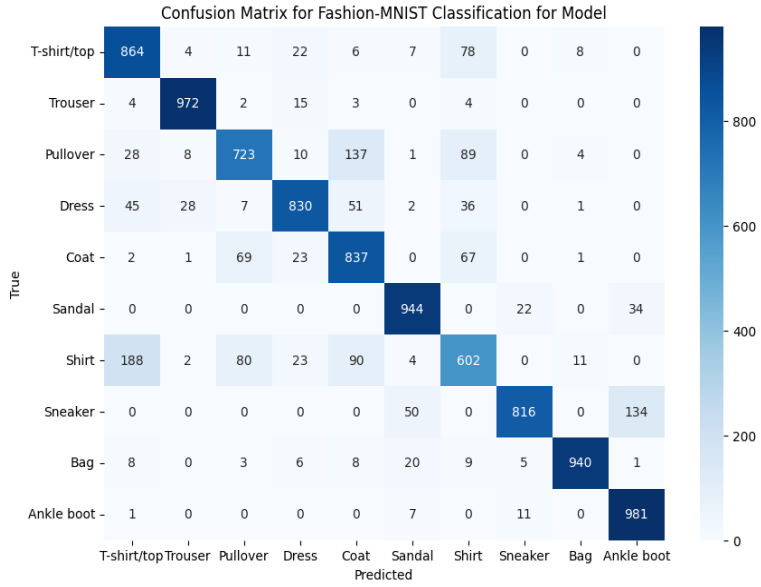

شکل 1-28. ماتریس آشفتگی طبقه بندی با اندازه دسته 64

اگر اندازه دسته خیلی زیاد باشد، سرعت همگرایی و پایداری افزایش اما میزان generalization کاهش می‌یابد.

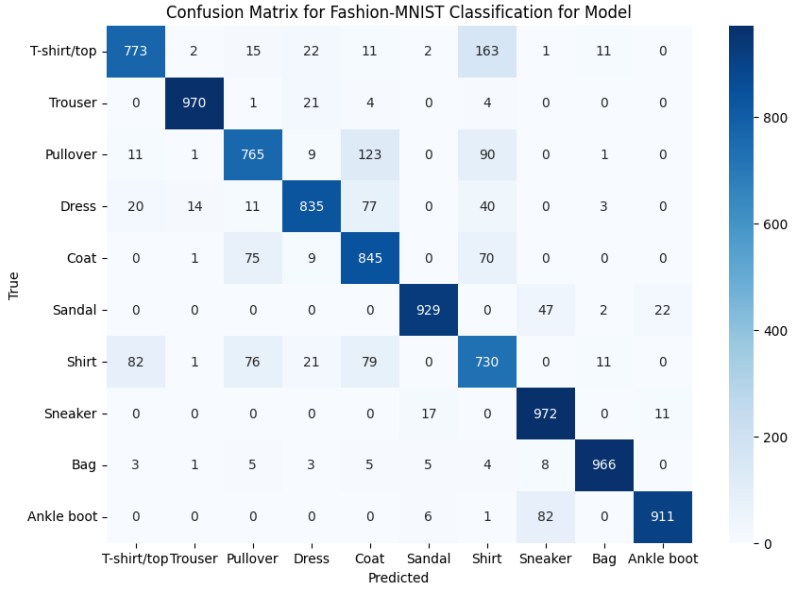
اگر اندازه دسته خیلی کم باشد، سرعت همگرایی و پایداری کاهش اما میزان generalization افزایش می‌یابد.

برای ایجاد یک مدل دارای هر 3 ویژگی، باید یک چیز متوسط انتخاب شود تا میزان اشتباهات کاهش یابد. به همین دلیل در اندازه دسته 32، کمترین اشتباهات را داریم.

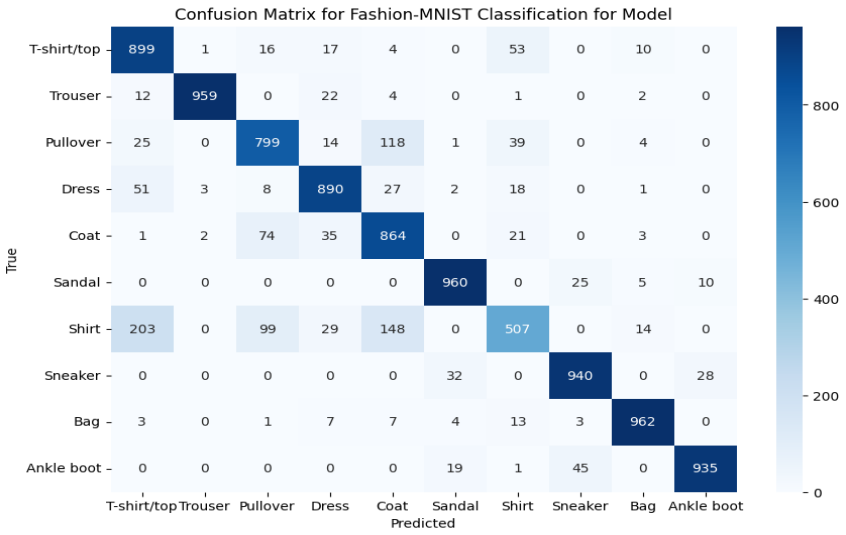
## معماری مدل (تعداد لایه‌های مخفی و نورون‌ها):




شکل 1-29. ماتریس آشفتگی طبقه بندی با یک لایه مخفی با 64 نورون




شکل 1-30. ماتریس آشفتگی طبقه بندی با یک لایه مخفی با 128 نورون




شکل 1-31. ماتریس آشفتگی طبقه بندی با یک لایه مخفی با 128 نورون

مطابق نتایج، مدل ما نباید خیلی پیچیده و نباید خیلی ساده باشد تا میزان overfitting و underfitting به حداقل برسد و مدل general باشد. به همین دلیل مدل با یک لایه مخفی با 128 نورون کمترین میزان اشتباهات را دارد.

.

# پرسش ۲ – آموزش و ارزیابی یک شبکه عصبی ساده

## ۲-۱. آموزش یک شبکه عصبی

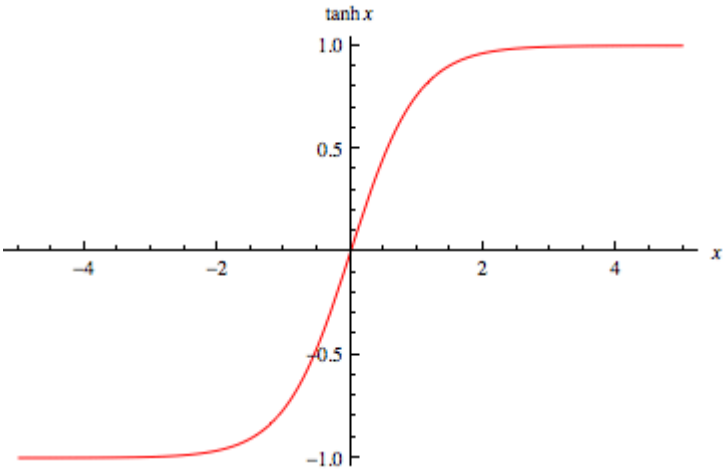
الف) نوشتن تابع forward

```
def forward(X, W1, W2):
    Z = np.tanh(np.dot(X, W1.T))
    y_pred = np.dot(Z, W2.T)

    return y_pred, Z
```

در این تابع ورودی از طریق یک لایه پنهان با یک تابع فعال‌سازی غیرخطی (Tanh) عبور می‌کند و سپس از طریق یک لایه خروجی با یک تابع خطی به پیش‌بینی نهایی می‌رسد.

فرمول تابع فعال‌سازی Tanh به شکل زیر است:

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$




شکل 2- 1. نمودار تابع **Tanh**

ابعاد ماتریس ورودی X ابعادی برابر (N, D) است که در آن N تعداد نمونه‌ها و D تعداد ویژگی‌های ورودی است. ماتریس وزن $W_1$ برای لایه پنهان به ابعاد (M, D) است که M تعداد نورون‌ها در لایه پنهان است.

تبدیل از لایه ورودی به لایه پنهان بر اساس فرمول زیر انجام می‌شود:

$$Z = \tanh(XW_1^T)$$

بنابراین، هر نورون در لایه پنهان یک مجموع وزنی از ویژگی‌های ورودی دریافت می‌کند که سپس از طریق تابع Tanh عبور کرده و فعال‌سازی‌هایی تولید می‌شود که در بازه [1 ,1-] قرار دارند.

اکنون فعال‌سازی‌های Z از لایه پنهان به ورودی لایه خروجی تبدیل می‌شوند.

$$y_{\text{pred}} = ZW_2^T$$

در این حالت، $y_{\text{pred}}$ یک بردار پیش‌بینی برای هر نمونه ورودی است. از آنجا که پس از این لایه هیچ تابع فعال‌سازی وجود ندارد، این لایه به‌طور خطی عمل می‌کند.

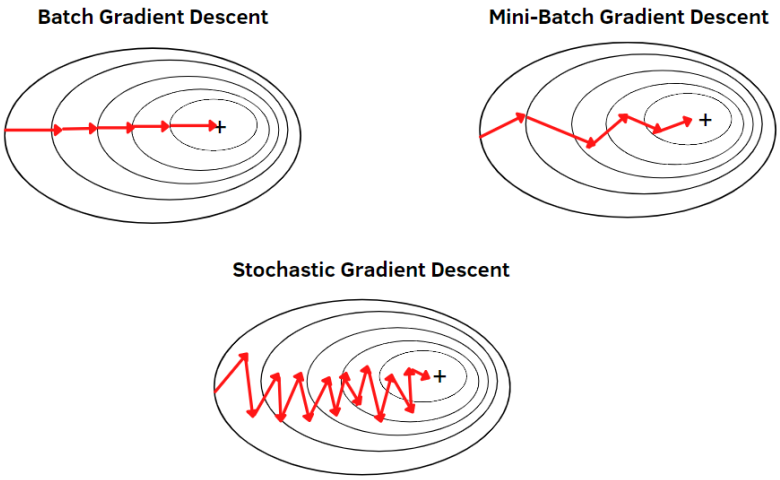
ب) نوشتن تابع backward

```
def backward(X, y, M, iters, lr):
    N, D = X.shape

    W1 = np.random.randn(M, D) * np.sqrt(1 / D)
    W2 = np.random.randn(1, M) * np.sqrt(1 / M)
    error_over_time = []

    for i in range(iters):
        idx = np.random.randint(0, N)
        X_sample = X[idx:idx+1]
        y_sample = y[idx:idx+1]

        y_pred, Z = forward(X_sample, W1, W2)

        loss = mse(y_sample, y_pred)
        error_over_time.append(loss)

        dZ2 = y_pred - y_sample  # 1 x 1
        dW2 = np.dot(dZ2, Z)  # (1 x 1) @ (1 x M) = 1 x M
        dZ1 = np.dot(dZ2, W2) * (1 - Z**2)  # (1 x 1) @ (1 x M) *
(1 x M) = 1 x M
        dW1 = np.dot(dZ1.T, X_sample)  # (M x 1) @ (1 x D) = M x D


        W1 -= lr * dW1
        W2 -= lr * dW2

    return W1, W2, np.array(error_over_time)
```

مقداردهی اولیه وزن ها بر اساس Xavier Initialization انجام شده است. در این روش، وزن‌ها به‌گونه‌ای مقداردهی می‌شوند که مانع از Vanishing Gradientیا Exploding Gradient که به دلیل ضرب ماتریسی‌های متوالی به وجود می‌آیند، شود. اگر وزن‌ها بیش از حد بزرگ یا کوچک باشند، به مرور در

فرآیند back propagation، مقدار loss یا خیلی کوچک و یا خیلی بزرگ می‌شوند که باعث کند شدن یا حتی توقف یادگیری می‌شود.

در هر iteration، یک نمونه تصادفی از داده‌ها انتخاب می‌شود و prediction و loss آن محاسبه می‌شود. این تکنیک یادگیری Stochastic Gradient Descent است که به جای استفاده از تمام داده‌ها در هر iteration، یک نمونه تصادفی انتخاب میکند که باعث میشود زمان محاسبات بسیار کاهش یابد. همچنین این روش به دلیل تصادفی بودن به جلوگیری از گیر افتادن در local minimum کمک کرده و به همگرایی به مینیمم‌های بهتر نزدیک‌تر می‌شود. البته این روش معایبی نیز دارد. مانند نوسانات در فرآیند همگرایی که ممکن است باعث شود مدل به جای optimal minimum، به نقاط نامناسبی برسد.




شکل 2-2. نمایی از نواع **Gradient Descent**

برای به‌روزرسانی وزن‌ها، در ابتدا تابع forward برای محاسبه پیش‌بینی‌ها و همچنین خروجی لایه پنهان فراخوانی می‌شود. سپس با استفاده از back propagation، گرادیان‌ها محاسبه می‌شوند.

خطای پیش‌بینی $dZ2$ از فرمول زیر بدست میاید. این مقدار نشان‌دهنده خطا در خروجی شبکه است و برای بهبود پیش‌بینی، این خطا برای به‌روزرسانی وزن‌ها استفاده می‌شود.

$$dZ2 = y_{\text{pred}} - y_{\text{sample}}$$

سپس، گرادیان وزن‌های $W2$ به شکل زیر محاسبه می‌شود:

$$dW2 = dZ2 \cdot Z$$

این فرموت نشان‌دهنده میزان تغییر مورد نیاز در وزن‌های لایه خروجی است. ماتریس گرادیان $dW2$ از حاصل‌ضرب خطای پیش‌بینی $dZ2$ و خروجی لایه پنهان Z به دست می‌آید.

برای به‌روزرسانی وزن‌های لایه پنهان، ابتدا گرادیان خروجی برای نورون‌های لایه پنهان $dZ1$ محاسبه می‌شود. این مقدار با درنظرگرفتن خطای لایه خروجی و مشتق تابع فعال‌سازی Tanh به دست می‌آید و مشخص میکند که ب چگونه گرادیان خطا از لایه خروجی به لایه پنهان propagate می‌شود و به نوعی ارتباط بین خطاهای این دو لایه را فراهم می‌کند.

$$dZ1 = dZ2 \cdot W_2 \cdot (1 - Z^2)$$

گرادیان وزن‌های $W\_1$ با استفاده از فرمول زیر محاسبه می‌شود:

$$dW1 = dZ1^T \cdot X_{\text{sample}}$$

پس از محاسبه گرادیان‌ها، وزن‌ها با استفاده از learning rate تنظیم می‌شوند. این به‌روزرسانی به گونه‌ای انجام می‌شود که در هر تکرار وزن‌ها به سمت کاهش خطا حرکت کنند:

$$W1 = W1 - \text{lr} \cdot dW1$$
$$W2 = W2 - \text{lr} \cdot dW2$$

## ۲-۲. آزمون شبکه عصبی بر روی یک مجموعه داده

در این قسمت از دیتاست wine quality برای آموزش شبکه عصبی طراحی شده استفاده میکنیم.

| | fixed acidity | volatile acidity | citric acid | residual sugar | chlorides | free sulfur dioxide | total sulfur dioxide | density | pH | sulphates | alcohol | quality |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 7.4 | 0.70 | 0.00 | 1.9 | 0.076 | 11.0 | 34.0 | 0.9978 | 3.51 | 0.56 | 9.4 | 5 |
| 1 | 7.8 | 0.88 | 0.00 | 2.6 | 0.098 | 25.0 | 67.0 | 0.9968 | 3.20 | 0.68 | 9.8 | 5 |
| 2 | 7.8 | 0.76 | 0.04 | 2.3 | 0.092 | 15.0 | 54.0 | 0.9970 | 3.26 | 0.65 | 9.8 | 5 |
| 3 | 11.2 | 0.28 | 0.56 | 1.9 | 0.075 | 17.0 | 60.0 | 0.9980 | 3.16 | 0.58 | 9.8 | 6 |
| 4 | 7.4 | 0.70 | 0.00 | 1.9 | 0.076 | 11.0 | 34.0 | 0.9978 | 3.51 | 0.56 | 9.4 | 5 |

شکل 2-3. نمونه هایی از دیتاست **wine quality**

در ابتدا دیتاست را با ratio = 0.5 به دو دسته train و test تقسیم میکنیم. سپس دیتاهای train و test نرمالیزه می‌شوند تا ویژگی ها یک scale مشابه داشته باشند. در اینجا نرمالایز کردن دیتا با استفاده از Z-score Normalization انجام می‌شود:

$$X_{\text{train}} = \frac{X_{\text{train}} - \mu}{\sigma}, \quad X_{\text{test}} = \frac{X_{\text{test}} - \mu}{\sigma}$$

توجه کنید که نرمالایز کردن داده train و test هر دو با استفاده از میانگین $(\mu)$ و انحراف معیار $(\sigma)$ داده‌های **train** انجام می‌شود. این کار برای جلوگیری از Data Leakage است، به این معنی که دیتاهای test نباید بر فرآیند آموزش تاثیر بگذارند و باید از اطلاعات train برای نرمالیزه کردن دیتاهای test استفاده کرد.

سپس باید bias به ستون‌های train و test اضافه کرد. در مدل‌های خطی، افزودن bias از این نظر مهم است تا مدل بتواند داده‌ها را به‌درستی تغییر مقیاس داده و از مرکز مختصات به‌خوبی جدا کند. به طور کلی، این ستون بایاس باعث می‌شود که مدل برای پیش‌بینی از فرمول زیر استفاده کند:

$$y = w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n$$

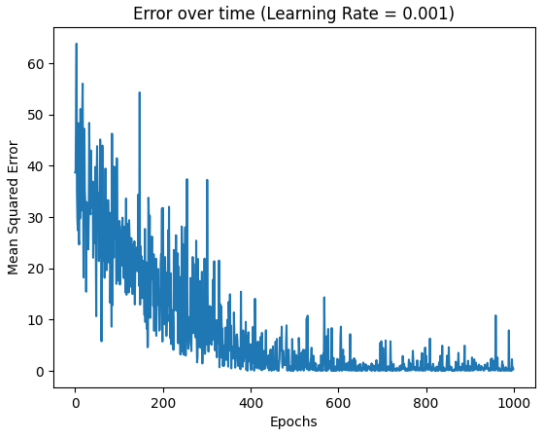
که در آن $w_0$ به عنوان bias عمل می‌کند. بنابراین، اضافه‌کردن این ستون از یک‌ها معادل اضافه‌کردن $w_0$ به وزن‌ها است که مدل را انعطاف‌پذیرتر میکند.

در بخش بعدی به آموزش مدل در 1000 epoch بر اساس learning rate های مختلف میپردازیم و خطای مدل را با معیار های Mean Squared Error و Root Mean Squared Error گزارش میکنیم.

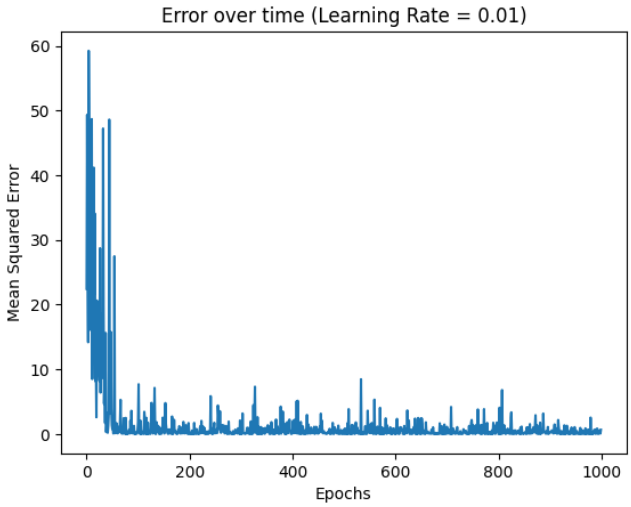
$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}\left(y_{\text{true},i} - y_{\text{pred},i}\right)^2$$

$$\text{RMSE} = \sqrt{MSE}$$

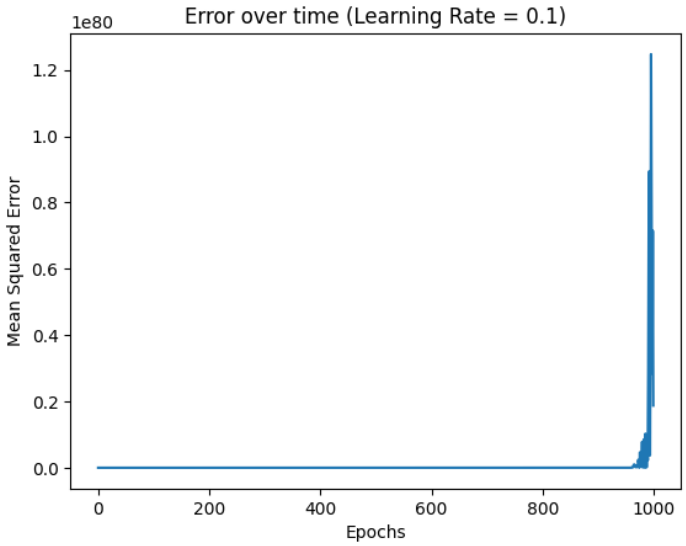
هر دو معیارهایی برای اندازه‌گیری دقت مدل‌های رگرسیون هستند. MSE به دلیل مربعی‌بودن خطاها، خطاهای بزرگ را بیشتر از خطاهای کوچک‌تر جریمه می‌کند و به همین دلیل به تغییرات و انحرافات بزرگ حساس است. RMSE ریشه‌ی MSE را می‌گیرد تا خطا را به واحد اصلی داده‌ها بازگرداند. این معیار به تفسیر خطای مدل کمک بیشتری می‌کند، زیرا در همان واحد متغیر اصلی است. RMSE نیز مانند MSE به خطاهای بزرگ حساس است، اما قابل تفسیرتر است و معمولا برای ارزیابی بهتر مدل ترجیح داده می‌شود.




شکل 2-4. نمودار خطا **MSE** بر حسب زمان با **learnig rate = 0.001**



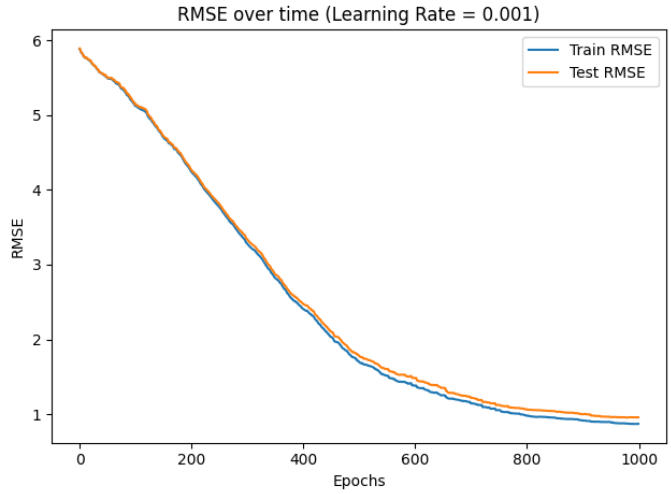

شکل 2-5. نمودار خطا **MSE** بر حسب زمان با **learnig rate = 0.01**



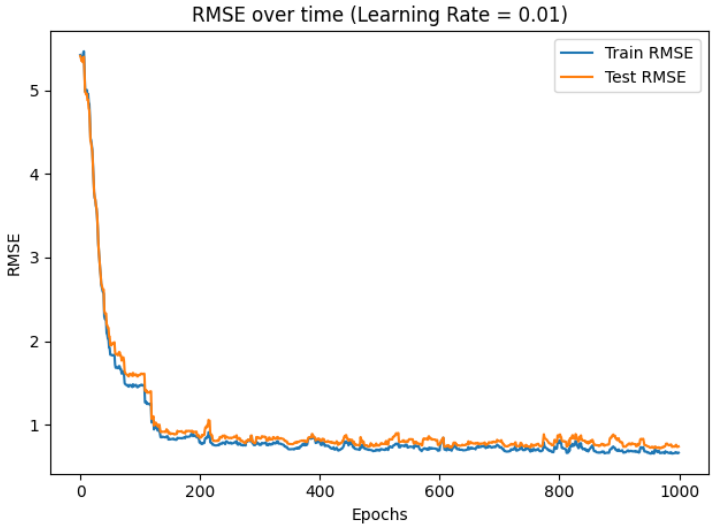

شکل 2-6. نمودار خطا **MSE** بر حسب زمان با **learnig rate = 0.1**

جدول 2-1. گزارش **RMSE** بر اساس **learning rate**

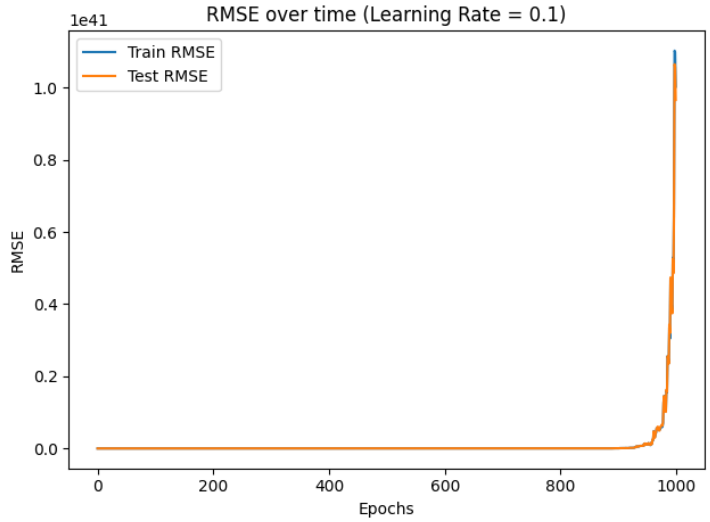
| **learning rate** | 0.001 | 0.01 | 0.1 |
|---|---|---|---|
| **RMSE** | 0.848 | 0.692 | 9.326e+39 |




شکل 2-7. نمودار خطا **RMSE** بر حسب زمان با **learnig rate = 0.001**




شکل 2-8 نمودار خطا **RMSE** بر حسب زمان با **learnig rate = 0.01**




شکل 2-9 نمودار خطا **RMSE** بر حسب زمان با **learnig rate = 0.1**

تحلیل هر بخش:

**Learning Rate = 0.001:**

- در این حالت، خطای MSE به تدریج کاهش می‌یابد. کاهش خطا نشان‌دهنده آن است که مدل در حال یادگیری و بهبود است.
- نرخ یادگیری 0.001 به مدل اجازه می‌دهد تا به صورت آهسته و پیوسته به سمت کمینه بهینه حرکت کند. به همین دلیل، خطا به آرامی کاهش می‌یابد و در نهایت در یک مقدار ثابت تثبیت می‌شود.

**Learning Rate = 0.01:**

- با این نرخ یادگیری، کاهش خطای MSE نسبت به نرخ یادگیری 0.001 سریع‌تر است. در حدود 200 epoch، خطا به مقدار کمی می‌رسد و نوساناتی در آن دیده می‌شود، اما بهبود مدل ادامه دارد.
- این نرخ یادگیری سریع‌تر باعث می‌شود که مدل زودتر به local minium برسد، اما همچنان قابل کنترل است و به سمت همگرایی می‌رود.

**Learning Rate = 0.1:**

- در این حالت ، مدل به شدت ناپایدار است و خطای MSE به سرعت افزایش می‌یابد.
- نرخ یادگیری 0.1 برای این شبکه بسیار بالا بوده و این نرخ یادگیری باعث می‌شود که به‌روزرسانی‌های وزن‌ها بسیار شدید و خارج از کنترل، و یا Exploding Gradients رخ بدهد که منجر به نوسانات شدید یا واگرایی مدل از مسیر بهینه می‌شود.

**جمع‌بندی:**

در اینجا بهترین Learning Rate 0.01 است، زیرا این مقدار همزمان با سرعت مناسب و بدون نوسانات شدید، باعث کاهش خطای MSE و بهبود عملکرد مدل می‌شود. این نرخ به مدل اجازه می‌دهد تا به سرعت به سمت کمینه بهینه حرکت کند و در مدت زمان معقولی همگرا شود. این در مقایسه با Learning Rate 0.001 که کندتر است، یک تعادل خوب بین سرعت و دقت بهینه‌سازی فراهم می‌کند.

همچنین برخلاف Learning Rate 0.1 که منجر به ناپایداری و Exploding Gradients شد، Learning Rate 0.01 توانست مدل را در مسیر بهینه نگه دارد و با ثبات، به سمت همگرایی حرکت کند.

**آیا این نتیجه در شبکه‌های عصبی دیگر هم صدق می‌کند؟**

در کل میزان بهینه Learning Rate به ساختار شبکه و داده بستگی دارد. در شبکه‌های عصبی عمیق‌تر (مانند شبکه‌های convolutional یا recurrent)، تنظیم نرخ یادگیری مناسب بسیار مهم و همچنین وابسته به ساختار شبکه است. نرخ‌های یادگیری بالا باعث بروز Exploding Gradients یا ناپایدار می‌شود که موجب واگرایی مدل می‌گردد. در این شبکه‌ها، نرخ‌های یادگیری کوچک‌تر معمولا پایداری بیشتری به همراه دارند.

همچنین داده‌های پیچیده‌تر و نویزی ممکن است نیاز به نرخ یادگیری کوچک‌تری داشته باشند تا از نوسانات شدید جلوگیری شود. به همین دلیل، معمولا نرخ یادگیری بین مدل‌ها و مسائل مختلف تنظیم می‌شود.

البته در برخی موارد Learning Rate بزرگ نیز می‌تواند مفید و حتی ضروری باشد. مثلا در مراحل ابتدایی آموزش مدل، یک نرخ یادگیری بزرگ می‌تواند کمک کند که مدل سریع‌تر به سمت بهینه نسبی حرکت کند. این کار باعث می‌شود که مدل در ابتدا تغییرات بزرگتری را تجربه کند و از نقاط شروع تصادفی دور شود. و یا در مدل‌های ساده‌تر یا با داده‌هایی که پیچیدگی کمتری و الگوهای ساده‌تری دارند، نرخ یادگیری بزرگ می‌تواند کمک کند که مدل سریع‌تر به بهینه نهایی برسد.

## پرسش ۳ – Madaline

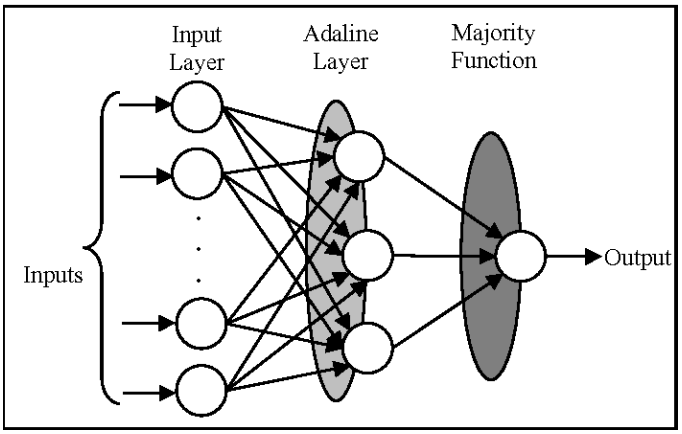
Adaline یک نوع واحد پردازشی ساده است که در اواخر دهه 1950 معرفی شد. این واحد بر اساس مدل نورون خطی کار می‌کند و از یک تابع فعال‌سازی پله‌ای استفاده می‌کند. یکی از ویژگی‌های اصلی Adaline این است که قادر به حل مسائل خطی جداپذیر است و تنها برای مسائل با الگوهای ساده و خطی مناسب است که فرمول آن به این شکل است

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

همچنین برخلاف شبکه های عصبی عمیق امروزی، از یک ترکیب خطی به همراه تابع پله‌ای استفاده می‌کند که در آن نیازی به محاسبه مشتق نیست.

$$\text{output} = \begin{cases} 1+ & \text{if } y \geq 0 \\ 1- & \text{if } y < 0 \end{cases}$$

MAdaline یک شبکه عصبی چندلایه است که از واحدهای Adaline در لایه‌های خود استفاده می‌کند. این شبکه اولین شبکه عصبی بود که به صورت سخت‌افزاری پیاده‌سازی شد.
برخلاف شبکه‌های عصبی امروزی که از back propagation خطا برای آموزش استفاده می‌کنند، MAdaline از قوانین MR-I و MR-II استفاده می‌کند که نیازی به مشتق‌پذیری تابع فعال‌سازی ندارند. به همین دلیل این شبکه‌ها می‌توانند از توابع پله‌ای استفاده کنند که در Adaline رایج هستند.




شکل 3-1. نمایی از یک شبکه MAdaline

### ۳-۱. MR-II

قانون MR-II یک قانون آموزش برای شبکه‌های MAdaline است که به‌ویژه در بهبود و گسترش فرآیند یادگیری این شبکه‌ها مورد استفاده قرار می‌گیرد. نسخه‌های ابتدایی MAdaline فقط وزن‌های لایه خروجی را به‌روزرسانی می‌کردند، اما قانون MR-II این محدودیت را از بین می‌برد و به شبکه این امکان را می‌دهد که وزن‌ها را **در تمام لایه‌ها** به‌طور هم‌زمان آموزش دهد که با استفاده از این روش، شبکه می‌تواند به‌طور موثرتری بهبود یابد و دقت مدل بالا رود. در این قانون برای هر ورودی، خطا به‌صورت مجموع مربعات خطاها در تمام واحدهای خروجی محاسبه می‌شود. این به این معناست که برای انجام به‌روزرسانی‌ها، تمام خروجی‌ها و خطاهای مربوط به آن‌ها در نظر گرفته می‌شود تا بتواند وزن‌ها را به‌درستی تنظیم کند.

MR-II از اصل Don’t Rock the Boat پیروی می‌کند که به این معناست که در هر مرحله از فرآیند یادگیری، کمترین تغییر ممکن در شبکه اعمال شود تا از unlearn کردن الگوهایی که شبکه قبلا آموزش دیده است جلوگیری شود. این اصل باعث می‌شود که فرآیند یادگیری به‌صورت تدریجی و با دقت انجام شود، به‌طوری که شبکه از یادگیری الگوهای جدید، الگوهای قدیمی‌تر را فراموش نکند.

(منبع این مطلب، کتاب Fundamentals of neural networks اثر Laurene V. Fausett است.)

## ۳-۲. نمودار پراکندگی داده‌ها

| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 0.459694 | -0.470583 | 0.0 |
| 1 | 0.797385 | -0.343030 | 0.0 |
| 2 | 0.235270 | 0.961296 | 0.0 |
| 3 | 0.765453 | -0.177644 | 0.0 |
| 4 | -0.335577 | -0.313893 | 0.0 |
| ... | ... | ... | ... |
| 195 | 0.744066 | -1.206548 | 1.0 |
| 196 | -0.457547 | 1.286227 | 1.0 |
| 197 | -1.020000 | -0.783926 | 1.0 |
| 198 | 1.363429 | -0.800250 | 1.0 |
| 199 | -1.246702 | -0.388615 | 1.0 |

شکل 3-2. نمونه هایی از دیتاست


شکل 3-3. نمودار پراکندگی داده‌ها

یکی از نکاتی که قبل از پیاده‌سازی مدل باید به آن توجه کرد این است که MAdaline به مقادیر target -1 و +1 نیاز دارد، نه 0 و 1. این به این دلیل است که شبکه‌های MAdaline با توابع فعال‌سازی پله‌ای کار می‌کنند که در بالا فرمول آن ارائه شد. پس از قبل باید کلاس 0 را به -1 تبدیل کرد تا بتوان شبکه MAdaline را بر روی این دیتاست اجرا کرد.

## ۳-۳. آموزش مدل

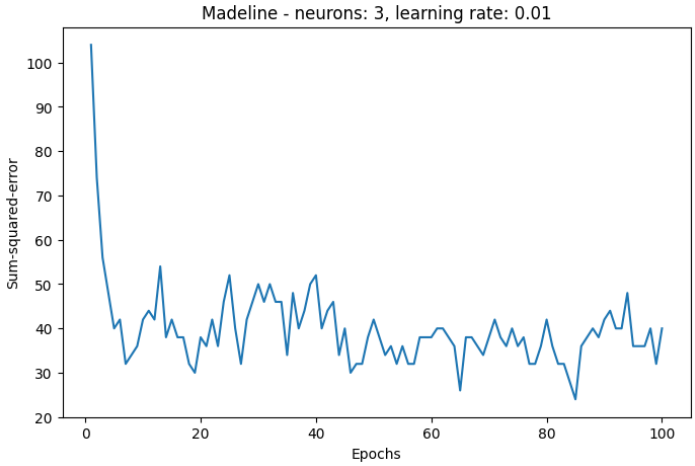
```
class Madaline:
    def __init__(self, n_neurons, lr, epochs):
        self.n_neurons = n_neurons
        self.lr = lr
        self.epochs = epochs

        self.bias = np.zeros(self.n_neurons) + 0.01
        self.output_layer_weights = np.zeros(self.n_neurons) + 1
        self.output_layer_bias = n_neurons - 1

    def activation_fn(self, x):
        return np.where(x >= 0.0, 1, -1)

    def loss(self, label, y_pred):
        return np.power((label - y_pred), 2) / 2.0

    def forward(self, x):
        z_input = np.sum(x * self.weights, axis=1) + self.bias
        z = np.array([self.activation_fn(x) for x in z_input])

        y_input = np.dot(z, self.output_layer_weights) +
self.output_layer_bias
        y_pred = self.activation_fn(y_input)
        return z_input, y_pred

    def update_weights(self, x, z_input, y_pred, label):
        if y_pred == label:
            return

        if label == 1.0:
            # For positive labels, update the neuron with maximum
activation
            max_activation_idx = z_input.argmax()
            target = 1.0
            mask = np.zeros_like(z_input, dtype=bool)
            mask[max_activation_idx] = True
        else:
            # For negative labels, update all neurons with positive
activation
            mask = z_input >= 0
            target = -1.0

        errors = target - z_input[mask]
        x_broadcast = x.reshape(1, -1) if mask.sum() > 1 else x
```

```
        self.weights[mask] += self.lr * errors.reshape(-1, 1) *
x_broadcast
        self.bias[mask] += self.lr * errors


    def fit(self, X, Y):
        self.weights = np.zeros(shape = (self.n_neurons,
X.shape[1])) + 0.01

        self.cost_per_epoch = []
        for epoch in tqdm(range(self.epochs)):
            cost = 0
            for x, label in zip(X, Y):
                z_input, y_pred = self.forward(x)
                cost += self.loss(label, y_pred)
                self.update_weights(x, z_input, y_pred, label)


            self.cost_per_epoch.append(cost)

        return self

    def predict(self, X):
        predicted = []
        for x in X:
            z_input, y_pred = self.forward(x)
            predicted.append(y_pred)

        return np.array(predicted)
```

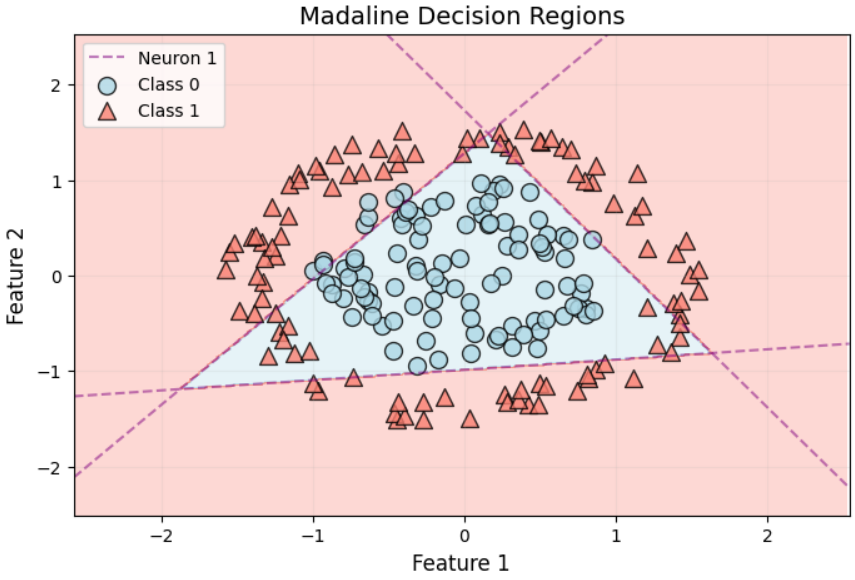
همانطور که در ابتدا گفته شد، تابع فعال‌سازی *activation_fn* یک تابع پله‌ای است که مقادیر ورودی را بر اساس شرط به 1 یا -1 تبدیل می‌کند. تابع *loss* خطای مربعی را محاسبه میکند تا میزان اختلاف پیش‌بینی از مقدار واقعی سنجیده شود.

تابع *forward* مقدار ورودی‌ها را به نورون‌های لایه میانی اعمال کرده و خروجی هر نورون را با استفاده از تابع فعال‌سازی محاسبه می‌کند. سپس خروجی لایه میانی برای پیش‌بینی نهایی به لایه خروجی داده می‌شود.

تابع *update_weights* به‌روزرسانی وزن‌ها را بر اساس خطای پیش‌بینی انجام می‌دهد. طبق اصل Don’t Rock the Boat اگر پیش‌بینی درست باشد، نیازی به به‌روزرسانی نیست؛ در غیر این صورت، برای کلاس مثبت نورونی با بیشترین فعال‌سازی و برای کلاس منفی همه نورون‌هایی که خروجی مثبت دارند، به‌روزرسانی انجام می‌شود.




شکل 3-4. نمودار خطا با 3 نورون

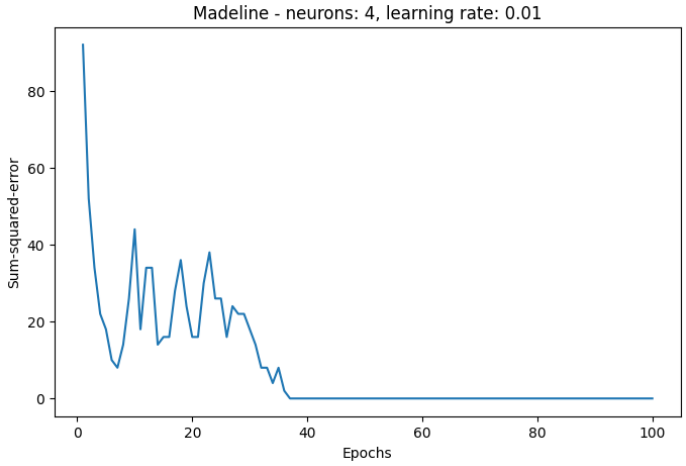



شکل 3-5. خطهای جداکننده با 3 نورون

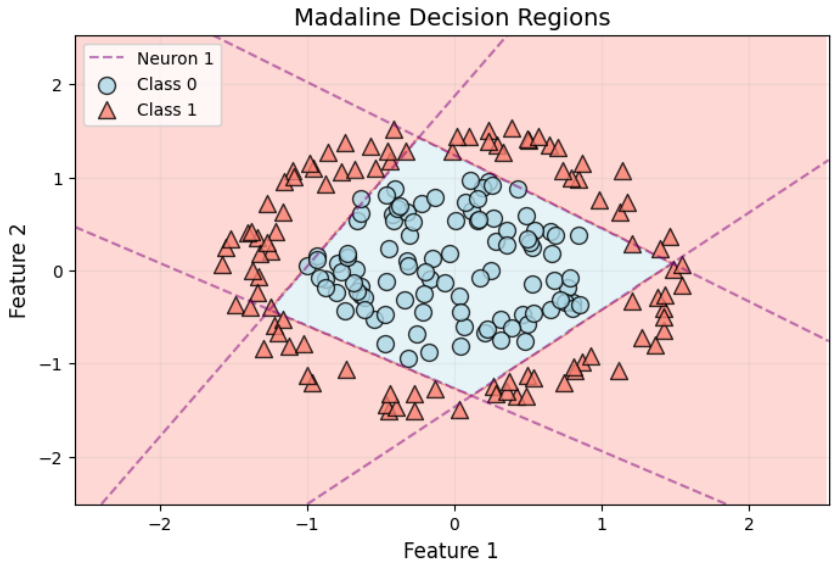
جدول 3-1. دقت آموزش و تست مدل با 3 نورون

| | |
|---|---|
| **Neurons** | 3 |
| **Train Accuracy** | 0.906 |
| **Test Accuracy** | 0.85 |

در این حالت چون تعداد نورون‌ها کم است، دقت چندان زیادی نمیتوانیم بگیریم و میبینیم که در epoch های بعدی هم خطا کم نمیشود و مشخص است که شبکه به اندازه کافی قدرت‌مند نیست.




شکل 3-6. نمودار خطا با 4 نورون

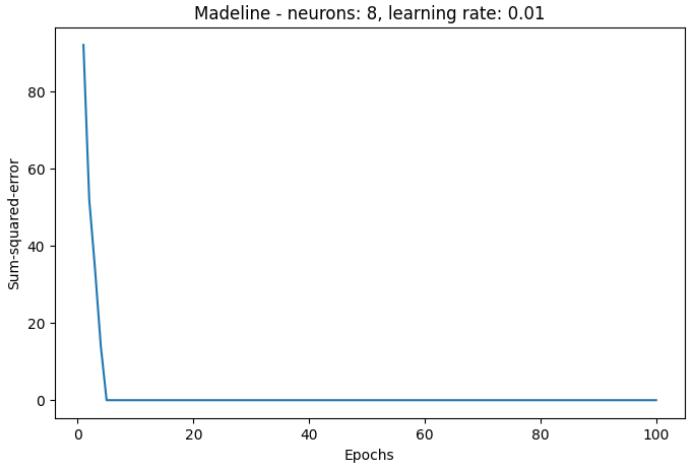



شکل 3-7. خط‌های جداکننده با 4 نورون

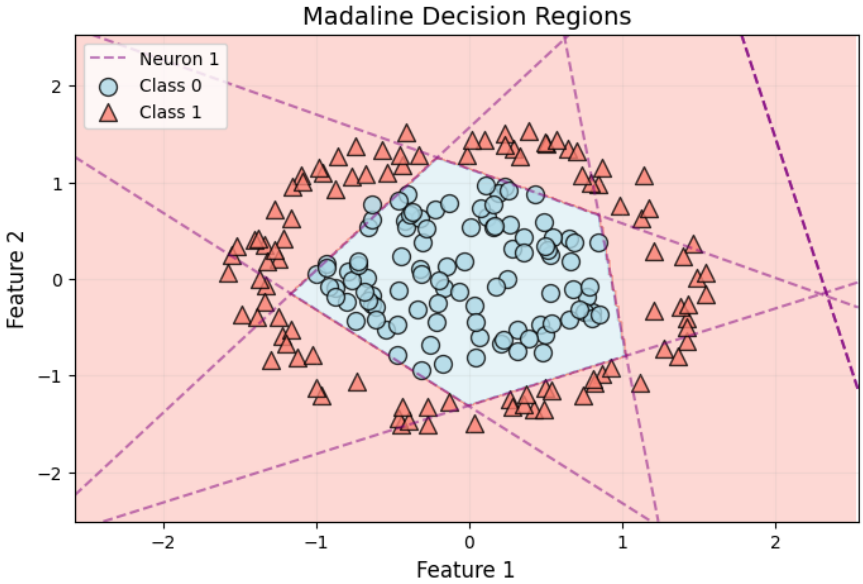
جدول 3-2. دقت آموزش و تست مدل با 4 نورون

| | |
|---|---|
| **Neurons** | 4 |
| **Train Accuracy** | 1.0 |
| **Test Accuracy** | 0.95 |

در این حالت مشخص است که تعداد نورون نسبت به 3 بهتر است و مدل به دقت کاملا خوبی رسیده است و به خوبی میتواند داده‌ها را طبقه‌بندی کند. همچنین بعد از حدودا 35 epoch، تغییری در مدل ایجاد نمیشود.




شکل 3-8. نمودار خطا با 8 نورون




شکل 3-9. خط‌های جداکننده با 8 نورون

جدول 3-3. دقت آموزش و تست مدل با 8 نورون

| **Neurons** | 8 |
|---|---|
| **Train Accuracy** | 1.0 |
| **Test Accuracy** | 0.975 |

در این حالت مشخص است که تعداد نورون نسبت به قبلی ها بهتر است و مدل به دقت بهتری از قبل رسیده است و به خوبی میتواند داده‌ها را طبقه‌بندی کند. همچنین به تعداد epoch بسیار کمتری برای آموزش نیاز داشته است. به دلیل تطابق خط ها روی یکدیگر، تعداد خط های روی شکل 8 عدد نیست.

**تحلیل نتایج:**

جدول 3-4. جمع‌بندی دقت آموزش و تست بر حسب تعداد نورون‌ها

| **Neurons** | 3 | 4 | 8 |
|---|---|---|---|
| **Train Accuracy** | 0.906 | 1.0 | 1.0 |
| **Test Accuracy** | 0.85 | 0.95 | 0.975 |

اگر بر اساس تعداد نورون بررسی کنیم، هرچقدر نورون های بیشتری در شبکه باشد، به دقت بیشتری میرسیم و همچنین در تعداد epoch کمتری به نتیجه خوب میرسیم. این مورد در مدل با 8 نورون به شدت قابل مشاهده است که در حدود 5 epoch، به دقتی بیشتر از مدل با 4 نورون در حدود 35 epoch رسید.

همچنین میتوانیم ببینیم که در مدل با 3 نورون، به همگرایی‌ای نرسیدیم که ممکن است به دلیل زیاد بودن نرخ یادگیری برای این تعداد نورون باشد و یا اینکه صرفا به این دلیل باشد که شبکه قدرت آموزش بیشتر بر روی دیتا را ندارد.

# پرسش ۴ – MLP

## ۴-۱. نمایش تعداد ستون

در این قسمت دیتاست داده شده را میخوانیم و تعداد Nan های هر ستون را بررسی میکنیم.

| date | price | bedrooms | bathrooms | sqft_living | sqft_lot | floors | waterfront | view | condition | grade | sqft_above | sqft_basement | yr_built | yr_renovated | zipcode | lat | long |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 20141013T000000 | 221900.0 | 3 | 1.00 | 1180 | 5650 | 1.0 | 0 | 0 | 3 | 7 | 1180 | 0 | 1955 | 0 | 98178 | 47.5112 | -122.257 |
| 20141209T000000 | 538000.0 | 3 | 2.25 | 2570 | 7242 | 2.0 | 0 | 0 | 3 | 7 | 2170 | 400 | 1951 | 1991 | 98125 | 47.7210 | -122.319 |
| 20150225T000000 | 180000.0 | 2 | 1.00 | 770 | 10000 | 1.0 | 0 | 0 | 3 | 6 | 770 | 0 | 1933 | 0 | 98028 | 47.7379 | -122.233 |
| 20141209T000000 | 604000.0 | 4 | 3.00 | 1960 | 5000 | 1.0 | 0 | 0 | 5 | 7 | 1050 | 910 | 1965 | 0 | 98136 | 47.5208 | -122.393 |
| 20150218T000000 | 510000.0 | 3 | 2.00 | 1680 | 8080 | 1.0 | 0 | 0 | 3 | 8 | 1680 | 0 | 1987 | 0 | 98074 | 47.6168 | -122.045 |

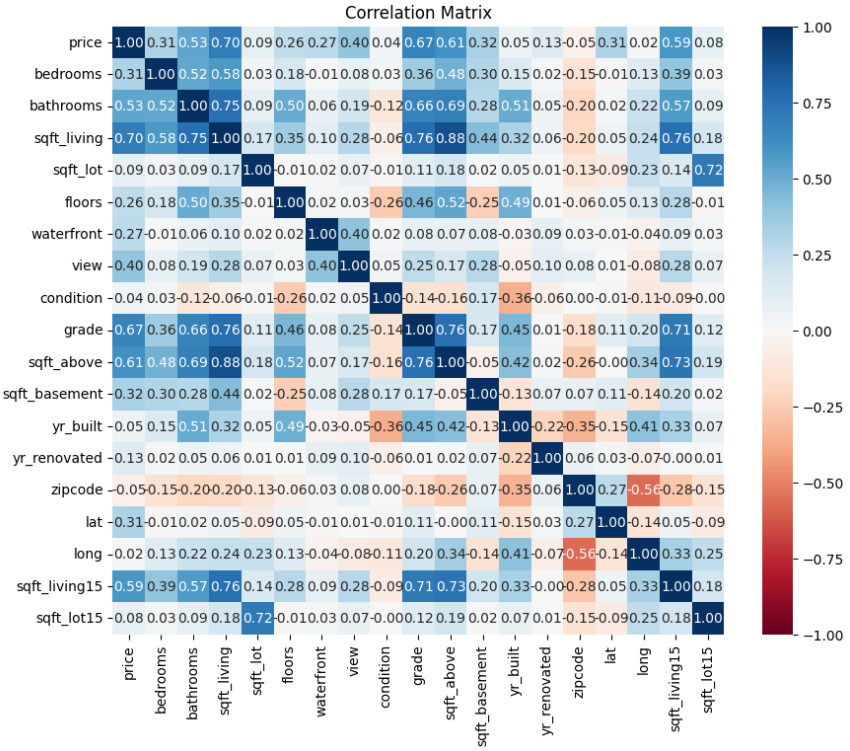
شکل 4-1. نمونه‌ای از دیتاست

```
1  data.isna().sum()
✓ 0.0s

date             0
price            0
bedrooms         0
bathrooms        0
sqft_living      0
sqft_lot         0
floors           0
waterfront       0
view             0
condition        0
grade            0
sqft_above       0
sqft_basement    0
yr_built         0
yr_renovated     0
zipcode          0
lat              0
long             0
sqft_living15    0
sqft_lot15       0
dtype: int64
```

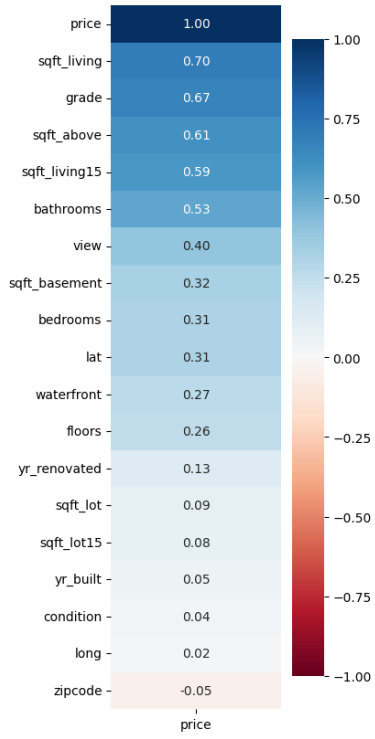
شکل 4-2. بررسی **Nan** های دیتاست

همانطور که می‌بینیم، دیتاست حاوی Nan نیست و هیچ missing value ای ندارد.

## ۴-۲. ماتریس همبستگی



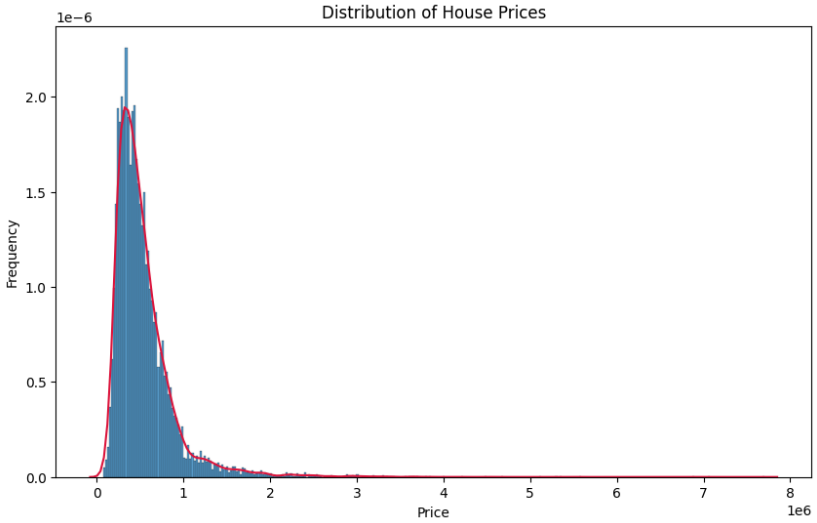

شکل 4-3. ماتریس همبستگی بین تمامی ویژگی‌ها



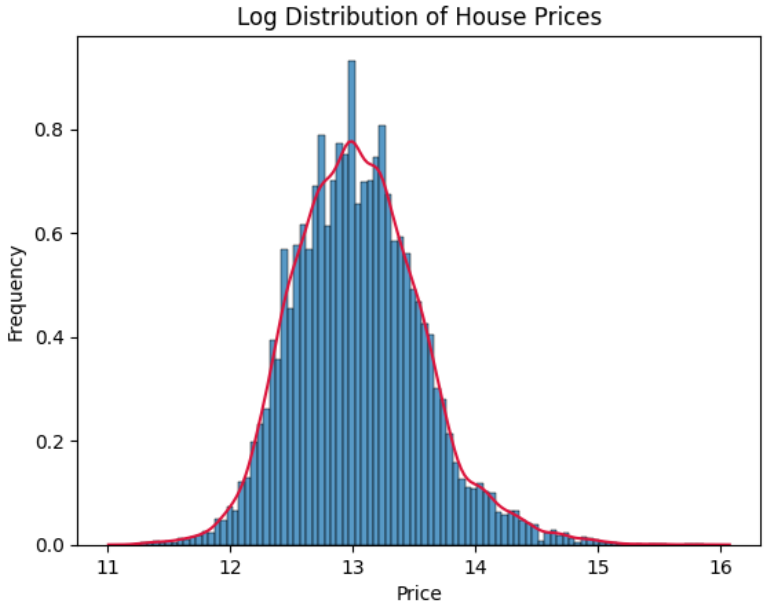

شکل 4-4. ماتریس همبستگی ویژگی‌ها با price

مشاهده می‌شود که sqft_living، grade و sqft_above بیشترین همبستگی با price را دارند.

## ۴-۳. رسم نمودار

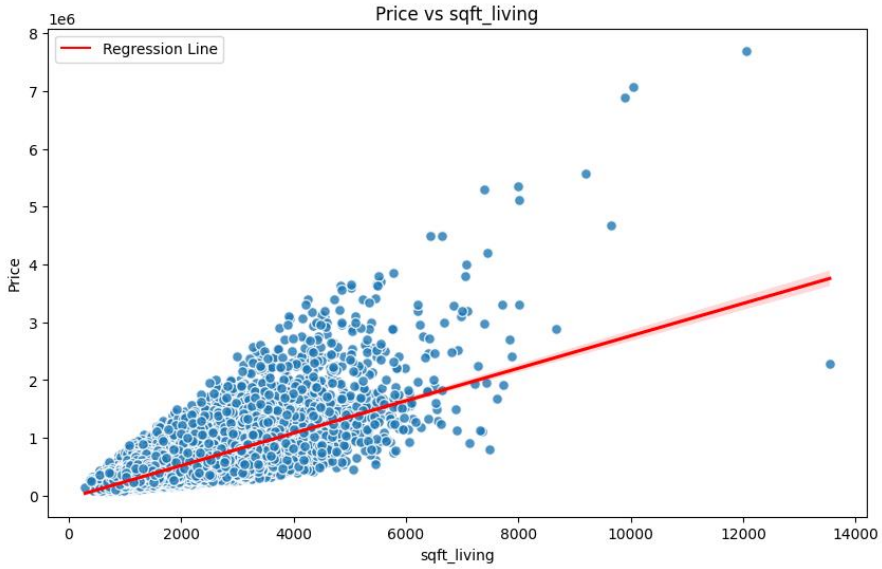



شکل 4-5. نمودار توزیع قیمت




شکل 4-6. نمودار لگاریتمی توزیع قیمت

معمول است که نمودار قیمت right skewd باشد. به همین دلیل log transformed قیمت را هم رسم می‌کنیم تا شهود بهتری از توزیع قیمت داشته باشیم و می‌بینیم که تقریبا در یک توزیع شبه نرمال قیمت‌ها توزیع شده اند.




شکل 4-7. نمودار قیمت و sqft_living

ویژگی sqft_living بیشترین همبستگی با قیمت را دارد که نمودار آن به همراه regression line مربوطه رسم شده است. یکی از کاربردهای این نمودار، پیدا کردن outlier ها است.

## ۴-۴. پیش‌پردازش داده

```
data['date'] = pd.to_datetime(data['date'], errors='coerce',
format='%Y%m%dT%H%M%S')

data['year'] = data['date'].dt.year
data['month'] = data['date'].dt.month

data = data.drop(columns=['date'])
```

با استفاده از این کد، ستون date که به فرمت مشخص شده است را به دو ستون year و month تبدیل و date را حذف میکنیم.

سپس دیتا را با ratio = 0.25 به دو دسته train و validation تقسیم میکنیم. دیتاست validation شامل داده‌هایی است که مدل در طول فرایند learn نمی‌بیند و برای تنظیم هایپرپارامترها و جلوگیری از overfitting به کار می‌رود. پس از آموزش مدل با داده‌های train، دیتاست validation به مدل کمک می‌کند تا عملکرد آن روی داده‌های جدید ارزیابی و تنظیم شود.

در آخر دیتای train و validation را به طور جداگانه برای جلوگیری از Data Leakage که در قبل توضیح دادیم، با استفاده از scale MinMaxScaling می‌کنیم. فرمول این متد در زیر آمده است.

$$X' = \frac{X - X_{\min}}{X_{\max} - X_{\min}}$$

## ۴-۵. پیاده‌سازی مدل

```
class HousePriceMLP(nn.Module):
    def __init__(self, input_size, hidden_layer_sizes):
        super(HousePriceMLP, self).__init__()
        layers = []
        previous_size = input_size
        for hidden_size in hidden_layer_sizes:
            layers.append(nn.Linear(previous_size, hidden_size))
            layers.append(nn.ReLU())
            previous_size = hidden_size
        layers.append(nn.Linear(previous_size, 1))
        self.model = nn.Sequential(*layers)

    def forward(self, x):
        return self.model(x).view(-1, 1)
```

کلاس HousePriceMLP یک مدل MLP است که به کمک PyTorch پیاده‌سازی شده است. این مدل در ابتدا ورودی‌ها را دریافت کرده و آن‌ها را از طریق چندین لایه پنهان (که هرکدام یک لایه خطی به همراه تابع فعال‌سازی ReLU دارند) عبور می‌دهد. در نهایت، خروجی مدل به یک مقدار واحد (پیش‌بینی قیمت) تبدیل می‌شود. توجه کنید که از hidden_layer_sizes برای پیاده‌سازی مدل با تعداد لایه های پنهان مختلف استفاده میکنیم. تابع *forward* مسیر داده‌ها از لایه‌ها تا خروجی نهایی را تعریف می‌کند.

## ۴-۶. آموزش مدل

برای اینکه بتوانیم مدل را به بهترین شکل آموزش دهیم، نیاز داریم تا هایپرپارامترهای مورد نیاز را پیدا کنیم. ما سه پارامتر اصلی که به دنبال بهترین ترکیب آنها بودیم را انتخاب کردیم: learnig rate, hidden_layer_config, lambda. منظور از lambda، همان weight_decay برای adam است. برای هر ترکیب از این هایپرپارامترها، مدل ایجاد شده، آموزش داده می‌شود و عملکرد آن با استفاده از معیارهای MAE و RMSE ارزیابی می‌شود.

**ابتدا برای مدل با یک لایه پنهان این را انجام می‌دهیم.**

جدول 4-1. هایپرپارامترها و عملکرد آنها در مدل MLP با **یک** لایه پنهان

| learning_rate | hidden_layers | lambda | MAE | RMSE |
|---|---|---|---|---|
| **0.05** | [256] | 0.0001 | 111814.2 | 178766 |
| **0.05** | [256] | 0.001 | 112342.8 | 179283.5 |
| **0.05** | [128] | 0.001 | 118602 | 191666.3 |

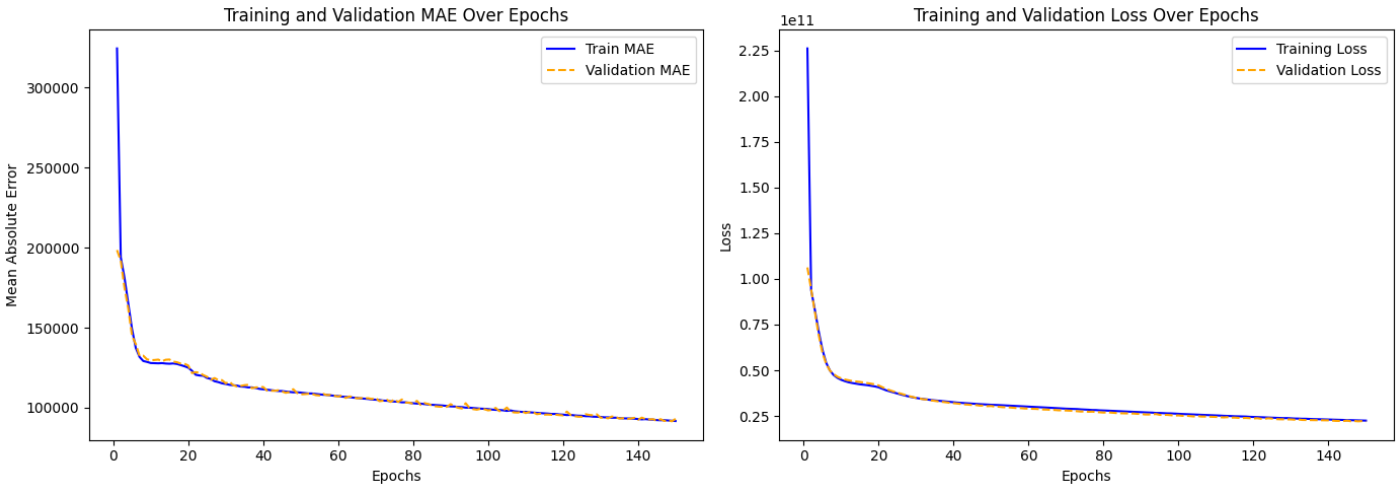
| | | | | |
|---|---|---|---|---|
| **0.05** | [128] | 0.0001 | 117992.6 | 193139.3 |
| **0.05** | [64] | 0.001 | 121484.7 | 201323.2 |
| **0.05** | [64] | 0.0001 | 127657.4 | 205865.8 |
| **0.01** | [256] | 0.001 | 129408.9 | 214294.9 |
| **0.01** | [256] | 0.0001 | 130313.7 | 214717.9 |
| **0.01** | [128] | 0.0001 | 133176.1 | 225272.5 |
| **0.01** | [128] | 0.001 | 133017.4 | 225586.8 |
| **0.01** | [64] | 0.0001 | 145287.8 | 245121.3 |
| **0.01** | [64] | 0.001 | 147344.2 | 247967.4 |

جدول براساسRMSE مرتب شده است و بهترین هایپرپارامتر پیدا شده، همان ردیف اول است.

برای optimizer از adam استفاده می‌کنیم. این optimizer به دلیل ترکیب ویژگی‌های سرعت همگرایی، تنظیم خودکار نرخ یادگیری، و مقاومت در برابر نوسانات، یکی از محبوب‌ترین و پرکاربردترین الگوریتم‌های بهینه‌سازی در یادگیری عمیق شناخته می‌شود.

همچنین برای loss function از معیار های MAE و RMSE استفاده می‌کنیم. MAE میانگین قدر مطلق تفاوت پیش‌بینی‌ها و مقادیر واقعی است. این معیار خطای مطلق را اندازه‌گیری می‌کند.

حال مدل را با استفاده از این هایپرپارامترها بر روی دیتا، آموزش می‌دهیم که نتایج بر روی دیتای train و validation در پایین آمده است.




شکل 4-8. نمودار خطای MAE و RMSE در طی آموزش مدل MLP با **یک** لایه پنهان

**حال برای MLP با دو لایه پنهان همین مراحل را انجام می‌دهیم.**

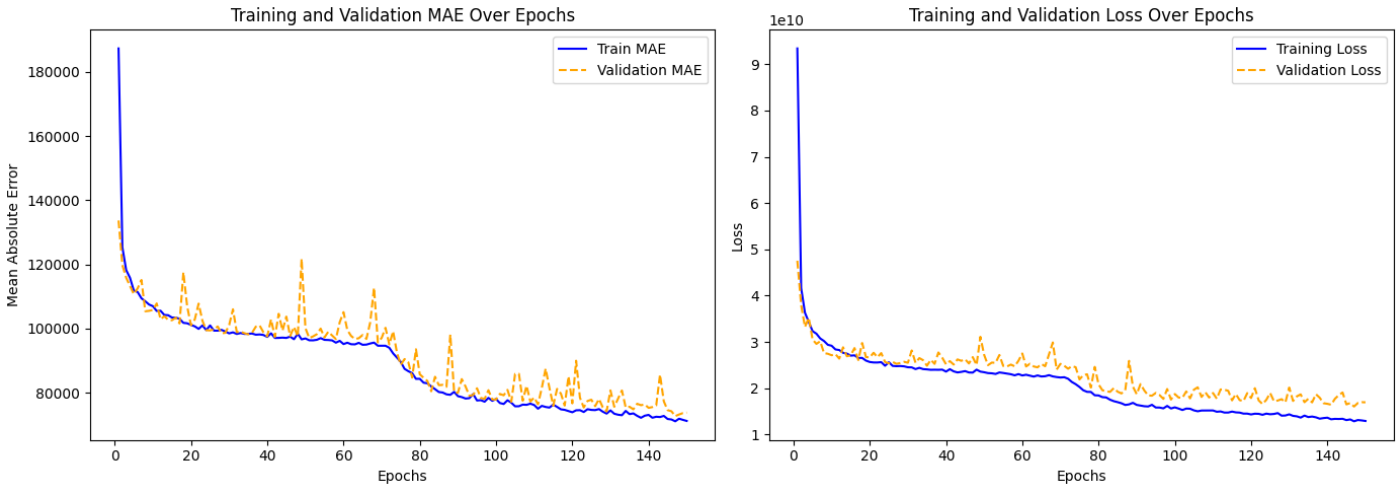
ابتدا باید هایپرپارامترها را پیدا کنیم.

جدول 4-2. هایپرپارامترها و عملکرد آنها در مدل MLP با **دو** لایه پنهان

| learning_rate | hidden_layers | lambda | MAE | RMSE |
|---|---|---|---|---|
| **0.05** | [64, 128] | 0.001 | 96449.28 | 155777 |
| **0.01** | [128, 256] | 0.001 | 100428.7 | 161249.5 |
| **0.01** | [64, 128] | 0.001 | 103326.1 | 162225.9 |
| **0.05** | [128, 256] | 0.001 | 108826.3 | 162262.3 |
| **0.01** | [64, 128] | 0.0001 | 103068.3 | 162676.1 |
| **0.05** | [64, 128] | 0.0001 | 99165.16 | 163016.3 |
| **0.05** | [128, 256] | 0.0001 | 98385.71 | 164304.6 |
| **0.01** | [128, 256] | 0.0001 | 112113.9 | 168995.9 |

جدول براساسRMSE مرتب شده است و بهترین هایپرپارامتر پیدا شده، همان ردیف اول است.

برای optimizer از adam استفاده می‌کنیم و loss function دوباره همان MAE و RMSE است.




شکل 4-8. نمودار خطای MAE و RMSE در طی آموزش مدل MLP با **دو** لایه پنهان

## ۴-۷. تحلیل نتایج

**مدل با یک لایه پنهان**:

- هر دو خطای MAE و RMSE برای داده‌های train و validation به طور پیوسته کاهش یافته و با افزایش تعداد epochها تثبیت می‌شوند.
- کاهش خطا یکنواخت بوده و نوسان کمی مشاهده می‌شود که نشان‌دهنده فرآیند آموزش پایدار است.
- Validation loss بسیار نزدیک به training loss است که نشان‌دهنده عملکرد خوب مدل در generalize کردن آموزش است.
- بهترین تعداد epoch برای این مدل حدود **60 تا 80** است، جایی که هر دو خطا ثابت می‌شوند.

**مدل با دو لایه پنهان**:

- خطای MAE و RMSE برای داده‌های train به طور پیوسته کاهش می‌یابند، اما خطاهای validation نوسانات زیادی نشان می‌دهند.
- Validation loss دارای افزایش و کاهش ناگهانی است که می‌تواند ناشی overfitting باشد یا حساسیت مدل به داده‌های آموزش را نشان دهد.
- Validation loss در اکثر مواقع بالاتر از training loss است و پایداری کمتری دارد.
- با وجود نوسانات، خطاها پس از حدود **100 تا 120** epoch تثبیت می‌شوند، بنابراین این بازه می‌تواند زمان مناسبی برای توقف آموزش باشد.

**تفاوت در عملکرد**:

- **مدل با یک لایه پنهان** پایداری بیشتری دارد و همگرایی یکنواخت‌تری نشان می‌دهد، در حالی که **مدل با دو لایه پنهان** نوسانات بیشتری به خصوص در معیارهای validation دارد.
- نوسانات بیشتر در **مدل با دو لایه پنهان** می‌تواند ناشی از موارد مختلفی باشد. اولین مورد overfitting است. **مدل با دو لایه پنهان** پیچیده‌تر است و ممکن است به جای یادگیری الگوهای مهم، نویز موجود در دیتای train را نیز یاد بگیرد که عملکرد مدل روی دیتای validation را تضعیف کند.

- همچنین شبکه‌های عمیق‌تر ممکن است با مشکلاتی مانند گیر افتادن در local minimum یا مشکلات مربوط به گرادیان مواجه شوند.

در آخر 5 نمونه از دیتای validation را انتخاب و با استفاده از مدل **یک** لایه، قیمت آنها را پیش‌بینی می‌کنیم.

جدول 4-3. نتیجه اجرای مدل بر روی چند نمونه تصادفی

| Sample | Prediction | Actual |
|---|---|---|
| **1799** | 773610.9 | 788000 |
| **4166** | 369904.5 | 295000 |
| **3936** | 354933.6 | 375000 |
| **4266** | 419953.1 | 420000 |
| **4575** | 614563.1 | 588000 |