به نام خالق رنگین کمان

ستاره باباجانی-99521109-تمرین دوم

سوال 1: الف)

Overfitting در شبکه‌های عصبی وقتی رخ می‌دهد که مدل به داده‌های آموزشی خود خیلی خوب عادت کرده باشد و به‌طور غیرمنطقی به جزئیات کوچک آن داده‌ها پاسخ دهد. (جزئیات غیرضروری داده های آموزشی را یادگرفته باشد.) به عبارت دیگر، مدل به جای یادگیری الگوهای کلی و عمومی، اطلاعات خاص به داده‌های آموزشی را حفظ کرده و در مقابل داده‌های جدید(داده های تست) به خوبی عمل نمی‌کند. برای حل این مشکل میتوان از data ugmentation, dropout, regularization استفاده کرد.

برعکس،Underfitting زمانی رخ می‌دهد که مدل به ندرت حتی الگوهای ساده‌تر هم در داده‌های آموزشی یاد گرفته باشد و در نتیجه نتواسته با دقت به داده‌های آموزشی و همچنین داده های تازه وارد شده پاسخ دهد. برای حل این مشکل میتوان تعداد لایه های میانی یا نورون های آنها را زیاد کرد و یا تعداد ایپاک یادگیری مدل را بیشتر کرد.

ب) عملکرد مدل روی داده های تست و آموزشی اگر دچار بیش برازش شده باشد خیلی قابل توجه است. از طریق انجام چند کار میتوان متوجه این تفاوت شد:

1. مشاهده نمودارها: با رسم نمودارهای آموزش و اعتبارسنجی (validation)، اگر دقت مدل در داده‌های آموزش بالاست، اما در داده‌های اعتبارسنجی پایین، احتمالاً برازش بیش از حد رخ داده است.
2. مشاهده تغییرات دقت: زمانی که مدل بیش از حد برازش دارد، دقت در داده‌های آموزش به مرور زمان افزایش پیدا می‌کند، اما دقت در داده‌های اعتبارسنجی ممکن است کاهش یابد.
3. استفاده از داده‌های اعتبارسنجی: جدا کردن یک بخش از داده‌ها به عنوان داده‌های اعتبارسنجی و استفاده از آن برای ارزیابی عملکرد مدل می‌تواند کمک کند. اگر دقت در داده‌های اعتبارسنجی پایینتر از داده‌های آموزش باشد، احتمالاً برازش بیش از حد رخ داده است.

پ) برای این سوال، از dropout معمولی استفاده میکنیم. برای نوشتن جدول تست، به p نیاز داریم که چون در dropout mask تعداد برابری از 0 و 1 وجود دارد، پس مقدار این متغیر 0.5 میشود.

حال برای یافتن خروجی حالت آموزش، باید ماتریس dropout mask را در ماتریس output لحاظ کنیم، یعنی هرجا 1 بود، مقدار خود خروجی و هرجا صفر بود، مقدار صفر قرار میگیرد. پس خروجی لایه در آموزش به این صورت میشود:

1.6	0	0	1.9
0	2.5	2.5	0
0	3.2	3.7	0
1.3	0	0	1.2

حال برای محاسبه ماتریس خروجی حالت تست، باید هرکدام از مقادیر ماتریس output را در p ضرب کنیم. پس خواهیم داشت:

0.8	-0.35	-0.1	0.95
-1.15	1.25	1.25	-0.45
-0.25	1.6	1.85	-0.2
0.65	-0.2	-1.3	0.6

سوال 2: الف)

الگوریتم نزدیک‌ترین همسایگی یک الگوریتم یادگیری ماشین مستند به داده Instance-Based Learning است که بر اساس فاصله بین نمونه‌های آموزشی در فضای ویژگی‌ها عمل می‌کند.

در این الگوریتم، برای تخمین یک مقدار یا کلاس برچسب برای نمونه‌های تست، k نزدیک‌ترین نمونه آموزشی به نمونه تست انتخاب می‌شوند و نتیجه بر اساس اکثریت برچسب‌ها یا میانگین مقادیر آن‌ها مشخص می‌شود.

- تاثیر تغییر k:
  - با افزایش مقدار k تعداد نمونه‌هایی که در تخمین مقدار یا برچسب نمونه تست مشارکت می‌کنند افزایش پیدا می‌کند.
  - با افزایش k، الگوریتم به نوعی از اطلاعات بیشتری در مورد توزیع داده‌ها برخوردار می‌شود.
  - با افزایش k، تأثیر نویز در داده‌ها کاهش می‌یابد و الگوریتم به سمت مدل‌های ساده‌تر و مقاوم‌تر به نویز حرکت می‌کند. این باعث کاهش واریانس مدل می‌شود.
  - با افزایشk ، ممکن است مدل به شدت ساده شود و بایاس آن افزایش یابد.(مدل الگوهای پیچیده را یاد نمیگیرد.)

به طور کلی، افزایش k می‌تواند منجر به کاهش واریانس و افزایش بایاس شود.

ب)

1. استفاده از منظم سازی، ممکن است باعث تضعیف عملکرد مدل شود:

   گزاره درست است. منظم‌سازی به عنوان یک تکنیک مهم در پیشگیری از overfitting استفاده می‌شود.

تکنیک های منظم سازی، مانند منظم سازی L1 یا L2، یک عبارت جریمه به تابع هزینه مدل بر اساس بزرگی وزن ها اضافه میکنند که این جریمه به کنترل پیچیدگی مدل کمک می‌کند و با جلوگیری از وزنه‌های بیش از حد بزرگ از برازش بیش از حد جلوگیری می‌کند.(عموما اینطور است.)

در حالی که منظم سازی بیش از حد ممکن است منجر به عدم تناسب (ساده سازی بیش از حد) و کاهش بالقوه عملکرد شود.(همچنین گاهی اگر داده خیلی پیچیده باشد، به وزن های بزرگ و پیچیده نیاز میشود که در این حالت منظم سازی باعث تضعیف عملکرد مدل میشود.)

2. اضافه کردن تعداد زیاد ویژگی های جدید، باعث جلوگیری از بیش برازش می شود:

گزاره غلط است زیرا بیش برازش زمانی اتفاق می‌افتد که یک مدل داده‌های آموزشی را به خوبی یاد می‌گیرد، از جمله نویز و نقاط پرت آن، تا حدی که در داده‌های جدید و دیده نشده ضعیف عمل می‌کند. افزودن ویژگی‌های بسیار زیاد، به‌ویژه موارد نامربوط یا پر نویز پیچیدگی مدل و احتمال تطبیق داده‌های آموزشی را افزایش می‌دهد.(خیلی مهم است چه ویژگی هایی اضافه میشوند. ویژگی ها باید الگوهای اساسی را نشان دهند در غیراینصورت بیش برازش بیشتر میشود.)

3. با زیاد کردن ضریب منظم سازی، احتمال بیش برازش بیشتر می شود:

گزاره غلط است زیرا ضریب منظم سازی، قدرت عبارت منظم سازی را در تابع هزینه مدل کنترل می کند و با افزایش ضریب تنظیم، جریمه وزنه های بزرگ نیز افزایش می یابد.

منظم‌سازی بیشتر مدل‌های پیچیده و وزن‌های بزرگ را ناامید می‌کند و مدل را به جای افزایش احتمال بیش برازش در برابر آن مقاوم‌تر می‌کند.(البته اگر این ضریب خیلی بزرگ شود میتواند باعث underfitting شود.)

پ)

- Wexp1

منظم کردن L2 باعث تشویق وزن های کوچک و نزدیک به هم می شود.(این وزنه ها نسبتاً کوچک و متعادل هستند که نشان دهنده عدم ترجیح هیچ ویژگی خاصی است.)

- Wexp2

این بردار وزن یک راه حل پراکنده با ترجیح قوی(1) برای یک ویژگی را پیشنهاد می کند.

تنظیم L1 با هدایت برخی از وزن ها تا دقیقاً صفر، پراکندگی را تشویق می کند. بنابراین، به احتمال زیاد از تنظیم L1 استفاده شده است.

- Wexp3

وزن ها بزرگ هستند و تفاوت قابل توجهی در بزرگی ها وجود دارد که نشان می دهد برخی از ویژگی ها از بقیه مهم تر هستند.(همچنین هیچ کدام صفر نیستند.) پس احتمالا از هیچ کدام از تنظیم ها استفاده نشده است.

- Wexp4

مشابه Wexp3، تفاوت قابل توجهی در اندازه ها وجود دارد، اما وزن صفر نیز وجود دارد که نشان دهنده درجه ای از پراکندگی است.

این ترکیب وزن های بزرگ و پراکندگی میتواند نشان دهنده استفاده از تنظیم L1 باشد(اگر L2 میبود، مقادیر کوچیک و نزدیک به هم باید میشدند و با احتمال کمتری صفر وجود داشت.)

سوال 3: الف)

تقطیر دانش فرآیندی در یادگیری عمیق است که در آن یک مدل کوچکتر و فشرده تر(دانش آموز) برای تکرار رفتار یک مدل بزرگتر و پیچیده تر(معلم) آموزش می بیند. هدف اولیه از تقطیر دانش، انتقال دانش یا اطلاعات کدگذاری شده در مدل معلم به مدل دانش آموز کوچکتر است، که به مدل دانش آموز اجازه می دهد تا عملکرد مشابه یا حتی برتر را داشته باشد.

فرآیند کلی تقطیر دانش شامل مراحل زیر است:

1. آموزش الگوی معلم: یک مدل (معلم) بزرگتر و پیچیده تر در مورد یک کار خاص آموزش داده میشود.
   که مدل معلم معمولا عمیق تر و از نظر محاسباتی گران تر از مدل کوچکتر مورد نظر است.
2. نسل هدف نرم: به جای برچسب های سخت، اهداف نرم یا توزیع احتمال را از مدل معلم ایجاد میکنیم. اهداف نرم اطلاعات بیشتری در مورد روابط بین کلاس ها ارائه می دهند و به هدایت فرآیند یادگیری مدل دانش آموز کمک می کنند.
3. آموزش الگوی دانشجویی: یک مدل کوچکتر (دانش آموز) را برای همان کار با استفاده از برچسب های سخت اصلی و اهداف نرم تولید شده توسط مدل معلم آموزش میدهیم.
   که الگوی دانش آموز برای تقلید از رفتار الگوی معلم و تعمیم دانش آن طراحی شده است.
4. تابع هدف: تابع ضرر استفاده شده در طول آموزش شامل اصطلاحاتی است که تفاوت بین پیش بینی های مدل دانش آموز و برچسب های سخت و اهداف نرم ارائه شده توسط مدل معلم را جریمه می کند.

مزایا:

- فشرده‌سازی مدل: مدل دانش‌آموز معمولاً کوچک‌تر و از نظر محاسباتی کارآمدتر از مدل معلم است و آن را برای استقرار در محیط‌های محدود به منابع استفاده میکنند.
- تعمیم بهبود یافته: انتقال دانش از مدل معلم به الگوی دانش آموز کمک می کند حتی با مجموعه داده کوچکتر تعمیم بهتری داشته باشد.
- فعال کردن یادگیری انتقالی: مدل دانش‌آموز می‌تواند دانش را از یک مدل معلم از قبل آموزش دیده به ارث ببرد و امکان همگرایی سریع‌تر و عملکرد بهتر در وظایف مرتبط را فراهم کند.

تقطیر دانش با موفقیت در حوزه‌های مختلف، از جمله طبقه‌بندی تصویر، پردازش زبان طبیعی، و تشخیص گفتار و سایر موارد به کار گرفته شده است. این روشی را برای ایجاد تعادل بین اندازه مدل و عملکرد فراهم می کند و مدل های یادگیری عمیق را برای کاربردهای دنیای واقعی کاربردی تر می کند.

منبع:

https://blog.roboflow.com/what-is-knowledge-distillation/#:~:text=Knowledge%20distillation%20is%20a%20powerful,language%20processing%2C%20and%20speech%20recognition.

ب) فرآیند یادگیری در معماری تقطیر دانش داده شده شامل آموزش مدل معلم (با m لایه) و مدل دانش آموز (با n لایه) برای انتقال دانش از معلم به دانش آموز است. توضیح گام به گام فرآیند به این صورت است:

1. آموزش الگوی معلم: خروجی لایه mام با استفاده از تابع softmax با دمای T=t پردازش می شود. این برچسب‌های نرمی که تولید میشوند نشان دهنده دانش معلم در مورد داده‌های ورودی هستند. این برچسب‌های نرم، عدم قطعیت و روابط بین طبقات مختلف را نشان می‌دهند.

2. تولید لیبل نرم: از برچسب های نرم تولید شده توسط مدل معلم به عنوان اهداف نرم برای آموزش مدل دانش آموز استفاده میشود.

3. آموزش نمونه دانش آموز: آموزش مدل دانش آموز با n لایه با استفاده از ترکیب برچسب های سخت (ground truth) و اهداف نرم تولید شده توسط مدل معلم انجام میشود.

خروجی لایه n از مدل دانشجویی به دو صورت استفاده می شود:

- برای تولید پیش بینی های نرم: یک تابع softmax با دمای T=t اعمال میشود. این پیش‌بینی‌های نرم با برچسب‌های نرم (هدف‌های نرم) با استفاده از یک تابع از ضرر مقایسه می‌شوند که از آن به عنوان fn loss یاد می‌شود.
- از یک تابع softmax با دمای T=1 برای تولید پیش بینی های سخت استفاده میشود. این پیش‌بینی‌های سخت با برچسب‌های سخت با

استفاده از تابع ضرر مقایسه می‌شوند که واگرایی بین پیش‌بینی‌های سخت دانش‌آموز و برچسب‌های واقعی را اندازه‌گیری می‌کند.

5. عملکرد تابع ضرر: تابع ضرر کلی برای آموزش مدل دانش آموز ترکیبی از ضرر تقطیر (برچسب نرم و پیش بینی نرم) و ضرر دانش آموز (برچسب سخت و پیش بینی سخت) است.

تابع ضرر مقطر، مدل دانش آموز را تشویق می کند تا از پیش بینی های نرم مدل معلم تقلید کند و روابط ظریف بین کلاس ها را به تصویر بکشد.

تابع ضرر دوم تضمین می کند که مدل دانش آموز یاد می گیرد که پیش بینی های سخت دقیق را انجام دهد و با برچسب های واقعی همسو شود.

پ) وزن های مدل دانش آموز بر اساس یک تابع ضرر ترکیبی(ترکیب دو تابع ضرر ذکر شده) آپدیت میشوند:

$$L = \lambda \,.\, distillation\_Loss + (1 - \lambda) \,.\, student\_Loss$$

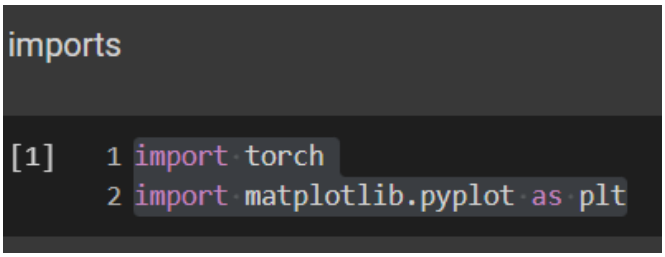
که در اینجا لاندا یک ابرپارامتر است.

منبع:

https://www.chat.openai.com/

سوال 4: توضیح کدها و خروجی های نوتبوک به شرح زیر است:

- ابتدا کتابخانه مورد نیاز صدا زده میشود. این کتابخانه ها شامل torch, matplotlib هستند:



- سپس یک کلاس به نام model ایجاد کردیم که یک مدل شبکه عصبی ساده را پیاده‌سازی می‌کند و قابلیت آموزش با چندین نرخ یادگیری مختلف و الگوریتم بهینه‌سازی را دارد. این مدل شامل دو لایه است و یک لایه مخفی است.

1. تابع __init__: این متد مقادیر اولیه متغیرهای کلاس را مقداردهی می‌کند. این شامل مقادیر x و y (ورودی و خروجی مدل) و همچنین فراخوانی توابع _initalize_parameters، _initalize_moms و _initalize_RMSs برای به ترتیب مقداردهی اولیه وزن‌ها و بایاس ها، اندازه‌گیری‌های momentum و RMS (Root Mean Square) است.

Momentum اطلاعاتی از تغییرات گذشته در وزن‌های شبکه عصبی نگه می‌دارد. این اطلاعات مشابه سرعت و جهت حرکت هستند که از آنها برای افزایش سرعت آموزش و پیشگیری از گیر

کردن در مینیمم‌های محلی استفاده می‌شود. در تابع ذکر شده، اندازه‌گیری‌های moms برای هر یک از وزن‌ها و بایاس‌ها به مقدار صفر مقداردهی شده اند.

RMS اطلاعاتی از تغییرات گذشته در وزن‌های شبکه عصبی را نگه می‌دارد. این اطلاعات نشان‌دهنده میزان تغییرات وارده در وزن‌ها هستند که از آنها برای تنظیم نرخ یادگیری به شکل خودکار و بهبود کیفیت آموزش استفاده می‌شود. در تابع ذکر شده ، اندازه‌گیری‌های RMS برای هر یک از وزن‌ها و بایاس‌ها به صفر مقداردهی شده اند.

2. تابع random_tensor_: این تابع یک تنسور تصادفی با ابعاد داده شده ایجاد می‌کند و requires_grad_ را فراخوانی می‌کند تا تنسور بتواند به عنوان وزن مدل بهینه‌سازی شود.

3. تابع nn_: این تابع شبکه عصبی را با ورودی xb پیاده‌سازی می‌کند. این شبکه دو لایه دارد و برای هر لایه ماتریس وزن و بایاس مربوطه را استفاده می‌کند. ابتدا ورودی به لایه اول اعمال شده و سپس خروجی لایه اول به لایه دوم منتقل می‌شود.

4. تابع loss_func_: این تابع معادله تابع هزینه (Mean Squared Error) را محاسبه می‌کند.

5. تابع train: این تابع برای آموزش مدل از چند نرخ یادگیری مختلف استفاده میکند. چندین نرخ یادگیری مختلف (lrs) را امتحان میکند و نتایج آموزش را برای هر نرخ یادگیری در نمودارهای جداگانه نشان می‌دهد. آموزش ادامه می‌یابد تا زمانی که ضرر بیشتر از 0.1 باشد یا تعداد قدم‌ها به 1000 برسد. پس از اتمام هر نرخ یادگیری، مدل به حالت اولیه با مقادیر تصادفی برگشته و آماده آموزش برای نرخ یادگیری بعدی می‌شود.

طول لیست losses در ابتدا برابر با صفر است و در هر مرحله از حلقه:

a) پیش‌بینی‌ها با فراخوانی nn_ برای ورودی self.x محاسبه می‌شوند.
b) مقدار تابع هزینه با استفاده از loss_func_ برای پیش‌بینی‌ها و مقادیر واقعی self.y محاسبه می‌شود.
c) مشتقات تابع هزینه نسبت به وزن‌ها و بایاس‌ها با استفاده از loss.backward() محاسبه و به‌روزرسانی می‌شوند.

d)سپس optimizer با فراخوانی برای هر وزن و بایاس با نرخ یادگیری مشخص lr و اندازه‌گیری‌های momentum و RMS متناظر، بهینه‌سازی انجام می‌دهد.

e)در نهایت مقدار خطا به لیست losses اضافه می‌شود.

هدف از این حلقه آموزش مدل با مختلف نرخ‌های یادگیری و مشاهده نمودار تغییرات خطای آموزش در طول زمان برای هر یک از این نرخ‌های یادگیری است که این کار امکان این را فراهم می‌کند تا بهترین نرخ یادگیری برای مدل را انتخاب کنیم و تاثیر آن را روی سرعت و کیفیت آموزش مدل مشاهده کنیم.

این تابع همچنین از توابع matplotlib برای نمایش نمودارها و مقایسه نتایج با استفاده از مقادیر مختلف نرخ یادگیری استفاده می‌کند.

```python
class model:
    def __init__(self, x, y):
        self.x = x
        self.y = y

        self._initalize_parameters()
        self._initalize_moms()
        self._initalize_RMSs()

    def _initalize_parameters(self):
        self.weights_1 = self._random_tensor((x.shape[1],3))
        self.bias_1 = self._random_tensor(1)
        self.weights_2 = self._random_tensor((3,1))
        self.bias_2 = self._random_tensor(1)

    def _random_tensor(self, size): return (torch.randn(size)).requires_grad_()

    def _initalize_moms(self):
        self.moms_w1, self.moms_b1 = [0], [0]
        self.moms_w2, self.moms_b2 = [0], [0]

    def _initalize_RMSs(self):
        self.RMSs_w1, self.RMSs_b1 = [0], [0]
        self.RMSs_w2, self.RMSs_b2 = [0], [0]
    def _nn(self, xb):
        l1 = xb @ self.weights_1 + self.bias_1
        l2 = l1.max(torch.tensor(0.0))
        l3 = l2 @ self.weights_2 + self.bias_2
        return l3
```

```python
    def _loss_func(self, preds, yb):
        return ((preds-yb)**2).mean()

    def train(self, optimizer):
        # Multiple learning rates to see how optimizers work with them
        lrs = [10E-4,10E-3,10E-2,10E-1]
        ## for plotting ##
        fig, axs = plt.subplots(2,2)
        ## for plotting ##
        all_losses = []
        for i, lr in enumerate(lrs):
            losses = []
            while(len(losses) == 0 or losses[-1] > 0.1 and len(losses) < 1000):
                preds = self._nn(self.x)
                loss = self._loss_func(preds, self.y)
                loss.backward()
                optimizer(self.weights_1, lr, self.moms_w1, self.RMSs_w1)
                optimizer(self.bias_1, lr, self.moms_b1, self.RMSs_b1)
                optimizer(self.weights_2, lr, self.moms_w2, self.RMSs_w2)
                optimizer(self.bias_2, lr, self.moms_b2, self.RMSs_b2)
                losses.append(loss.item())
            all_losses.append(losses)

            ## for plotting ##
            xi = i%2
            yi = int(i/2)
            axs[xi,yi].plot(list(range(len(losses))), losses)
            axs[xi,yi].set_ylim(0, 30)
            axs[xi,yi].set_title('Learing Rate: '+str(lr))
            ## for plotting ##
```

```
            # Setting seed makes sure the parameters are initalized the same way for better comparison
            torch.manual_seed(42)
            self._initalize_parameters()
            self._initalize_moms()
            self._initalize_RMSs()

        ## for plotting ##
        for ax in axs.flat:
            ax.set(xlabel='steps', ylabel='loss (MSE)')
        plt.tight_layout()
        ## for plotting ##
```

- تابع generate_fake_labels برای تولید مقادیر مصنوعی برای مدل‌های آموزش داده‌شده است. مقادیر ورودی x3، x2 و x1 باید به عنوان ورودی‌های واقعی مدل (مثلاً ویژگی‌ها) برای مسائل واقعی از داده‌های واقعی تعیین شوند.

This simple function is implemented to generate y values from x values.

```
[6]  1 def generate_fake_labels(x3, x2, x1):
     2     return (x3**3 * 0.8) + (x2**2 * 0.1) + (x1 * 0.5) + 4.
```

- در سلول بعدی ابتدا داده‌های ورودی x و مقادیر برچسب متناظر با آن‌ها در y تعریف شده‌اند. x یک تنسور به ابعاد 3*5 است که در هر ردیف آن داده‌های ورودی ویژگی‌ها برای مدل هستند. y نیز یک تنسور با 5 عنصر است که مقادیر برچسب‌ها برای داده‌های ورودی متناظر با هر ردیف از x را نشان می‌دهد.

سپس با استفاده از یک حلقه for برای هر ردیف از x، تابع generate_fake_labels به عنوان یک تابع ریاضی برای هر مجموعه از ویژگی‌ها فراخوانی می‌شود تا مقدار برچسب متناظر با آن ویژگی‌ها محاسبه شود.

```
x = torch.tensor([[0.7,0.3,0.7],
                  [0.4, 1., 0.4],
                  [0.2, 1.1, 0.1],
                  [0.4, 0.7, 0.2],
                  [0.1, 0.5, 0.3]])
y = torch.tensor([generate_fake_labels(i[0],i[1],i[2]) for i in x])
print(x.shape, y.shape, y)
```

```
torch.Size([5, 3]) torch.Size([5]) tensor([4.6334, 4.3512, 4.1774, 4.2002, 4.1758])
```

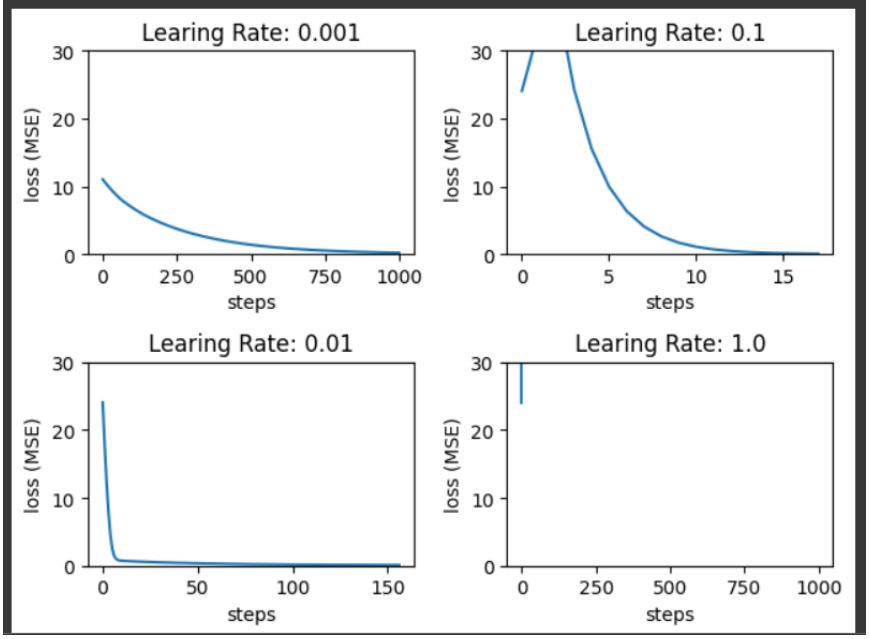
- در دو سلول بعدی، ابتدا مقادیر y در اینجا به صورت دستی تعریف شده‌اند و سپس مدل طراحی شده، صدا زده میشود.

```
[8]   y = torch.tensor([4.6334, 4.3512, 4.1774, 4.2002, 4.1758])

[9]   my_model = model(x, y)
```

- در سلول های بعدی از دو optimizer مختلف استفاده شده است که اولی با استفاده از SGD و دومی با استفاده از Momentum است. دومی توانایی دریافت تغییرات گذشته در مشتق وزن‌ها را دارد و از آن برای تنظیم نرخ یادگیری در مسیر مناسب استفاده می‌کند. این بهینه‌ساز معمولاً به سرعت آموزش و پیشگیری از گیر کردن در مینیمم محلی و نقطه زینی کمک می‌کند.

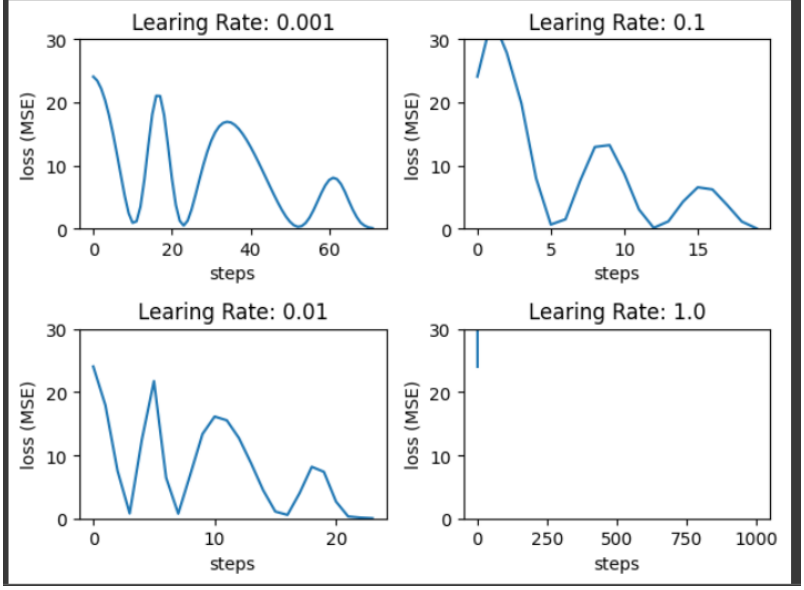
```python
def SGD(a, lr, _, __):
    a.data -= a.grad * lr
    a.grad = None
```

```python
#train model with SGD
my_model.train(SGD)
```




```python
def momentum(a, lr, moms, __):
    previous_momentum = moms[-1]

    mom = a.grad * lr + previous_momentum * (1 - lr)
    moms.append(mom)
    a.data -= mom
    a.grad = None
```

```python
#train model with momentum
my_model.train(momentum)
```



همان طور که از نمودار ها مشاهده میشود، بهینه ساز دوم اغلب در تعداد step خیلی خیلی کمتر توانسته به ضرر صفر برسد. بطور مثال، در کمترین نرخ یادگیری، برای اولین بهینه ساز 1000 قدم طول کشید ولی برای دومی حدود 80 قدم! (زیرا همانطور که مشاهده میشود مقادیر ضرر افت و خیز زیادی داشته تا از مینیمم های محلی رد شود!)

در نرخ یادگیری بعدی(0.01) بهینه ساز اول در حدود تعداد قدم 50 خیلی به ضرر 0 نزدیک شده است ولی احتمالا هی در اطراف آن در حال گردش بوده تا در قدم بیشتر از 150 به صفر برسد ولی بهینه ساز دوم با گذر از مینیمم های محلی متعدد و فراز و نشیب در ضرر، توانسته با حدود 30 قدم به ضرر صفر برسد.

در نرخ یادگیری 0.1 تعداد قدم دو تابع تقریبا مشابه است ولی دومین بهینه ساز فراز و نشیب بیشتری دارد.

ولی نرخ یادگیری آخر (کوچکترین آن)، دو بهینه ساز تقریبا خیلی بد عمل کردند و به ضرر کمتر از حدودا 25 دست نیافته اند. علت آن احتمالا گیر کردن در مینیمم محلی باشد که بهینه ساز دوم هم حتی با استفاده از مشتق های پیشین، نتوانسته از آن عبور کند. برای رفع آن میتوان از بقیه توابع بهینه سازی استفاده کرد.

سوال 5: مراحل تعریف مدل خواسته شده و مقایسه با خروجی این چنین است:

- ابتدا کتابخانه های مورد نیاز صدا زده میشوند:

```
[1]  1 import torch
     2 import torchvision
     3 import torchvision.datasets as datasets
     4 import torchvision.transforms as transforms
     5 import torch.nn as nn
     6 import torch.nn.functional as F
     7 import torch.optim as optim
     8 import matplotlib
     9 import matplotlib.pyplot as plt
    10 import numpy as np
    11 import pandas as pd
    12 import random
    13 import math
    14 from torch.utils.data.sampler import SubsetRandomSampler
```

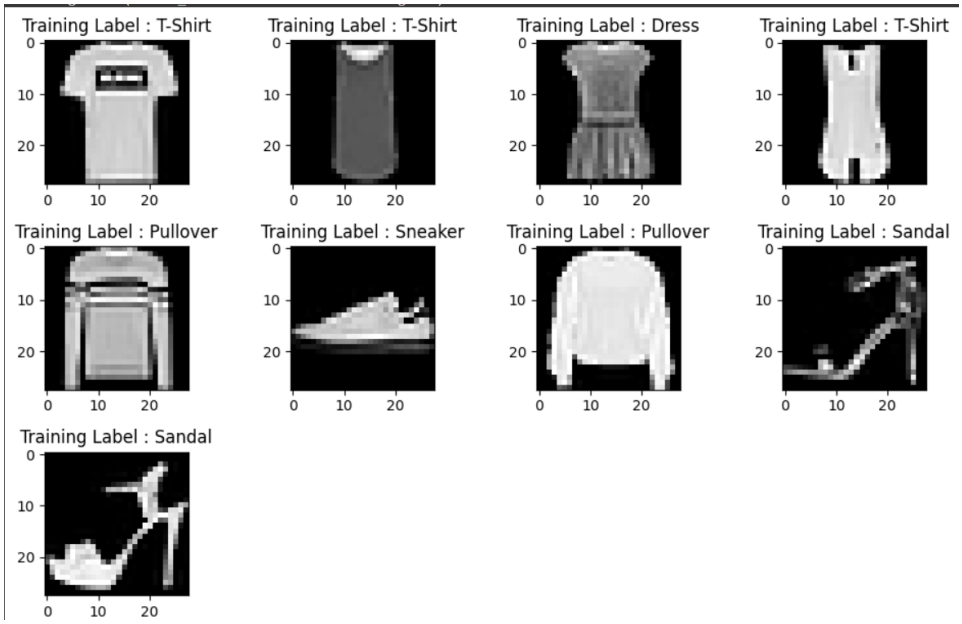
- سپس برای نرمال کردن از transform استفاده میکنیم و داده را برای آموزش و تست دانلود میکنیم:

```
# Define a transform to normalize the data
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))])

# Download and load training data
trainset = datasets.FashionMNIST('./data',download=True, train= True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size= 64, shuffle=True)

# Download and load test data
testset = datasets.FashionMNIST('./data',download=True, train= False, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size= 64, shuffle=True)
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz to ./data/FashionMNIST/raw/train-images-idx3-ubyte.gz
100%|██████████| 26421880/26421880 [00:01<00:00, 16876842.33it/s]
Extracting ./data/FashionMNIST/raw/train-images-idx3-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw/train-labels-idx1-ubyte.gz
100%|██████████| 29515/29515 [00:00<00:00, 272695.17it/s]
Extracting ./data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to ./data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz
100%|██████████| 4422102/4422102 [00:00<00:00, 5013699.58it/s]
Extracting ./data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz
100%|██████████| 5148/5148 [00:00<00:00, 17055511.05it/s]
Extracting ./data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw
```

- حال برای چند نمونه از داده آموزش، عکس و لیبل آن را نمایش میدهیم:

```python
labels_map = ('T-Shirt', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt','Sneaker', 'Bag', 'Ankle Boot')
fig = plt.figure(figsize=(10,10));
columns = 4;
rows = 5;
for i in range(1, 10):
    fig.add_subplot(rows, columns, i)
    fig.tight_layout()
    plt.imshow(trainset.train_data[i].numpy(), cmap='gray')
    plt.title('Training Label : %s' % labels_map[trainset.train_labels[i]])
plt.show()
```



- حال مدل را طراحی میکنیم. مدل طراحی شده یک مدل sequential است که ابتدا از Flatten استفاده کردیم تا یکنواخت شود و سپس 3 لایه اضافه کردیم(یک لایه ورودی، یک لایه میانی و یک لایه خروجی). همان طور که از عکس قابل مشاهده است، لایه اول تماما متصل است که اندازه ورودی 784(عکس های 28*28 پیکسلی) است که به 128 نورون در لایه میانی وصل شده و تابع فعال سازی آن RelU است. لایه میانی، 128 نورون ورودی دارد که تعداد را به 64 نورون کاهش میدهد.(تابع فعال سازی آن، مثل لایه ورودی است). در نهایت، لایه

خروجی 64 نورون دارد که آن را به 10 عدد که تعداد کلاس ها است کاهش میدهد.

```
input_size = 784
out_size = 10
```

```
## Define the model
################ Your code ############
model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(input_size, 128),
    nn.ReLU(),
    nn.Linear(128, 64),
    nn.ReLU(),
    nn.Linear(64, out_size)
)
#######################################
```

- در قدم بعدی، تابع ضرر و تابع بهینه ساز را طبق خواسته سوال تعریف میکنیم:

```
################ Your code ############
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)
#######################################
```

- حال مدل را پرینت میگیریم تا درک بهتری از لایه ها داشته باشیم:

```
print(model)
```

```
Sequential(
  (0): Flatten(start_dim=1, end_dim=-1)
  (1): Linear(in_features=784, out_features=128, bias=True)
  (2): ReLU()
  (3): Linear(in_features=128, out_features=64, bias=True)
  (4): ReLU()
  (5): Linear(in_features=64, out_features=10, bias=True)
)
```

- حال نوبت آموزش مدل است. تعداد ایپاک 10 است و نحوه آموزش آن بصورت گرادیان کاهشی است و ابتدا forward pass و در ادامه برای آپدیت کردن وزن ها backward استفاده شد. میزان ضرر در هر ایپاک نیز پرینت شده است:

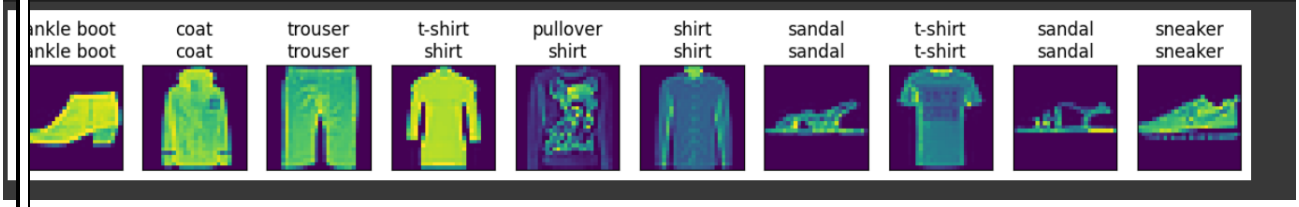
```python
## Train your model
epochs = 10

for e in range(epochs):
  running_loss = 0
  for images, labels in trainloader:

    images = images.view(images.shape[0],-1)

    #reset the default gradients
    optimizer.zero_grad()

    # forward pass
    ################ Your code ############
    output = model(images)
    loss = criterion(output, labels)
    #######################################

    loss.backward()
    optimizer.step()

    running_loss = running_loss+loss.item()
  else:
    print(f"Training loss: {running_loss/len(trainloader)} in epoch {e + 1}")
```

خروجی به اینصورت است(مقدار ضرر رو به کاهش است):

```
Training loss: 0.8750029215489877 in epoch 1
Training loss: 0.5109130711729593 in epoch 2
Training loss: 0.4558837284951576 in epoch 3
Training loss: 0.42331874707360256 in epoch 4
Training loss: 0.3991328792880847 in epoch 5
Training loss: 0.3811119918916017 in epoch 6
Training loss: 0.3657495561740927 in epoch 7
Training loss: 0.35525995263381044 in epoch 8
Training loss: 0.34387935384281915 in epoch 9
Training loss: 0.3346365075121556 in epoch 10
```

- حال عملکرد مدل را تست میکنیم و چند نمونه از خروجی مدل و لیبل واقعی داده را نمایش میدهیم:(همانطور که مشاهده میشود دقت مدل در حدود 80 درصد است.)

```
## Test your model
from d2l import torch as d2l
d2l.predict_ch3(model,testloader,n = 10)
```



سوال 6: ب) حال میخواهیم مدل ذکر شده را طوری تغییر دهیم که دچار overfitting شود. Overfit شدن مدل یعنی مدل روی داده های آموزشی مقدار ضرر خیلی کمی داشته باشد و جزئیات غیرضروری داده ها را یاد بگیرد و این باعث میشود که ضرر آن روی داده های تست متفاوت و خیلی بیشتر باشد. (Overfitting در شبکه‌های عصبی وقتی رخ می‌دهد که مدل به داده‌های آموزشی خود خیلی خوب عادت کرده باشد و به‌طور غیرمنطقی به جزئیات کوچک آن داده‌ها پاسخ دهد. (جزئیات غیرضروری داده های آموزشی را یادگرفته باشد.) به عبارت دیگر، مدل به جای یادگیری الگوهای کلی و عمومی، اطلاعات خاص به داده‌های آموزشی را حفظ کرده و در مقابل داده‌های جدید(داده های تست) به خوبی عمل نمی‌کند.)

برای overfit کردن مدل میتوان آن را پیچیده تر کرد(یعنی تعداد لایه ها را بیشتر کرد و تعداد نورون هر لایه را نیز بیشتر کرد.) و سپس تعداد ایپاک را بالا برد تا مدل خیلی روی داده آموزشی ریز شود.

بنابراین، مدل سوال قبل را به این صورت تغییر دادم:

```
# new model:
new_input_size = 784
new_out_size = 10
new_model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(new_input_size, 512), # increased neurons in the first layer
    nn.ReLU(),
    nn.Linear(512, 256), # additional layer with more neurons
    nn.ReLU(),
    nn.Linear(256, 128), # extra layer
    nn.ReLU(),
    nn.Linear(128, 64),
    nn.ReLU(),
    nn.Linear(64, new_out_size)
)

new_criterion = nn.CrossEntropyLoss()
new_optimizer = optim.SGD(new_model.parameters(), lr=0.01)

```

برای درک بهتر مدل میتوان آن را پرینت گرفت و خروجی را مشاهده کرد:

```
print(new_model)
```

```
Sequential(
  (0): Flatten(start_dim=1, end_dim=-1)
  (1): Linear(in_features=784, out_features=512, bias=True)
  (2): ReLU()
  (3): Linear(in_features=512, out_features=256, bias=True)
  (4): ReLU()
  (5): Linear(in_features=256, out_features=128, bias=True)
  (6): ReLU()
  (7): Linear(in_features=128, out_features=64, bias=True)
  (8): ReLU()
  (9): Linear(in_features=64, out_features=10, bias=True)
)
```

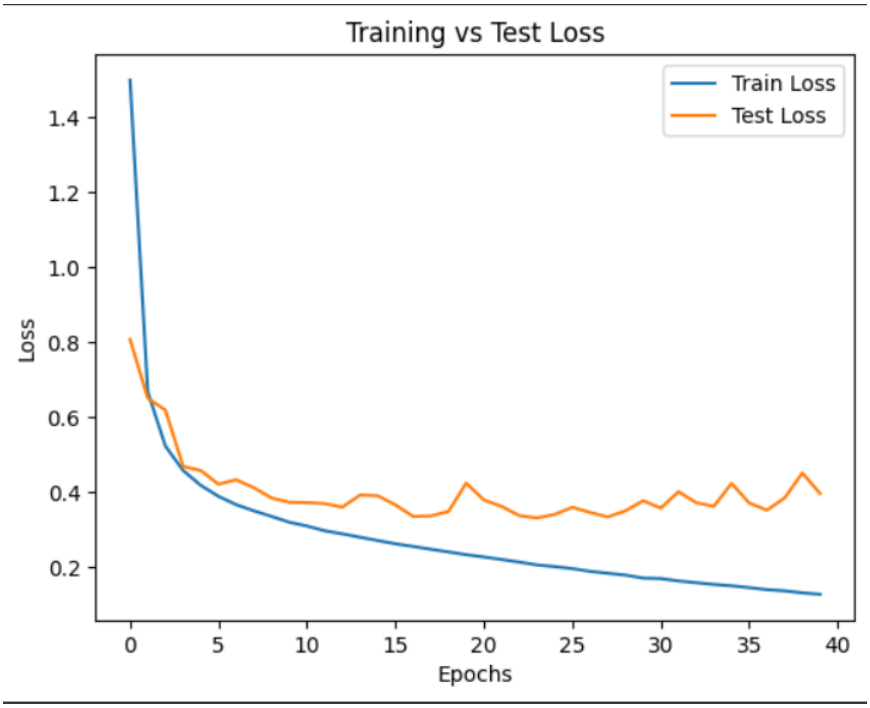
حال مدل جدید را آموزش داده و آن را روی داده تست، تست میکنیم و نتیجه ضرر هر ایپاک را پرینت میکنیم. در نهایت یک نمودار از روند تغییر ضرر در آموزش و تست مدل نشان میدهیم:

```
train_losses = []
test_losses = []

new_epochs = 40 # and increased epochs

for e in range(new_epochs):
    running_loss = 0
    correct = 0
    total = 0
    total_loss = 0

    # Training the model
    new_model.train()
    for images, labels in trainloader:
        images = images.view(images.shape[0], -1)
        new_optimizer.zero_grad()
        output = new_model(images)
        loss = new_criterion(output, labels)
        loss.backward()
        new_optimizer.step()
        running_loss += loss.item()

    train_loss = running_loss / len(trainloader)
    train_losses.append(train_loss)

```

```
    # Testing the model
    with torch.no_grad():
      for images, labels in testloader:
        images = images.view(images.shape[0], -1)
        output = new_model(images)
        loss = new_criterion(output, labels)
        total_loss += loss.item()
        _, predicted = torch.max(output.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    test_loss = total_loss / len(testloader)
    test_losses.append(test_loss)


    print(f"Epoch {e+1}/{new_epochs}, Train Loss: {train_loss:.4f}, Test Loss: {test_loss:.4f}")
# Plotting the losses
plt.plot(train_losses, label='Train Loss')
plt.plot(test_losses, label='Test Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Training vs Test Loss')
plt.show()
```

همان طور که مشاهده میشود، کد یادگیری و تست عین سوال قبل است و صرفا متغیر هایی برای ذخیره ضرر هر ایپاک تعریف شده اند. خروجی کد داده شده و نمودار ذکر شده به اینصورت است:

```
Epoch 1/40, Train Loss: 1.4998, Test Loss: 0.8086
Epoch 2/40, Train Loss: 0.6688, Test Loss: 0.6517
Epoch 3/40, Train Loss: 0.5239, Test Loss: 0.6198
Epoch 4/40, Train Loss: 0.4588, Test Loss: 0.4698
Epoch 5/40, Train Loss: 0.4191, Test Loss: 0.4588
Epoch 6/40, Train Loss: 0.3899, Test Loss: 0.4222
Epoch 7/40, Train Loss: 0.3677, Test Loss: 0.4341
Epoch 8/40, Train Loss: 0.3512, Test Loss: 0.4129
Epoch 9/40, Train Loss: 0.3363, Test Loss: 0.3855
Epoch 10/40, Train Loss: 0.3209, Test Loss: 0.3740
Epoch 11/40, Train Loss: 0.3108, Test Loss: 0.3732
Epoch 12/40, Train Loss: 0.2982, Test Loss: 0.3707
Epoch 13/40, Train Loss: 0.2899, Test Loss: 0.3612
Epoch 14/40, Train Loss: 0.2810, Test Loss: 0.3935
Epoch 15/40, Train Loss: 0.2720, Test Loss: 0.3915
Epoch 16/40, Train Loss: 0.2633, Test Loss: 0.3672
Epoch 17/40, Train Loss: 0.2563, Test Loss: 0.3364
Epoch 18/40, Train Loss: 0.2488, Test Loss: 0.3377
Epoch 19/40, Train Loss: 0.2416, Test Loss: 0.3496
Epoch 20/40, Train Loss: 0.2343, Test Loss: 0.4251
Epoch 21/40, Train Loss: 0.2282, Test Loss: 0.3802
Epoch 22/40, Train Loss: 0.2215, Test Loss: 0.3631
Epoch 23/40, Train Loss: 0.2146, Test Loss: 0.3385
Epoch 24/40, Train Loss: 0.2070, Test Loss: 0.3324
Epoch 25/40, Train Loss: 0.2025, Test Loss: 0.3416
Epoch 26/40, Train Loss: 0.1972, Test Loss: 0.3607
Epoch 27/40, Train Loss: 0.1899, Test Loss: 0.3470
Epoch 28/40, Train Loss: 0.1848, Test Loss: 0.3349
Epoch 29/40, Train Loss: 0.1799, Test Loss: 0.3512
Epoch 30/40, Train Loss: 0.1720, Test Loss: 0.3782
Epoch 31/40, Train Loss: 0.1707, Test Loss: 0.3585
Epoch 32/40, Train Loss: 0.1642, Test Loss: 0.4024
Epoch 33/40, Train Loss: 0.1597, Test Loss: 0.3729
Epoch 34/40, Train Loss: 0.1550, Test Loss: 0.3635
Epoch 35/40, Train Loss: 0.1514, Test Loss: 0.4245
Epoch 36/40, Train Loss: 0.1465, Test Loss: 0.3720
Epoch 37/40, Train Loss: 0.1411, Test Loss: 0.3531
Epoch 38/40, Train Loss: 0.1380, Test Loss: 0.3857
Epoch 39/40, Train Loss: 0.1325, Test Loss: 0.4523
Epoch 40/40, Train Loss: 0.1287, Test Loss: 0.3973
```



همان طور که مشاهده میشود عملکرد مدل روی داده آموزشی بسیار خوب بوده و ضرر آن رو به کاهش بوده است ولی روی داده تست، این چنین نبوده و مقدار ضرر افزایش و کاهش های متعددی داشته و در نهایت نسبت به ضرر آموزش، بیشتر است. برای اینکه مقدار بیش برازش ملموس تر شود، میتوان دیتای ورودی را تغییر داد و یا تعداد ایپاک را خیلی بیشتر کرد، که متاسفانه برای این تمرین امکان پذیر نبود.

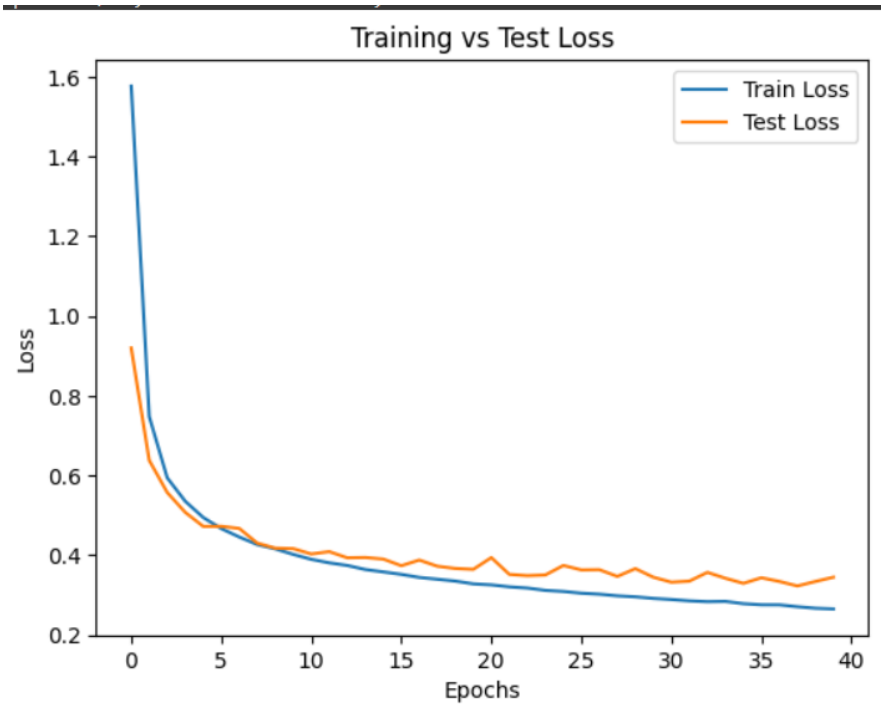
پ) برای استفاده از داده افزایی در بهبود بیش برازش، میتوان از دو راه حل random rotation, horizontal flipping استفاده کرد.

حال ابتدا transformation ها و تغییرات داده ها را لحاظ میکنیم:

```python
import torch
from torchvision import datasets, transforms

# Define transformations for data augmentation
transform_train = transforms.Compose([
    transforms.RandomRotation(degrees=10),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

# Download and load training data with the defined transformations
trainset = datasets.FashionMNIST('./data', download=True, train=True, transform=transform_train)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)

# Download and load test data with normalization
testset = datasets.FashionMNIST('./data', download=True, train=False, transform=transform_test)
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=False)

```

همان طور که مشاهده میکنید، trainloader, testloader طبق transformation_train جدید(که به آن خط 6 و 7 اضافه شده است.) تغییر کرده اند. حال همان کد های قبلی را دوباره قرار داده و مدل را آموزش و در نهایت تست میکنیم. خروجی نهایی به شرح زیر است:

```
Epoch 1/40, Train Loss: 1.5762, Test Loss: 0.9195
Epoch 2/40, Train Loss: 0.7476, Test Loss: 0.6377
Epoch 3/40, Train Loss: 0.5939, Test Loss: 0.5571
Epoch 4/40, Train Loss: 0.5351, Test Loss: 0.5073
Epoch 5/40, Train Loss: 0.4944, Test Loss: 0.4721
Epoch 6/40, Train Loss: 0.4666, Test Loss: 0.4723
Epoch 7/40, Train Loss: 0.4455, Test Loss: 0.4670
Epoch 8/40, Train Loss: 0.4265, Test Loss: 0.4301
Epoch 9/40, Train Loss: 0.4158, Test Loss: 0.4175
Epoch 10/40, Train Loss: 0.4018, Test Loss: 0.4163
Epoch 11/40, Train Loss: 0.3895, Test Loss: 0.4029
Epoch 12/40, Train Loss: 0.3807, Test Loss: 0.4088
Epoch 13/40, Train Loss: 0.3741, Test Loss: 0.3933
Epoch 14/40, Train Loss: 0.3639, Test Loss: 0.3940
Epoch 15/40, Train Loss: 0.3579, Test Loss: 0.3901
Epoch 16/40, Train Loss: 0.3518, Test Loss: 0.3735
Epoch 17/40, Train Loss: 0.3441, Test Loss: 0.3877
Epoch 18/40, Train Loss: 0.3396, Test Loss: 0.3722
Epoch 19/40, Train Loss: 0.3349, Test Loss: 0.3666
Epoch 20/40, Train Loss: 0.3280, Test Loss: 0.3644
Epoch 21/40, Train Loss: 0.3255, Test Loss: 0.3938
Epoch 22/40, Train Loss: 0.3205, Test Loss: 0.3517
Epoch 23/40, Train Loss: 0.3175, Test Loss: 0.3486
Epoch 24/40, Train Loss: 0.3117, Test Loss: 0.3503
Epoch 25/40, Train Loss: 0.3089, Test Loss: 0.3741
Epoch 26/40, Train Loss: 0.3046, Test Loss: 0.3630
Epoch 27/40, Train Loss: 0.3021, Test Loss: 0.3635
Epoch 28/40, Train Loss: 0.2980, Test Loss: 0.3467
Epoch 29/40, Train Loss: 0.2954, Test Loss: 0.3662
Epoch 30/40, Train Loss: 0.2915, Test Loss: 0.3446
Epoch 31/40, Train Loss: 0.2887, Test Loss: 0.3322
Epoch 32/40, Train Loss: 0.2853, Test Loss: 0.3348
Epoch 33/40, Train Loss: 0.2834, Test Loss: 0.3568
Epoch 34/40, Train Loss: 0.2840, Test Loss: 0.3420
Epoch 35/40, Train Loss: 0.2783, Test Loss: 0.3293
Epoch 36/40, Train Loss: 0.2755, Test Loss: 0.3433
Epoch 37/40, Train Loss: 0.2754, Test Loss: 0.3340
Epoch 38/40, Train Loss: 0.2706, Test Loss: 0.3227
Epoch 39/40, Train Loss: 0.2668, Test Loss: 0.3340
Epoch 40/40, Train Loss: 0.2651, Test Loss: 0.3446
```



همان طور که مشاهده میکنید با استفاده از این دو نوع از داده افزایی، داده های بیشتری به مجموعه داده خود اضافه کردیم تا مدل روی داده های آموزشی حساس نشود و جزئیات غیر ضروری را یاد نگیرد.(مقدار ضرر در آموزش و تست خیلی بهم نزدیک و مقداری کم دارند که ناشی از عملکرد خوب مدل است و رفع مشکل بیش برازش است.)

ت) در این قسمت برای رفع مشکل بیش برازش مدل قسمت (ب)، از منظم سازی L2 (که توضیح آن در سوالات قبل ذکر شده) استفاده میکنیم. کد تغییر داده شده به شرح زیر است: (پارامتر weight_decay مدل را از بیش برازش دور میکند و بصورت دلخواه و تجربی به مقدار 0.001 تنظیم شده است. شاید اگر این متغیر مقدار دیگری میگرفت نتیجه بهتری حاصل میشد ولی آموزش و تست مدل در این تعداد ایپاک، کار سختی است.)

```python
# the last introduced model:
new_input_size = 784
new_out_size = 10
new_model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(new_input_size, 512),
    nn.ReLU(),
    nn.Linear(512, 256),
    nn.ReLU(),
    nn.Linear(256, 128),
    nn.ReLU(),
    nn.Linear(128, 64),
    nn.ReLU(),
    nn.Linear(64, new_out_size)
)

new_criterion = nn.CrossEntropyLoss()
new_optimizer = optim.SGD(new_model.parameters(), lr=0.01, weight_decay=0.001) # adding L2 regularization

```

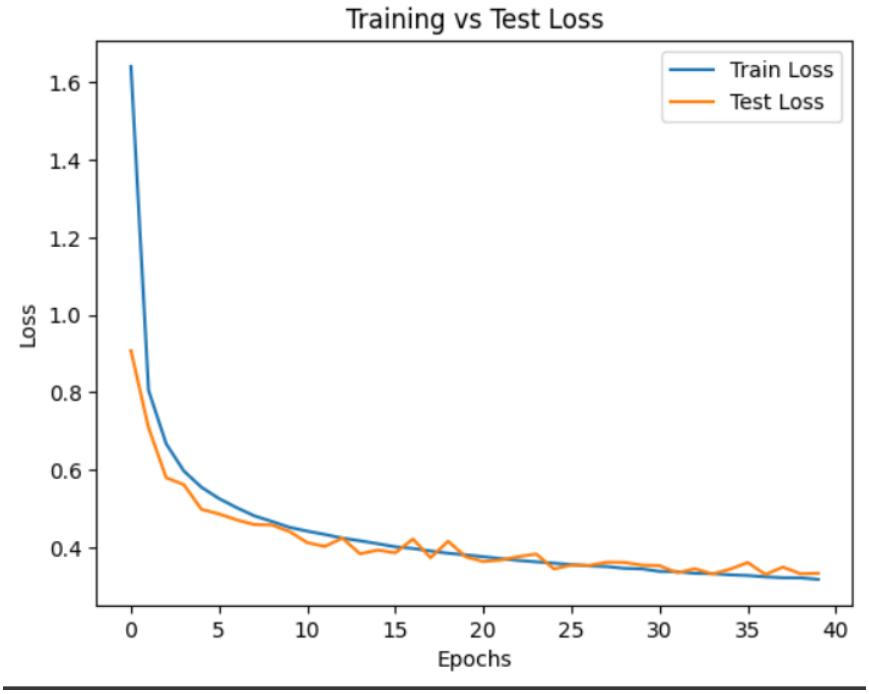
و در زمان آموزش مدل:

```python
train_losses = []
test_losses = []

new_epochs = 40 # and increased epochs

for e in range(new_epochs):
    running_loss = 0
    correct = 0
    total = 0
    total_loss = 0

    # Training the model
    new_model.train()
    for images, labels in trainloader:
        images = images.view(images.shape[0], -1)
        new_optimizer.zero_grad()
        output = new_model(images)
        loss = new_criterion(output, labels)

        # Calculating L2 regularization loss
        l2_reg = 0
        for param in new_model.parameters():
            l2_reg += torch.norm(param, 2)

        loss += 0.001 * l2_reg # with the regularization strength
        loss.backward()
        new_optimizer.step()
        running_loss += loss.item()

    train_loss = running_loss / len(trainloader)
    train_losses.append(train_loss)

```

کد بخش تست مدل و نمایش نمودار عین بخش های قبل(بدون تغییر است.)

حال نتیجه نهایی به شرح زیر است: (ضرر های تست و آموزش هر دو خیلی کم و نزدیک به هم هستند که نشان دهنده رفع مشکل بیش برازش است.)

```
Epoch 1/40, Train Loss: 1.6406, Test Loss: 0.9071
Epoch 2/40, Train Loss: 0.8048, Test Loss: 0.7082
Epoch 3/40, Train Loss: 0.6670, Test Loss: 0.5795
Epoch 4/40, Train Loss: 0.5970, Test Loss: 0.5617
Epoch 5/40, Train Loss: 0.5550, Test Loss: 0.4979
Epoch 6/40, Train Loss: 0.5257, Test Loss: 0.4860
Epoch 7/40, Train Loss: 0.5022, Test Loss: 0.4707
Epoch 8/40, Train Loss: 0.4812, Test Loss: 0.4586
Epoch 9/40, Train Loss: 0.4666, Test Loss: 0.4575
Epoch 10/40, Train Loss: 0.4514, Test Loss: 0.4405
Epoch 11/40, Train Loss: 0.4416, Test Loss: 0.4121
Epoch 12/40, Train Loss: 0.4331, Test Loss: 0.4023
Epoch 13/40, Train Loss: 0.4236, Test Loss: 0.4236
Epoch 14/40, Train Loss: 0.4167, Test Loss: 0.3831
Epoch 15/40, Train Loss: 0.4092, Test Loss: 0.3929
Epoch 16/40, Train Loss: 0.4013, Test Loss: 0.3857
Epoch 17/40, Train Loss: 0.3969, Test Loss: 0.4213
Epoch 18/40, Train Loss: 0.3905, Test Loss: 0.3732
Epoch 19/40, Train Loss: 0.3846, Test Loss: 0.4155
Epoch 20/40, Train Loss: 0.3804, Test Loss: 0.3753
Epoch 21/40, Train Loss: 0.3756, Test Loss: 0.3632
Epoch 22/40, Train Loss: 0.3707, Test Loss: 0.3671
Epoch 23/40, Train Loss: 0.3660, Test Loss: 0.3760
Epoch 24/40, Train Loss: 0.3625, Test Loss: 0.3827
Epoch 25/40, Train Loss: 0.3589, Test Loss: 0.3442
Epoch 26/40, Train Loss: 0.3553, Test Loss: 0.3541
Epoch 27/40, Train Loss: 0.3521, Test Loss: 0.3532
Epoch 28/40, Train Loss: 0.3503, Test Loss: 0.3614
Epoch 29/40, Train Loss: 0.3457, Test Loss: 0.3610
Epoch 30/40, Train Loss: 0.3443, Test Loss: 0.3539
Epoch 31/40, Train Loss: 0.3378, Test Loss: 0.3526
Epoch 32/40, Train Loss: 0.3370, Test Loss: 0.3343
Epoch 33/40, Train Loss: 0.3328, Test Loss: 0.3446
Epoch 34/40, Train Loss: 0.3321, Test Loss: 0.3312
Epoch 35/40, Train Loss: 0.3288, Test Loss: 0.3436
Epoch 36/40, Train Loss: 0.3272, Test Loss: 0.3607
Epoch 37/40, Train Loss: 0.3237, Test Loss: 0.3298
Epoch 38/40, Train Loss: 0.3214, Test Loss: 0.3489
Epoch 39/40, Train Loss: 0.3211, Test Loss: 0.3316
Epoch 40/40, Train Loss: 0.3173, Test Loss: 0.3330
```



ث) حال در این بخش ترکیبی از کل بخش های دیگر و dropout را استفاده میکنیم. در این بخش ابتدا طبق قسمت (پ) داده افزایی کرده و مجموعه داده خود را تغییر میدهیم:

```
transform = transforms.Compose([
    transforms.RandomRotation(15),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

# Download and load training data with the defined transformations
trainset = datasets.FashionMNIST('./data', download=True, train=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)
# Normalized test data without data augmentation
test_transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))])
testset = datasets.FashionMNIST('./data', download=True, train=False, transform=test_transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=False)
```

سپس مدل منظم سازی شده جدید را با لایه های dropout تعریف میکنیم. این مدل شامل 5 لایه(بدون در نظر گرفتن flatten) است که برای هر لایه به جز لایه خروجی، یک dropout گذاشته شده است(مقدار ورودی آن یک

فراپارامتر است که طبق تجربه این مقدار بهترین است.) در نهایت از منظم سازی L2 استفاده شده است.

مدل طراحی شده به شکل زیر است:

```python
# new model with Data Augmentation, Regularization, and Dropout
new_input_size = 784
new_out_size = 10

class RegularizedModel(nn.Module):
    def __init__(self):
        super(RegularizedModel, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(new_input_size, 512)
        self.relu1 = nn.ReLU()
        self.dropout1 = nn.Dropout(0.3) # dropout added after first layer

        self.fc2 = nn.Linear(512, 256)
        self.relu2 = nn.ReLU()
        self.dropout2 = nn.Dropout(0.3) # dropout added after second layer

        self.fc3 = nn.Linear(256, 128)
        self.relu3 = nn.ReLU()
        self.dropout3 = nn.Dropout(0.3) # dropout added after third layer

        self.fc4 = nn.Linear(128, 64)
        self.relu4 = nn.ReLU()

        self.fc5 = nn.Linear(64, new_out_size)

    def forward(self, x):
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.relu1(x)
        x = self.dropout1(x)

        x = self.fc2(x)
        x = self.relu2(x)
        x = self.dropout2(x)

        x = self.fc3(x)
        x = self.relu3(x)
        x = self.dropout3(x)

        x = self.fc4(x)
        x = self.relu4(x)

        x = self.fc5(x)
        return x

new_model = RegularizedModel()
new_criterion = nn.CrossEntropyLoss()
new_optimizer = optim.SGD(new_model.parameters(), lr=0.01, weight_decay=0.001) # with weight decay(L2)

```

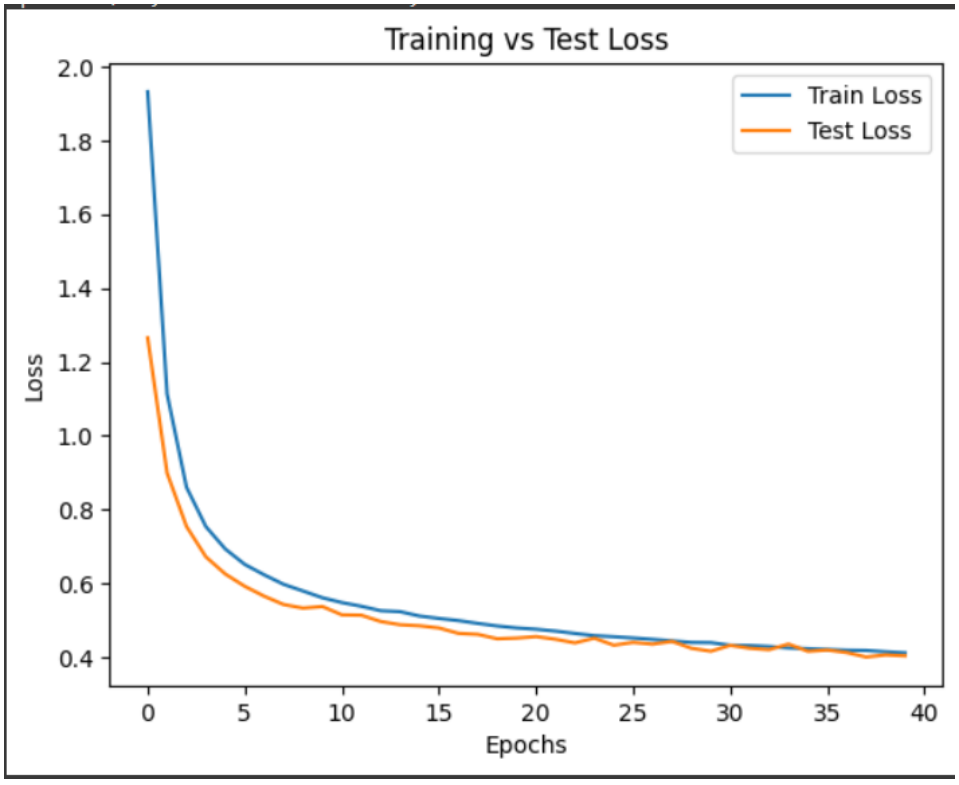
کد نوشته شده برای قسمت آموزش و تست مدل، عین قسمت (ت) سوال است:

```
train_losses = []
test_losses = []

new_epochs = 40 # and increased epochs

for e in range(new_epochs):
    running_loss = 0
    correct = 0
    total = 0
    total_loss = 0

    # Training the model
    new_model.train()
    for images, labels in trainloader:
        images = images.view(images.shape[0], -1)
        new_optimizer.zero_grad()
        output = new_model(images)
        loss = new_criterion(output, labels)

        # Calculating L2 regularization loss
        l2_reg = 0
        for param in new_model.parameters():
            l2_reg += torch.norm(param, 2)

        loss += 0.001 * l2_reg # with the regularization strength
        loss.backward()
        new_optimizer.step()
        running_loss += loss.item()

    train_loss = running_loss / len(trainloader)
    train_losses.append(train_loss)
```

```
    # Testing the model
    with torch.no_grad():
      for images, labels in testloader:
        images = images.view(images.shape[0], -1)
        output = new_model(images)
        loss = new_criterion(output, labels)
        total_loss += loss.item()
        _, predicted = torch.max(output.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    test_loss = total_loss / len(testloader)
    test_losses.append(test_loss)

    print(f"Epoch {e+1}/{new_epochs}, Train Loss: {train_loss:.4f}, Test Loss: {test_loss:.4f}")

# Plotting the losses
plt.plot(train_losses, label='Train Loss')
plt.plot(test_losses, label='Test Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Training vs Test Loss')
plt.show()
```

در نهایت خروجی مدل جدید با آن روش های رفع مشکل بیش برازش، به این صورت است: (همان طور که مشاهده میشود، ضرر ها به هم نزدیک و دارای مقدار کمی هستند که ناشی از رفع مشکل بیش برازش و عملکرد درست مدل بر روی داده ها میباشد. همچنین عملکرد مدل روی داده هایی که تا حالا ندیده است، بهتر است.)

```
Epoch 1/40, Train Loss: 1.9316, Test Loss: 1.2651
Epoch 2/40, Train Loss: 1.1135, Test Loss: 0.8992
Epoch 3/40, Train Loss: 0.8608, Test Loss: 0.7546
Epoch 4/40, Train Loss: 0.7540, Test Loss: 0.6722
Epoch 5/40, Train Loss: 0.6935, Test Loss: 0.6258
Epoch 6/40, Train Loss: 0.6518, Test Loss: 0.5926
Epoch 7/40, Train Loss: 0.6237, Test Loss: 0.5658
Epoch 8/40, Train Loss: 0.5979, Test Loss: 0.5432
Epoch 9/40, Train Loss: 0.5799, Test Loss: 0.5336
Epoch 10/40, Train Loss: 0.5616, Test Loss: 0.5379
Epoch 11/40, Train Loss: 0.5486, Test Loss: 0.5153
Epoch 12/40, Train Loss: 0.5385, Test Loss: 0.5144
Epoch 13/40, Train Loss: 0.5265, Test Loss: 0.4973
Epoch 14/40, Train Loss: 0.5239, Test Loss: 0.4885
Epoch 15/40, Train Loss: 0.5121, Test Loss: 0.4857
Epoch 16/40, Train Loss: 0.5056, Test Loss: 0.4798
Epoch 17/40, Train Loss: 0.4998, Test Loss: 0.4652
Epoch 18/40, Train Loss: 0.4921, Test Loss: 0.4625
Epoch 19/40, Train Loss: 0.4852, Test Loss: 0.4505
Epoch 20/40, Train Loss: 0.4799, Test Loss: 0.4524
Epoch 21/40, Train Loss: 0.4764, Test Loss: 0.4565
Epoch 22/40, Train Loss: 0.4713, Test Loss: 0.4493
Epoch 23/40, Train Loss: 0.4650, Test Loss: 0.4395
Epoch 24/40, Train Loss: 0.4590, Test Loss: 0.4522
Epoch 25/40, Train Loss: 0.4561, Test Loss: 0.4331
Epoch 26/40, Train Loss: 0.4527, Test Loss: 0.4407
Epoch 27/40, Train Loss: 0.4490, Test Loss: 0.4360
Epoch 28/40, Train Loss: 0.4448, Test Loss: 0.4435
Epoch 29/40, Train Loss: 0.4406, Test Loss: 0.4248
Epoch 30/40, Train Loss: 0.4401, Test Loss: 0.4168
Epoch 31/40, Train Loss: 0.4329, Test Loss: 0.4327
Epoch 32/40, Train Loss: 0.4322, Test Loss: 0.4249
Epoch 33/40, Train Loss: 0.4295, Test Loss: 0.4211
Epoch 34/40, Train Loss: 0.4258, Test Loss: 0.4361
Epoch 35/40, Train Loss: 0.4237, Test Loss: 0.4166
Epoch 36/40, Train Loss: 0.4215, Test Loss: 0.4201
Epoch 37/40, Train Loss: 0.4196, Test Loss: 0.4132
Epoch 38/40, Train Loss: 0.4191, Test Loss: 0.4011
Epoch 39/40, Train Loss: 0.4153, Test Loss: 0.4068
Epoch 40/40, Train Loss: 0.4125, Test Loss: 0.4044
```


Training vs Test Loss
Train Loss
Test Loss
Loss
Epochs
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0
5
10
15
20
25
30
35
40

پایان