

به نام خدا



دانشگاه تهران

دانشکده مهندسی برق و کامپیووتر

مدل‌های مولد عمیق

تمرين دوم

محمد طاها مجلسی کوپایی	نام و نام خانوادگی
۸۱۰۱۰۱۵۰۴	شماره دانشجویی
۱۴۰۳/۹/۷	تاریخ ارسال گزارش

فهرست

۳	سوال اول - مدل های مبتنی بر جریان
۳	سوال اول زیر بخش دوم
۱۱	سوال اول زیر بخش سوم
۱۰	سوال اول زیر بخش چهارم
۲۱	سوال اول زیر بخش پنجم (امتیازی)
۲۸	بخش دوم
۲۹	زیر بخش اول (۱۰ نمره)
۳۰	زیر بخش دوم (۵ نمره)
۳۴	سوال دوم - مدل های مولد مختصات
۳۴	بخش اول - GAN
۳۵	زیر بخش اول (۵ نمره)
۳۷	زیر بخش دوم (۵ نمره)
۳۹	زیر بخش سوم (۳ نمره)
۴۲	بخش دوم - wasserstein gan
۴۲	زیر بخش اول (۴ نمره)
۴۴	زیر بخش دوم (۴ نمره)
۴۵	زیر بخش سوم (۴ نمره)
۴۹	زیر بخش چهارم (۲ نمره)
۵۲	بخش سوم - پیاده سازی مدل های GAN
۵۲	زیر بخش اول (۴ نمره)

۵۷	زیر بخش دوم (۳ نمره)
۶۳	زیر بخش سوم (۶ نمره)
۶۴	زیر بخش چهارم (۴ نمره)
۶۵	زیر بخش پنجم (۲ نمره)
۶۷	زیر بخش ششم (۵ نمره)
۷۰	مراجع

سؤال اول - مدل های مبتنی بر جریان

بخش اول :

بخش اول کد:

```
class CouplingLayer(nn.Module):
    def __init__(self, input_dim, hidden_dim, mask):
        super(CouplingLayer, self).__init__()
        self.register_buffer('mask', mask)
        self.scale_net = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, input_dim),
            nn.Tanh()
        )
        self.translate_net = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, input_dim)
    )

    def forward(self, x):
        x_masked = x * self.mask
        s = self.scale_net(x_masked) * (1 - self.mask)
        t = self.translate_net(x_masked) * (1 - self.mask)
        y = x_masked + (1 - self.mask) * (x * torch.exp(s) + t)
        log_det_jacobian = (s * (1 - self.mask)).sum(dim=1)
        return y, log_det_jacobian

    def inverse(self, y):
        y_masked = y * self.mask
        s = self.scale_net(y_masked) * (1 - self.mask)
        t = self.translate_net(y_masked) * (1 - self.mask)
        x = y_masked + (1 - self.mask) * (y - t) * torch.exp(-s)
        return x
```

این کد یک لایه اتصال (Coupling Layer) را در شبکه های نرمال سازی جریان (Normalizing Flows) پیاده سازی می کند. هدف این لایه، تغییر توزیع داده های ورودی به توزیعی ساده تر (مثل توزیع نرمال

استاندارد) به صورت برگشتپذیر است. این ویژگی برای مدلسازی چگالی احتمال داده‌ها و تولید داده‌های جدید بسیار مفید است.

عملکرد:

- ماسک: بخشی از داده‌ها ثابت نگه داشته می‌شوند و تغییرات فقط روی قسمت مشخصی اعمال می‌شود.
- تغییرات (Translation و Scaling): داده‌های متغیر با:
 1. مقیاس‌دهی برای تغییر مقدارها.
 2. انتقال برای جابه‌جایی مقادیر، تغییر داده می‌شوند.
- برگشتپذیری: این لایه به‌گونه‌ای طراحی شده که می‌توان داده‌های اصلی را از داده‌های تبدیل شده بازسازی کرد.

کاربردها:

- مدلسازی چگالی احتمال: یادگیری دقیق توزیع داده‌ها.
- تولید داده‌های جدید: ایجاد داده‌هایی مشابه داده‌های واقعی.
- تحلیل داده‌ها: یافتن ساختارهای پنهان در توزیع داده‌ها.

```

class RealNVP(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_coupling_layers):
        super(RealNVP, self).__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.num_coupling_layers = num_coupling_layers

        masks = []
        for i in range(num_coupling_layers):
            mask = np.zeros(input_dim)
            if i % 2 == 0:
                mask[::2] = 1
            else:
                mask[1::2] = 1
            mask = torch.from_numpy(mask.astype(np.float32))
            masks.append(mask)

        self.layers = nn.ModuleList()
        for i in range(num_coupling_layers):
            self.layers.append(CouplingLayer(input_dim, hidden_dim, masks[i]))

    def forward(self, x):
        log_det_jacobian = 0
        for layer in self.layers:
            x, ldj = layer(x)
            log_det_jacobian += ldj
        return x, log_det_jacobian

    def inverse(self, z):
        for layer in reversed(self.layers):
            z = layer.inverse(z)
        return z

```

این کد پیاده‌سازی مدل **RealNVP** است که یکی از مدل‌های شبکه‌های نرمال‌سازی جریان (Normalizing Flows) محسوب می‌شود. هدف این مدل، تبدیل یک توزیع پیچیده به توزیع ساده‌تر (مثل توزیع نرمال) به صورت برگشت‌پذیر است. این مدل از لایه‌های اتصال (Coupling Layers) استفاده می‌کند که در کد قبلی توضیح داده شد.

عملکرد:

1. ورودی‌ها:

- **input_dim**: تعداد ویژگی‌های ورودی.
- **hidden_dim**: تعداد نورون‌های لایه مخفی.
- **num_coupling_layers**: تعداد لایه‌های اتصال.

2. ماسک‌ها:

- ماسک‌ها به صورت متناوب ساخته می‌شوند (ویژگی‌های زوج و فرد به طور متناوب ثابت یا تغییر می‌کنند). این ماسک‌ها کمک می‌کنند تا داده‌ها به صورت کنترل شده تغییر کنند.

3. ساختار مدل:

- مدل شامل چندین لایه اتصال (Coupling Layers) است که هر کدام بخشی از ورودی را تغییر داده و برگشت‌پذیری را حفظ می‌کنند.
- لایه‌ها به کمک `nn.ModuleList` ذخیره می‌شوند تا به صورت ترتیبی روی داده‌ها اعمال شوند.

4. فرآیند رفت (Forward):

- ورودی X از طریق تمام لایه‌های اتصال عبور داده می‌شود.
- در هر لایه، تغییرات داده‌ها و دترمینان ژاکوبین (برای احتمال محوری) محاسبه می‌شود.
- خروجی نهایی شامل داده‌های تغییریافته y و مجموع دترمینان ژاکوبین است.

5. فرآیند برگشت (Inverse):

- داده‌های خروجی Z از طریق همان لایه‌ها، اما به ترتیب معکوس، به ورودی اصلی X بازمی‌گردند.

تعاریف پارامترهای مدل :

```
input_dim = 28 * 28
hidden_dim = 1024
num_coupling_layers = 8
num_epochs = 10
learning_rate = 1e-3
```

همان طور که دیده میشود تعداد لایه های coupling در اینجا ۸ عدد تعریف شده است.

Optimizer :

```
model = RealNVP(input_dim, hidden_dim, num_coupling_layers).to(device)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

زیر بخش دوم:

تقسیم کردن دیتاست به ۸۰ درصد و ۲۰ درصد برای تست :

```
train_dataset = torchvision.datasets.FashionMNIST(root='./data', train=True, download=True, transform=transform)
batch_size = 128
|
train_size = int(0.8 * len(train_dataset))
validation_size = len(train_dataset) - train_size
|
train_subset, validation_subset = random_split(train_dataset, [train_size, validation_size])
|
train_loader = DataLoader(train_subset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(validation_subset, batch_size=batch_size, shuffle=False)
```

در این قسمت بخش های S و T تعریف شده اند.

```

class CouplingLayer(nn.Module):
    def __init__(self, input_dim, hidden_dim, mask):
        super(CouplingLayer, self).__init__()
        self.register_buffer('mask', mask)
        self.scale_net = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, input_dim),
            nn.Tanh()
        )
        self.translate_net = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, input_dim)
        )

```

جزئیات معماری:

هر دو شبکه شامل سه لایه اصلی هستند:

1. لایه اول (Linear Layer):

- تعداد نورون‌های ورودی برابر با `input_dim`, یعنی تعداد ویژگی‌های ورودی.
- تعداد نورون‌های خروجی برابر با `hidden_dim`, که یک فضای نمایشی با ابعاد بزرگ‌تر را برای یادگیری ویژگی‌ها فراهم می‌کند.

2. تابع فعال‌سازی غیرخطی (ReLU):

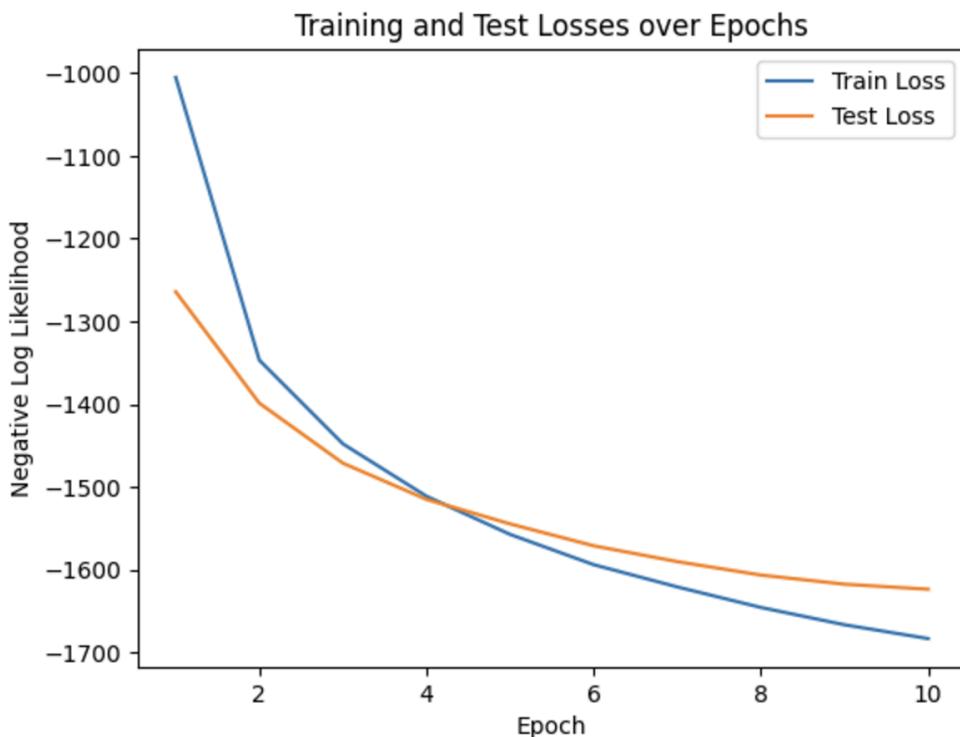
- برای افزایش ظرفیت یادگیری مدل، یک **ReLU (Rectified Linear Unit)** استفاده می‌شود که ویژگی‌های غیرخطی را به شبکه اضافه می‌کند.

3. لایه دوم (Linear Layer):

- تعداد نورون‌های ورودی برابر با `hidden_dim`, و تعداد خروجی برابر با `input_dim`, یعنی به ابعاد اصلی داده بازمی‌گردد.

4. تابع خروجی (Output Function):

- برای شبکه S: خروجی از طریق \tanh محدود می‌شود. این تابع باعث می‌شود که مقادیر مقیاس‌دهی در بازه $(-1, 1)$ قرار بگیرند و تغییرات مدل پایدار بمانند.
- برای شبکه T: خروجی بدون محدودسازی است، زیرا مقدار انتقال می‌تواند در هر بازه‌ای باشد.



تحلیل نمودار:

1. خطای آموزش (Train Loss - خط آبی):

- این خط نشان می‌دهد که مدل چگونه داده‌های آموزشی را یاد می‌گیرد.
- مقدار خطای آموزش در ابتدا بالا است اما به مرور زمان و با افزایش تعداد ایپاک‌ها کاهش می‌یابد، که نشان‌دهنده یادگیری بهتر مدل است.

2. خطای تست (Test Loss - خط نارنجی):

- این خط خطا مدل روی داده‌های تست (داده‌هایی که مدل هنگام آموزش ندیده است) را نشان می‌دهد.
 - خطای تست نیز مشابه خطای آموزش کاهش می‌یابد، که نشان می‌دهد مدل در یادگیری داده‌های جدید و تعمیم به داده‌های نادیده عملکرد خوبی دارد.
-

1. کاهش خطا:

- هر دو خطای آموزش و تست در طول ایپاک‌ها کاهش می‌یابند که نشان می‌دهد مدل به درستی در حال یادگیری است.
- کاهش خطای تست همراه با آموزش به این معناست که مدل **Overfitting** (بیش‌پرازش) نکرده است و به خوبی به داده‌های جدید تعمیم می‌دهد.

2. همگرایی (Convergence):

- کاهش خطا در ایپاک‌های ابتدایی (تا حدود ایپاک 5) سریع‌تر است، اما بعد از آن نرخ کاهش خطا کمتر می‌شود. این رفتار نشان‌دهنده نزدیک شدن مدل به همگرایی است.

3. عملکرد مناسب:

- خطای تست به خطای آموزش نزدیک است، که نشان‌دهنده این است که مدل داده‌های نادیده را به خوبی پیش‌بینی می‌کند

برخی از تصاویر تولید شده توسط مدل در پایان به این شکل خواهند بود :



زیر بخش سوم

تشخیص داده‌های خارج از توزیع (Out-of-Distribution Detection)

تشخیص داده‌های خارج از توزیع یکی از چالش‌های کلیدی در سیستم‌های یادگیری ماشین است. این موضوع به معنای شناسایی داده‌هایی است که به‌وضوح از توزیع داده‌هایی که مدل روی آن‌ها آموزش دیده، متفاوت هستند. این داده‌ها می‌توانند باعث عملکرد نامناسب مدل شوند، بهویژه در زمینه‌هایی مانند پزشکی، خودروهای خودران و امنیت سایبری، جایی که تصمیم‌گیری اشتباه ممکن است عواقب جدی داشته باشد.

مفهوم داده‌های خارج از توزیع

۱. **داده‌های داخل توزیع (In-Distribution Data)**: داده‌هایی که از همان توزیع آماری‌ای که مدل برای آن آموزش دیده است می‌آیند. این داده‌ها باید توسط مدل به خوبی پردازش شوند.

۲. **داده‌های خارج از توزیع (Out-of-Distribution Data)**: داده‌هایی که از توزیع آماری کاملاً متفاوت نسبت به داده‌های آموزشی می‌آیند. مدل معمولاً توانایی خوبی برای پیش‌بینی یا پردازش این داده‌ها ندارد.

هدف: طراحی مدلی که بتواند داده‌های خارج از توزیع را تشخیص دهد و از تصمیم‌گیری اشتباه جلوگیری کند.

روش‌های مبتنی بر احتمال برای تشخیص داده‌های خارج از توزیع

یک از رویکردهای اصلی در تشخیص داده‌های خارج از توزیع، استفاده از **مدل‌های احتمالاتی** است. این مدل‌ها به داده‌ها احتمال (**Likelihood**) اختصاص می‌دهند و داده‌هایی که احتمال بسیار کمی دارند را به عنوان داده‌های خارج از توزیع شناسایی می‌کنند.

مثال‌هایی از مدل‌های احتمالاتی:

۱. مدل‌های نرمال‌سازی جریان (Normalizing Flows)

- مدل‌هایی مانند RealNVP و Glow، داده‌ها را به یک توزیع ساده (ممکن‌گویی) تبدیل کرده و احتمال هر داده را محاسبه می‌کنند.
- داده‌هایی که توزیع آن‌ها با داده‌های آموزشی متفاوت است، احتمال کمتری دریافت می‌کنند و به عنوان داده خارج از توزیع شناسایی می‌شوند.

2. مدل‌های خودبازگشتی (Autoregressive Models)

- مدل‌هایی مانند WaveNet یا PixelCNN، احتمال شرطی هر بخش از داده (مثل هر پیکسل) را مدل می‌کنند.

- این مدل‌ها به داده‌های ناآشنا احتمال کمتری اختصاص می‌دهند.

3. مدل‌های مبتنی بر انرژی (Energy-Based Models)

- این مدل‌ها به داده‌ها مقدار انرژی اختصاص می‌دهند. داده‌های داخل توزیع انرژی کمتری دارند، در حالی که داده‌های خارج از توزیع انرژی بیشتری خواهند داشت.
- استفاده از امتیاز انرژی به جای احتمال خام به بهبود تشخیص داده‌های خارج از توزیع کمک می‌کند.

4. مدل‌های مبتنی بر یادگیری عمیق احتمالاتی:

- مدل‌های وریانس پایین مانند Bayesian Neural Networks از عدم قطعیت مدل برای تشخیص داده‌های خارج از توزیع استفاده می‌کنند. داده‌هایی که مدل روی آنها قطعیت پایینی دارد، می‌توانند به عنوان داده خارج از توزیع شناخته شوند.

یکی از روش‌های موجود: مدل‌های انرژی (Energy-Based Models)

مدل‌های انرژی یکی از روش‌های موثر برای تشخیص داده‌های خارج از توزیع هستند. در این مدل‌ها، به هر داده یک مقدار انرژی اختصاص داده می‌شود که نشان‌دهنده سطح "تناسب" آن داده با مدل است.

ایده اصلی:

- داده‌های داخل توزیع معمولاً انرژی کمی دارند، زیرا مدل آن‌ها را به خوبی یاد گرفته است.
- داده‌های خارج از توزیع انرژی بیشتری دارند، زیرا مدل نمی‌تواند آن‌ها را به خوبی بازنمایی کند.

مزایای مدل‌های انرژی:

- امکان بهینه‌سازی مستقیم برای تشخیص داده‌های خارج از توزیع.
 - عملکرد بهتر در مقایسه با مدل‌های مبتنی بر احتمال خام.
-

دلایل ناکامی برخی مدل‌ها در تشخیص داده‌های خارج از توزیع

1. اختصاص احتمال بالا به داده‌های خارج از توزیع:

- مدل‌های مبتنی بر احتمال (مانند Glow و RealNVP) گاهی به داده‌های خارج از توزیع احتمال‌های بالایی اختصاص می‌دهند. برای مثال، ممکن است به تصاویر ساده یا نویز سفید احتمال بالا بدهند، زیرا ساختار آن‌ها ساده‌تر است.

2. مشکل معیار احتمال خام:

- احتمال خام نمی‌تواند به طور موثر داده‌های داخل توزیع را از داده‌های خارج از توزیع متمايز کند. به همین دلیل، استفاده از معیارهای مکمل (مثل امتیاز انرژی یا ترکیب احتمالات) ضروری است.

3. وابستگی به توزیع داده‌های آموزشی:

- مدل‌هایی که صرفاً بر اساس داده‌های داخل توزیع آموزش دیده‌اند، توانایی تعمیم به داده‌های جدید و متفاوت را ندارند.

4. عدم یادگیری ویژگی‌های کافی:

- اگر داده‌های آموزشی تنوع کمی داشته باشند، مدل نمی‌تواند به خوبی ویژگی‌های کلیدی داده‌های داخل توزیع را یاد بگیرد، که باعث می‌شود تشخیص داده‌های خارج از توزیع دشوارتر شود.

5. عدم بهینه‌سازی مستقیم برای داده‌های خارج از توزیع:

- بسیاری از مدل‌ها فقط برای یادگیری داده‌های داخل توزیع طراحی شده‌اند و هیچ مکانیسم مستقیمی برای شناسایی داده‌های ناآشنا ندارند.

جمع‌بندی

- تشخیص داده‌های خارج از توزیع بخش مهمی از یادگیری ماشین است، به‌ویژه در سیستم‌هایی که نیازمند اطمینان بالا هستند.
- روش‌های مبتنی بر احتمال مانند Normalizing Flows و مدل‌های انرژی ابزارهای مؤثری هستند، اما در برخی موارد (مانند تخصیص احتمال بالا به داده‌های اشتباه) محدودیت دارند.
- ترکیب چندین روش یا استفاده از معیارهای تکمیلی مانند امتیاز انرژی می‌تواند به بهبود عملکرد تشخیص کمک کند. همچنین آموزش مدل‌ها با داده‌های متنوع‌تر و استفاده از مکانیسم‌های خاص برای تشخیص داده‌های خارج از توزیع از راهکارهای موثر برای بهبود این فرآیند هستند.

زیر‌بخش چهارم

```
def compute_log_likelihood(model, data_loader):  
    model.eval()  
    log_likelihoods = []  
    with torch.no_grad():  
        for data, _ in data_loader:  
            data = data.view(-1, input_dim).to(device)  
            z, log_det_jacobian = model(data)  
            log_pz = -0.5 * torch.sum(z ** 2 + np.log(2 * np.pi), dim=1)  
            log_px = log_pz + log_det_jacobian  
            log_likelihoods.append(log_px.cpu().numpy())  
    return np.concatenate(log_likelihoods)
```

این تابع لگاریتم درستنمایی (Log-Likelihood) داده‌های ورودی را محاسبه می‌کند، که معیاری از احتمال این داده‌ها نسبت به توزیع آموخته شده توسط مدل است. عملکرد آن را می‌توان به چند مرحله تقسیم کرد:

1. تبدیل داده‌ها به فضای نهفته (Latent Space) :

- مدل داده‌های ورودی را به یک فضای نهفته ساده‌تر (معمولًاً توزیع گوسی با میانگین صفر و واریانس یک) نگاشت می‌کند.
- این تبدیل به مدل امکان می‌دهد داده‌های پیچیده را به شکلی ساده‌تر نمایان کند.

2. محاسبه احتمال داده‌ها در فضای نهفته ($\log p(z)$) :

- احتمال داده‌ها در فضای نهفته بر اساس فرض توزیع گوسی محاسبه می‌شود.
- مقادیر داده‌هایی که به خوبی با این توزیع مطابقت دارند، احتمال بالاتری خواهند داشت.

3. بازگشت به فضای اصلی ($\log p(x)$) :

- برای بازگشت به فضای اصلی داده‌ها، تأثیر تغییرات ناشی از مدل (از طریق دترمینان ژاکوبین) لحاظ می‌شود.
- این مقدار نشان‌دهنده تأثیر تبدیل داده از فضای اصلی به فضای نهفته است و به احتمال داده‌ها در فضای اصلی اضافه می‌شود.

4. محاسبه نهایی:

- ترکیب احتمال فضای نهفته $\log | \det J |$ و تأثیر تبدیل $\log p(z)$ مقدار لگاریتم درستنمایی در فضای اصلی $\log p(x)$ را به دست می‌دهد.

کاربرد:

- تشخیص داده‌های خارج از توزیع (OOD Detection): داده‌هایی که احتمال کمتری دارند (لگاریتم درست‌نمایی پایین)، معمولاً خارج از توزیع داده‌های آموزشی هستند.
- ارزیابی مدل: بررسی می‌کند که مدل چگونه داده‌های ورودی را توضیح می‌دهد و آیا توانایی تعمیم‌دهی مناسبی دارد یا خیر.

```

▶ fashion_train_ll = compute_log_likelihood(model, train_loader)
fashion_test_ll = compute_log_likelihood(model, test_loader)
mnist_ll = compute_log_likelihood(model, mnist_loader)
kmnist_ll = compute_log_likelihood(model, kmnist_loader)

print("Fashion MNIST Train Log Likelihood:", np.mean(fashion_train_ll))
print("Fashion MNIST Test Log Likelihood:", np.mean(fashion_test_ll))
print("MNIST Log Likelihood:", np.mean(mnist_ll))
print("KMNIST Log Likelihood:", np.mean(kmnist_ll))

⇒ Fashion MNIST Train Log Likelihood: 1706.9332
Fashion MNIST Test Log Likelihood: 1621.3966
MNIST Log Likelihood: 890.0156
KMNIST Log Likelihood: -1850.9489

```

همان طور که مشاهده می‌شود برای داده‌های قسمت train مقدار میانگین likelihood مقدار بالاتری نسبت به داده‌های تست بود.

اما در log likelihood برای داده‌های Mnist انتظار داشتیم که مقدار میانگین مقداری پایین تری باشد چون داده‌های اعداد ربط و شباهت زیادی به داده‌های fashion MNIST ندارد و انتظار داشتیم که مدل احتمال‌های کمتری را به آن‌ها اختصاص داده باشد و در داده‌های KMNIST مقدار میانگین کم شده که نشان دهنده‌ی دادن احتمال‌های پایین تر به این داده‌ها می‌باشد

پس در حالت کلی داریم :

تحلیل مقادیر Log-Likelihood برای داده‌های مختلف

در این آزمایش، مقادیر میانگین لگاریتم درست‌نمایی (Log-Likelihood) برای داده‌های **FashionISTA** (آموزشی و تست)، **MNIST** و **KMNIST** محاسبه و مقایسه شده‌اند. این مقادیر نشان‌دهنده میزان احتمال اختصاص داده‌شده توسط مدل به این داده‌ها هستند. تحلیل نتایج به شرح زیر است:

1. داده‌های **Fashion MNIST** (آموزشی و تست):

- مقدار **Log-Likelihood** برای داده‌های آموزشی **FashionMNIST** بالاتر از داده‌های تستی است: **FashionMNIST**
- **FashionMNIST Train Log Likelihood:** 1706.9332
- **FashionMNIST Test Log Likelihood:** 1621.3966

تحلیل:

- این رفتار طبیعی است زیرا مدل روی داده‌های آموزشی تمرین کرده و آن‌ها را به خوبی یاد گرفته است، بنابراین احتمال بالاتری به این داده‌ها اختصاص می‌دهد.
 - داده‌های تست از همان توزیع هستند اما مدل هرگز آن‌ها را ندیده است، به همین دلیل احتمال پایین‌تری نسبت به داده‌های آموزشی دریافت می‌کنند.
 - این تفاوت نشان‌دهنده تعمیم‌دهی مناسب مدل است.
-

2. داده‌های **MNIST**:

- مقدار میانگین **Log-Likelihood** برای داده‌های **MNIST**: 890.0156

تحلیل:

- مدل به داده‌های **MNIST**، که شامل اعداد دست‌نویس است، احتمال نسبتاً بالایی اختصاص داده است.

- انتظار می‌رفت که این مقدار بسیار پایین‌تر باشد، زیرا داده‌های MNIST از نظر محتوایی (اعداد) با داده‌های **FashionMNIST** (تصاویر لباس) تفاوت زیادی دارند.
 - این رفتار نشان‌دهنده یکی از مشکلات مدل‌های مولد مانند **RealNVP** است. این مدل‌ها گاهی به داده‌هایی که ساده‌تر هستند (مثل اعداد MNIST) احتمال بالایی اختصاص می‌دهند، حتی اگر این داده‌ها خارج از توزیع اصلی باشند. دلیل این مسئله می‌تواند به ساختار ساده داده‌های **MNIST** و شباهت‌های آماری اولیه (مثل روش‌نایی یا الگوهای ساده) مربوط باشد.
-

3. داده‌های KMNIST :

- مقدار میانگین Log-Likelihood برای داده‌های KMNIST:
○ 1850.9489-
 - تحلیل:
 - مدل به داده‌های KMNIST، که شامل کاراکترهای ژاپنی است، احتمال بسیار پایینی اختصاص داده است.
 - این رفتار قابل انتظار است زیرا داده‌های KMNIST هم از نظر محتوا (حروف ژاپنی) و هم از نظر ساختاری (الگوهای پیچیده‌تر) کاملاً متفاوت از داده‌های **FashionMNIST** هستند.
 - مقدار بسیار پایین Log-Likelihood نشان‌دهنده این است که مدل این داده‌ها را به عنوان داده‌های خارج از توزیع (Out-of-Distribution) شناسایی کرده است.
-

نتیجه‌گیری کلی:

1. مدل احتمال بالاتری به داده‌های آموزشی اختصاص داده است، که نشان‌دهنده یادگیری موفق توزیع داده‌های داخل توزیع است.
2. مقدار Log-Likelihood پایین‌تر برای داده‌های تست، اما همچنان قابل قبول، نشان‌دهنده عملکرد مناسب مدل در تعمیم‌دهی به داده‌های نادیده است.

3. اختصاص احتمال نسبتاً بالا به داده‌های **MNIST** نشان‌دهنده یکی از محدودیت‌های مدل است. این مدل احتمال بالایی به داده‌های ساده و آماری مشابه (مثل اعداد MNIST) اختصاص می‌دهد، حتی اگر آن‌ها از توزیع اصلی نباشند.

4. اختصاص احتمال بسیار پایین به داده‌های **KMNIST** نشان می‌دهد که مدل می‌تواند داده‌هایی که کاملاً خارج از توزیع اصلی هستند را شناسایی کند. این رفتار نشان‌دهنده توانایی مدل در تشخیص داده‌های نامرتبط است.

پیشنهاد بهبود:

- استفاده از روش‌های انرژی (Energy-Based Models) یا ترکیب چند معیار برای کاهش مسئله تشخیص احتمال بالا به داده‌های ساده مانند MNIST.
- آموزش مدل روی داده‌های متنوع‌تر یا استفاده از تکنیک‌های تنظیم‌گرایی (Regularization) برای بهبود عملکرد در تشخیص داده‌های خارج از توزیع.

بخش پنجم امتیازی :

```
▶ class Decoder(nn.Module):
    def __init__(self, latent_dim, output_dim):
        super(Decoder, self).__init__()
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, 512),
            nn.ReLU(),
            nn.Linear(512, output_dim)
        )
    def forward(self, x):
        return self.decoder(x)
```

```
▶ class Encoder(nn.Module):
    def __init__(self, input_dim, latent_dim):
        super(Encoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, 512),
            nn.ReLU(),
            nn.Linear(512, latent_dim)
        )
    def forward(self, x):
        return self.encoder(x)
```

در ابتدا یک انکودر و یک دیکودر ای را ترین کرده ایم که تصاویر را بتواند به بخش Latent خودشان ببرند .
یعنی تصاویری که گرفته میشود به فضای نهان توسط انکودر برده خواهد شد .

```

    encoder = Encoder(input_dim, latent_dim).to(device)
    decoder = Decoder(latent_dim, input_dim).to(device)

    encoder_decoder_optimizer = optim.Adam(list(encoder.parameters()) + list(decoder.parameters()), lr=learning_rate

    for epoch in range(num_epochs_encoder):
        encoder.train()
        decoder.train()
        train_loss = 0
        for data, _ in train_loader:
            data = data.view(-1, input_dim).to(device)
            encoder_decoder_optimizer.zero_grad()
            latent = encoder(data)
            reconstructed = decoder(latent)
            loss = nn.MSELoss()(reconstructed, data)
            loss.backward()
            encoder_decoder_optimizer.step()
            train_loss += loss.item()
        print(f"Epoch {epoch + 1}, Encoder-Decoder Train Loss: {train_loss / len(train_loader)}")

```

در گام بعدی آن را با داده های train آموزش داده ایم که بتوانیم آن داده هارا به قسمت تقریبا مشخص ای از latent space برسانیم.

```

8] def encode_dataset(encoder, data_loader):
    encoded_data = []
    with torch.no_grad():
        for data, _ in data_loader:
            data = data.view(-1, input_dim).to(device)
            latent = encoder(data)
            encoded_data.append(latent.cpu())
    return torch.cat(encoded_data)

# Encode datasets
fashion_train_latent = encode_dataset(encoder, train_loader)
fashion_test_latent = encode_dataset(encoder, test_loader)
mnist_latent = encode_dataset(encoder, mnist_loader)
kmnist_latent = encode_dataset(encoder, kmnist_loader)

```

int: num_coupling_layer

: encoder latent که داده ها را encode کند به فضای

```

latent_input_dim = latent_dim
realnvp_model = RealNVP(latent_input_dim, hidden_dim, num_coupling_layers).to(device)
realnvp_optimizer = optim.Adam(realnvp_model.parameters(), lr=learning_rate)

for epoch in range(num_epochs):
    realnvp_model.train()
    train_loss = 0
    for i in range(0, len(fashion_train_latent), batch_size):
        batch = fashion_train_latent[i:i + batch_size].to(device)
        realnvp_optimizer.zero_grad()
        z, log_det_jacobian = realnvp_model(batch)
        log_pz = -0.5 * torch.sum(z ** 2 + np.log(2 * np.pi), dim=1)
        loss = -torch.mean(log_pz + log_det_jacobian)
        loss.backward()
        realnvp_optimizer.step()
        train_loss += loss.item()
    print(f"Epoch {epoch + 1}, RealNVP Train Loss: {train_loss / len(train_loader)}")

```

Epoch 1, RealNVP Train Loss: -87.72435881169638

در این قسمت مدل realNVP را با داده های فضای LATENT آموزش داده ایم .

که مدل یاد بگیرد داده های فضای latent را به تصاویر اصلی مپ کند

یعنی مثل حالت قلبی نیست که از فضای نرمال استاندارد به فضای تصاویر رفته باشیم داریم از فضای Latent یک انکودر که قبل از تصاویر ترین شده بود به داده های اصلی مپ میکنیم.

پس باید منطقی به نظر برسد که داده های دیگر که توسط انکودر یاد گرفته نشده بودند به فضای پرتو در فضای نهان در انکودر بروند و به همین نسبت در زمان دادن احتمال توسط مدل جریان احتمالی که به آنها نسبت داده شده است مقدار کمتری شده باشد .

```

def compute_log_likelihood_latent(model, data):
    model.eval()
    log_likelihoods = []
    with torch.no_grad():
        for i in range(0, len(data), batch_size):
            batch = data[i:i + batch_size].to(device)
            z, log_det_jacobian = model(batch)
            log_pz = -0.5 * torch.sum(z ** 2 + np.log(2 * np.pi), dim=1)
            log_px = log_pz + log_det_jacobian
            log_likelihoods.append(log_px.cpu().numpy())
    return np.concatenate(log_likelihoods)

fashion_train_ll = compute_log_likelihood_latent(realnvp_model, fashion_train_latent)
fashion_test_ll = compute_log_likelihood_latent(realnvp_model, fashion_test_latent)
mnist_ll = compute_log_likelihood_latent(realnvp_model, mnist_latent)
kmnist_ll = compute_log_likelihood_latent(realnvp_model, kmnist_latent)

print("FashionMNIST Train Log-Likelihood:", np.mean(fashion_train_ll))
print("FashionMNIST Test Log-Likelihood:", np.mean(fashion_test_ll))
print("MNIST Log-Likelihood:", np.mean(mnist_ll))
print("KMNIST Log-Likelihood:", np.mean(kmnist_ll))

```

FashionMNIST Train Log-Likelihood: 166.30286
 FashionMNIST Test Log-Likelihood: 151.63997
 MNIST Log-Likelihood: 2.3912132
 KMNIST Log-Likelihood: -66.17415

همان طور که میبینیم مطابق با نتایج به جواب درست رسیدیم یعنی به داده های خارج از توزیع اصلی مقادیر کمتری نسبت داده شده بود .

تحلیل نتایج :Log-Likelihood

نتایج محاسبه Log-Likelihood برای داده های مختلف به شرح زیر است:

- FashionMNIST Train Log-Likelihood:** 166.30286 •
- FashionMNIST Test Log-Likelihood:** 151.63997 •
- MNIST Log-Likelihood:** 2.3912132 •
- KMNIST Log-Likelihood:** -66.17415 •

حال این مقادیر را تحلیل میکنیم:

(آموزشی و تست) FashionMNIST .1

:FashionMNIST Train

- مقدار Log-Likelihood: 166.30166.30
- این مقدار بالاترین مقدار است که نشان می‌دهد مدل به خوبی داده‌های آموزشی FashionMNIST را یاد گرفته است.
- مقدار بالای Log-Likelihood برای داده‌های آموزشی نشان‌دهنده این است که مدل می‌تواند توزیع داده‌هایی را که روی آن‌ها آموزش دیده، به خوبی توصیف کند.

:FashionMNIST Test

- مقدار Log-Likelihood: 151.63151.63
- این مقدار پایین‌تر از داده‌های آموزشی است، اما همچنان بالا است.
- این کاهش نشان‌دهنده این است که مدل داده‌های تست را ندیده است، اما به دلیل شباهت ساختاری این داده‌ها با داده‌های آموزشی، مدل می‌تواند آن‌ها را به خوبی توضیح دهد.

تحلیل:

- این اختلاف میان داده‌های آموزشی و تست کاملاً طبیعی است و نشان می‌دهد که مدل توانایی تعمیم‌دهی خوبی دارد.
- مقادیر بالای Log-Likelihood برای هر دو مجموعه نشان‌دهنده این است که مدل به درستی داده‌های داخل توزیع (In-Distribution) را توصیف می‌کند.

(خارج از توزیع) MNIST .2

- مقدار Log-Likelihood: 2.392.39
- این مقدار بسیار پایین‌تر از داده‌های FashionMNIST است.
- با توجه به اینکه MNIST شامل اعداد دست‌نویس است و ساختار آن متفاوت از داده‌های FashionMNIST (تصاویر لباس) است، انتظار می‌رود مدل به این داده‌ها احتمال کمتری اختصاص دهد.

تحلیل:

- مقدار بسیار پایین Log-Likelihood برای MNIST نشان می‌دهد که مدل توانسته این داده‌ها را به عنوان داده‌های خارج از توزیع (OOD) شناسایی کند.

- این نشان می‌دهد که استفاده از فضای نهان (Latent Space) باعث کاهش شباهت‌های سطحی (مانند روشنایی و الگوهای ساده) بین داده‌های MNIST و FashionMNIST شده است.
-

(خارج از توزیع و متفاوت‌تر) KMNIST .3

- مقدار Log-Likelihood: -66.17-66.17
- این مقدار حتی از MNIST نیز پایین‌تر است. KMNIST شامل کاراکترهای ژاپنی است که ساختار آن بسیار متفاوت‌تر از FashionMNIST است.
- مدل احتمال بسیار کمی به این داده‌ها اختصاص داده است، که نشان‌دهنده توانایی خوب مدل در تشخیص داده‌های بسیار متفاوت از توزیع اصلی است.

تحلیل:

- مقدار بسیار پایین‌تر Log-Likelihood برای KMNIST نسبت به MNIST نشان‌دهنده این است که مدل داده‌های KMNIST را به عنوان داده‌های کاملاً خارج از توزیع شناسایی کرده است.
 - این رفتار مطابق انتظار است، زیرا KMNIST از لحاظ ساختاری شباهت کمتری به FashionMNIST دارد.
-

4. مقایسه و نتیجه‌گیری

نتایج برای داده‌های داخل توزیع (FashionMNIST):

- مقادیر Log-Likelihood برای داده‌های آموزشی و تست FashionMNIST بالا هستند و مدل به خوبی می‌تواند این داده‌ها را توضیح دهد.
- اختلاف منطقی میان داده‌های آموزشی و تست نشان‌دهنده توانایی تعمیم‌دهی مدل است.

نتایج برای داده‌های خارج از توزیع (KMnist و MNIST):

- مقدار Log-Likelihood پایین‌تری نسبت به FashionMNIST دارد، اما همچنان مثبت است. این نشان می‌دهد که مدل آن را به عنوان داده‌ای نسبتاً نامرتبط شناسایی کرده است.

- مقدار Log-Likelihood بسیار پایین‌تری نسبت به MNIST دارد. این نشان می‌دهد که مدل داده‌های KMNIST بسیار متفاوت را به خوبی شناسایی می‌کند.
-

5. اثر استفاده از فضای نهان

- انتقال داده‌ها به فضای نهان با انکودر باعث شده مدل ویژگی‌های معنادار را بهتر درک کند و داده‌های خارج از توزیع را دقیق‌تر تشخیص دهد.
 - در مقایسه با اجرای مستقیم RealNVP روی داده‌های خام:
 1. مقدار Log-Likelihood برای داده‌های FashionMNIST همچنان بالا است، که نشان می‌دهد مدل دقیق‌تر را برای داده‌های داخل توزیع حفظ کرده است.
 2. تفاوت معناداری بین Log-Likelihood داده‌های KMNIST و MNIST مشاهده می‌شود، که نشان‌دهنده توانایی بهتر مدل در تمایز داده‌های خارج از توزیع است.
-

6. پیشنهادات بهبود

- اگر فاصله بین MNIST و FashionMNIST در Log-Likelihood کمتر از انتظار باشد، می‌توان انکودر را با تنظیمات دقیق‌تری (مثل اضافه کردن لایه‌های بیشتر یا استفاده از Reconstruction Loss بهبود داد).
- مدل می‌تواند برای تشخیص دقیق‌تر داده‌های OOD با ترکیب RealNVP و روش‌های انرژی‌محور (Energy-Based) (Models) تقویت شود.

جمع‌بندی نهایی

- مدل با استفاده از فضای نهان توانسته داده‌های داخل توزیع را به خوبی توضیح دهد و داده‌های خارج از توزیع (به‌ویژه KMNIST) را به طور دقیق شناسایی کند.
- این نتایج نشان می‌دهند که استفاده از انکودر و فضای نهان موفق بوده و بهبود عملکرد مدل در تشخیص داده‌های خارج از توزیع را فراهم کرده است.

بخش دوم :

پاسخ به زیر بخش اول :

مدل‌های جریان باقی‌مانده (Residual Flows)

مدل‌های جریان باقی‌مانده یکی از مدل‌های پیشرفته یادگیری هستند که برای مدل‌سازی توزیع‌های پیچیده و تولید نمونه‌های دقیق از داده‌ها طراحی شده‌اند. هدف اصلی این مدل‌ها استفاده از تبدیل‌های معکوس‌پذیر است که ضمن حفظ دقت، محاسبات پیچیده را ساده‌تر می‌کنند. باید قدم‌به‌قدم ساختار این مدل‌ها را توضیح دهیم.

معکوس‌پذیری یعنی چه؟

فرض کنید می‌خواهیم داده‌ای را از یک فضای ساده (مثل توزیع نرمال) به فضای پیچیده (مثل یک تصویر) تبدیل کنیم یا برعکس. برای این کار نیاز داریم که این تبدیل به‌گونه‌ای باشد که:

1. یک به یک باشد (یعنی هر ورودی یک خروجی یکتا داشته باشد و برعکس).
2. برگشت‌پذیر باشد (بتوانیم از خروجی دوباره ورودی را بدست آوریم).

مدل‌های جریان باقی‌مانده از همین خاصیت برگشت‌پذیری استفاده می‌کنند تا توزیع‌های پیچیده داده‌ها را با تبدیل‌هایی که به راحتی قابل محاسبه هستند، مدل کنند.

ساختار مدل

ساختار اصلی این مدل‌ها بر اساس یک بلوک باقی‌مانده معکوس‌پذیر است که به شکل زیر تعریف می‌شود:

$$y = f(x) = x + g(x)$$

- x : ورودی به بلوک است.
- y : خروجی بلوک است.

$g(x)$: یک تابع ساده (معمولًاً یک شبکه عصبی) است که تغییرات لازم را روی xxx اعمال می‌کند.

ویژگی مهم این فرمول این است که چون $f(x)$ را به راحتی به عقب برگردانیم یک تابع ساده است، می‌توانیم $g(x)$ را به راحتی به عقب برگردانیم (معکوس کنیم).

چگونه معکوس‌پذیری تضمین می‌شود؟

برای اینکه $f(x)$ معکوس‌پذیر باشد، نیاز داریم که $g(x)$ یک نگاشت انقباضی باشد. یعنی وقتی $g(x)$ روی ورودی‌های مختلف اعمال شود، این ورودی‌ها بیش از حد به هم نزدیک نشوند.

به زبان ساده، اگر فاصله بین دو نقطه x_1 و x_2 برابر d باشد، بعد از اعمال $g(x)$ ، فاصله بین خروجی‌ها باید کمتر از dd باشد:

$$\|g(x_1) - g(x_2)\| \leq L \|x_1 - x_2\|$$

که L باید کوچک‌تر از یک باشد

چگونه معکوس را پیدا کنیم؟

برای یافتن معکوس $f(x)$ ، کافی است معادله زیر را حل کنیم:

$$x = f^{-1}(y) = y - g(x)$$

چون $g(x)$ به x بستگی دارد، این معادله به صورت مستقیم قابل حل نیست. اما می‌توانیم از یک روش ساده به نام تکرار نقطه ثابت استفاده کنیم:

1. مقدار اولیه‌ای برای x انتخاب می‌کنیم (معمولاً $x_0 = y$).
2. مقدار x را با رابطه زیر به روزرسانی می‌کنیم: $x_{n+1} = y - g(x_n)$
3. این کار را تا زمانی تکرار می‌کنیم که مقدار x به جواب دقیق همگرا شود.

چگونه $g(x)$ را انقباضی می‌کنیم؟

برای اطمینان از انقباضی بودن $g(x)$ از نرمال‌سازی طیفی استفاده می‌کنیم. این روش، ماتریس‌های وزن در طوری تنظیم می‌کند که بزرگ‌ترین مقدار ویژه آن‌ها کمتر از ۱ باشد. این کار در سه مرحله انجام می‌شود:

1. بزرگ‌ترین مقدار ویژه (σ) را محاسبه می‌کنیم.
2. وزن‌ها را نرمال‌سازی می‌کنیم:

$$W_{\text{normalized}} = \frac{W}{\sigma(W)}$$

-
3. مقدار نرمال‌شده را با یک ضریب $c < 1$ ضرب می‌کنیم.

مزایای این مدل‌ها

1. معکوس‌پذیری دقیق: به دلیل انقباضی بودن $g(x)$ ، می‌توانیم هر تبدیل را به راحتی به عقب برگردانیم.
 2. کارایی محاسباتی: نیازی به ذخیره‌سازی ماتریس‌های بزرگ یا انجام محاسبات پیچیده نیست.
 3. مدل‌سازی دقیق: این مدل‌ها می‌توانند توزیع‌های پیچیده داده را با دقت بالایی یاد بگیرند.
-

پاسخ به زیر بخش دوم :

محاسبه دترمینان ژاکوبین در مدل‌های جریان باقی‌مانده (Residual Flows)

یکی از مسائل کلیدی در مدل‌های جریان، محاسبه‌ی دترمینان ژاکوبین است که نشان‌دهنده تغییرات چگالی احتمال هنگام اعمال یک تبدیل است. در مدل‌های جریان باقی‌مانده، به جای محاسبات سنگین

و مستقیم، از روش‌های هوشمندانه‌ای مثل تخمین‌گر هاچینسون و رولت روسی استفاده می‌شود. در ادامه، این مفاهیم را ساده و روان توضیح می‌دهیم.

چرا به دترمینان ژاكوبین نیاز داریم؟

زمانی که یک تبدیل $f(x)$ روی داده‌ها اعمال می‌شود، چگالی احتمال آن‌ها تغییر می‌کند. بر اساس قضیه تغییر متغیرها، چگالی جدید به این صورت محاسبه می‌شود:

$$f(x) = x + g(x)$$

در اینجا:

$J_f(x)$ ژاكوبین تبدیل $f(x)$ است.

$\log \det(J_f(x))$ تغییر چگالی احتمال ناشی از تبدیل را نشان می‌دهد.

بنابراین برای مدل‌سازی درست، نیاز داریم $\log \det(J_f(x))$ را محاسبه کنیم.

ساختار تبدیل در جریان باقی‌مانده

در مدل‌های جریان باقی‌مانده، تبدیل $f(x)$ به این صورت تعریف می‌شود:

$$f(x) = x + g(x)$$

• ورودی است: XXX

• $g(x)$: یک تابع ساده (معمولاً یک شبکه عصبی) است که تغییرات را اعمال می‌کند.

ژاکوبین این تبدیل برابر است با:

$$J_f(x) = \frac{\partial f}{\partial x} = I + J_g(x)$$

که I ماتریس همانی است و $J_g(x)$ ژاکوبین تابع $g(x)$ است.

چگونه $\log \det(J_f(x))$ را محاسبه می‌کنیم؟

محاسبه دترمینان $J_f(x)$ به صورت مستقیم، بهویژه برای داده‌های با ابعاد بالا، بسیار هزینه‌بر است. در مدل‌های جریان باقی‌مانده، این مسئله با بازنویسی دترمینان به یک سری توانی حل می‌شود:

$$\log \det(J_f(x)) = \text{Tr}(\log(I + J_g(x))) = \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k} \text{Tr}([J_g(x)]^k)$$

اینجا Tr نشان‌دهنده **ردیف** (Trace) ماتریس است. این سری توانی به ما امکان می‌دهد بدون محاسبه مستقیم دترمینان، آن را تخمین بزنیم.

استفاده از تخمین‌گر هاچینسون

محاسبه ردیف $\text{Tr}([J_g(x)]^k)$ نیز می‌تواند هزینه‌بر باشد، بنابراین از یک روش تخمینی به نام **تخمین‌گر هاچینسون** استفاده می‌کنیم. ایده این تخمین‌گر این است که به جای محاسبه مستقیم ردیف، آن را با استفاده از ضرب داخلی محاسبه کنیم:

$$\text{Tr}(A) = \mathbb{E}_v[v^T A v]$$

۷: یک بردار تصادفی با عناصر نرمال استاندارد (میانگین صفر و واریانس یک) است. •

$v^T A v$: ضرب داخلی که تخمینی از ردیف ماتریس AAA ارائه می‌دهد. •

بنابراین هر جمله در سری توانی به این صورت تخمین زده می‌شود:

$$\text{Tr}([J_g(x)]^k) \approx \mathbb{E}_v[v^T [J_g(x)]^k v]$$

کاهش تعداد جملات سری با "رولت روسی"

محاسبه تعداد بی‌نهایت جمله در سری غیرممکن است. برای حل این مشکل، از تکنیکی به نام **رولت روسی** استفاده می‌کنیم. این روش به صورت تصادفی تعیین می‌کند که محاسبه سری در کدام جمله متوقف شود، در حالی که تضمین می‌کند تخمین نهایی بدون خطا (بدون تورش) باشد.

رولت روسی به این شکل عمل می‌کند:

1. **انتخاب یک عدد تصادفی:** تعداد جملاتی که در سری محاسبه می‌کنیم، بر اساس یک توزیع تصادفی (مثل توزیع هندسی) تعیین می‌شود.
 2. **وزن دهنده جملات:** هر جمله بر اساس احتمال حضور آن در سری، مقیاس‌بندی می‌شود.
-

مزایای این روش

1. **کاهش هزینه محاسباتی:** نیازی به محاسبه مستقیم دترمینان یا تعداد زیادی از جملات سری نیست.
2. **دقت بالا:** تخمین نهایی بدون تورش است.

۳. صرفه‌جویی در حافظه: به جای ذخیره‌سازی ماتریس‌های بزرگ، از ضرب داخلی با بردارهای تصادفی استفاده می‌شود.

جمع‌بندی

در مدل‌های جریان باقی‌مانده، دترمینان ژاکوبین با استفاده از سری توانی و تخمین‌گر هاچینسون به صورت کارآمد محاسبه می‌شود. استفاده از روش رولت روسی نیز باعث کاهش هزینه محاسباتی می‌شود، در حالی که دقت تخمین حفظ می‌شود. این رویکرد نه تنها محاسبات را ساده‌تر می‌کند، بلکه امکان استفاده از مدل‌های پیچیده‌تر را نیز فراهم می‌کند

سؤال دوم

زیر بخش اول ۱:

در مدل‌های GAN، تولیدکننده (Generator) سعی می‌کند نمونه‌هایی بسازد که نتوانند توسط تشنیوچی (تشخیص‌دهنده) به عنوان نمونه‌های تقلیبی تشخیص داده شوند. تابع زیان تولیدکننده به صورت زیر تعریف می‌شود:

$$L_G(\theta) = \mathbb{E}_{z \sim N(0, I)} [\log(1 - D_\phi(G_\theta(z)))]$$

هدف این است که نشان دهیم وقتی Discriminator به طور کامل قادر به تشخیص نمونه‌های تولید شده باشد (یعنی $\nabla_\theta D_\phi(G_\theta(z)) \approx 0$ ، گرادیان تقریباً صفر می‌شود).

گرادیان تابع زیان تولیدکننده

گرادیان تابع زیان تولیدکننده نسبت به پارامترهای آن (θ) به صورت زیر تعریف می‌شود:

$$\nabla_{\theta} L_G = \mathbb{E}_{z \sim N(0, I)} [\nabla_{\theta} \log(1 - D_{\phi}(G_{\theta}(z)))]$$

برای محاسبه گرادیان، ابتدا از قاعده زنجیره‌ای استفاده می‌کنیم:

$$\nabla_{\theta} \log(1 - D_{\phi}(G_{\theta}(z))) = \frac{-1}{1 - D_{\phi}(G_{\theta}(z))} \cdot \nabla_{\theta} D_{\phi}(G_{\theta}(z))$$

و با استفاده از گرادیان کلی به صورت زیر نوشته می‌شود:

$$\nabla_{\theta} D_{\phi}(G_{\theta}(z)) = \nabla_x D_{\phi}(x) \Big|_{x=G_{\theta}(z)} \cdot \nabla_{\theta} G_{\theta}(z)$$

پس گرادیان کلی به صورت زیر نوشته می‌شود:

$$\nabla_{\theta} L_G = \mathbb{E}_{z \sim N(0, I)} \left[\frac{-\nabla_x D_{\phi}(G_{\theta}(z)) \cdot \nabla_{\theta} G_{\theta}(z)}{1 - D_{\phi}(G_{\theta}(z))} \right]$$

$$D_{\phi}(G_{\theta}(z)) \approx 0 \quad \text{وقتی:}$$

فرض کنیم **Discriminator** بتواند نمونه‌های تولید شده را کاملاً تشخیص دهد، یعنی:

$$D_{\phi}(G_{\theta}(z)) \approx 0$$

در این حالت:

$$D\phi(G\theta(z)) \approx 1 \quad \bullet$$

(مخرج تقریباً برابر با 1 است، پس اثر آن ختنی می‌شود).

$$\nabla_x D\phi(G\theta(z)) \quad \bullet$$

اما گرادیان نیز کوچک می‌شود.

چرا؟ چون $D\phi(x)$ ازتابع سیگموئید استفاده می‌کند:

$$D\phi(x) = \sigma(h_\phi(x)) = \frac{1}{1 + e^{-h_\phi(x)}}$$

وقتی $h\phi(G\theta(z)) \ll 0$ (عدد بسیار منفی)، در این شرایط:

$$\sigma'(h_\phi(x)) = \sigma(h_\phi(x))(1 - \sigma(h_\phi(x)))$$

$D\phi(G\theta(z)) \approx 0$ که برای

$$\sigma'(h_\phi(x)) \approx 0$$

$\nabla x D\phi(G\theta(z)) \approx 0$ بنابراین خواهد بود.

نتیجه‌گیری:

در نهایت، وقتی **Discriminator** نمونه‌های تولید شده را کاملاً تشخیص دهد گرادیان $\nabla_{\theta} L_G \approx 0$ تقریباً صفر می‌شود:

$$\nabla_{\theta} L_G \approx 0$$

این به این معنی است که **Generator** دیگر اطلاعات مفیدی برای بهبود عملکرد خود دریافت نمی‌کند و آموزش آن متوقف می‌شود.

زیر بخش دوم :

تعریف تابع زیان **:Discriminator**

تابع زیان Discriminator به صورت زیر تعریف می‌شود:

$$L_D(\phi; \theta) = -\mathbb{E}_{x \sim P_{data}(x)} [\log D_\phi(x)] - \mathbb{E}_{x \sim P_\theta(x)} [\log(1 - D_\phi(x))]$$

در اینجا:

$P_{data}(x)$: توزیع داده واقعی.

$P_\theta(x)$: توزیع داده تولید شده توسط **Generator**.

هدف این است که نشان دهیم در حالت بهینه، Discriminator به صورت زیر عمل می‌کند:

$$D * (x) = \frac{P_{data}(x)}{P_{data}(x) + P_\theta(x)}$$

بهینه‌سازی **:LDL_D**

برای بهینه‌سازی LDL_D، تابع زیان را به شکل زیر بازنویسی می‌کنیم:

$$L_D(D) = - \int P_{data}(x) \log D(x) dx - \int P_\theta(x) \log(1 - D(x)) dx$$

در اینجا، $D(x)$ را به عنوان متغیر بهینه‌سازی در نظر می‌گیریم. برای پیدا کردن مقدار بهینه، باید

این عبارت را نسبت به $D(x)$ مشتق بگیریم و برابر صفر قرار دهیم.

:**مشتقگیری از LDL_D**

تابع داخل انتگرال را به صورت زیر تعریف می‌کنیم:

$$\ell(x, D(x)) = -P_{data}(x) \log D(x) - P_{\theta}(x) \log(1 - D(x))$$

با مشتقگیری نسبت به $D(x)$:

$$\frac{\partial \ell}{\partial D} = -\frac{P_{data}(x)}{D(x)} + \frac{P_{\theta}(x)}{1 - D(x)}$$

با برابر صفر قرار دادن مشتق:

$$-\frac{P_{data}(x)}{D(x)} + \frac{P_{\theta}(x)}{1 - D(x)} = 0$$

حل معادله:

با ساده‌سازی داریم:

$$\frac{P_{data}(x)}{D(x)} = \frac{P_{\theta}(x)}{1 - D(x)}$$

هر دو طرف را در $D(x)(1-D(x))D(x)(1 - D(x))$ ضرب می‌کنیم:

$$P_{data}(x) - P_{data}(x)D(x) = P_{\theta}(x)D(x)$$

بازنویسی می‌کنیم:

$$P_{data}(x) - P_{data}(x)D(x) = P_{\theta}(x)D(x)$$

: $D(x)$ جمع‌بندی ترم‌های

$$P_{data}(x) = D(x)(P_{data}(x) + P_{\theta}(x))$$

: $D(x)$ حل برای

$$D^*(x) = \frac{P_{data}(x)}{P_{data}(x) + P_{\theta}(x)}$$

زیر بخش سوم :

فرض مسئله:

اگر خروجی $D_{\phi}(x)$ برابر سیگموئید لگیت باشد:

$$D_{\phi}(x) = \sigma(h_{\phi}(x)) = \frac{1}{1 + e^{-h_{\phi}(x)}}$$

و اگر $D(x) = D^*(x)$ ، نشان دهید که:

$$h_\phi(x) = \log \frac{P_{data}(x)}{P_\theta(x)}$$

اثبات:

$(D\phi(x)D_{\phi}(x))$

$$D_\phi(x) = \frac{1}{1 + e^{-h_\phi(x)}}$$

: $D^*(x)$ با جایگذاری مقدار

$$\frac{1}{1 + e^{-h_\phi(x)}} = \frac{P_{data}(x)}{P_{data}(x) + P_\theta(x)}$$

هر دو طرف را در $1 + e^{-h_\phi(x)}$ ضرب می‌کنیم:

$$1 = \frac{P_{data}(x)}{P_{data}(x) + P_\theta(x)} (1 + e^{-h_\phi(x)})$$

بازنویسی می‌کنیم:

$$1 = \frac{P_{data}(x)}{P_{data}(x) + P_\theta(x)} + \frac{P_{data}(x)}{P_{data}(x) + P_\theta(x)} e^{-h_\phi(x)}$$

ترم اول را کم می‌کنیم:

$$1 - \frac{P_{data}(x)}{P_{data}(x) + P_{\theta}(x)} = \frac{P_{data}(x)}{P_{data}(x) + P_{\theta}(x)} e^{-h_{\phi}(x)}$$

طرف چپ را ساده می‌کنیم:

$$\frac{P_{\theta}(x)}{P_{data}(x) + P_{\theta}(x)} = \frac{P_{data}(x)}{P_{data}(x) + P_{\theta}(x)} e^{-h_{\phi}(x)}$$

هر دو طرف را در ضرب می‌کنیم:

$$P_{\theta}(x) = P_{data}(x) e^{-h_{\phi}(x)}$$

حل برای $h_{\phi}(x)$:

$$e^{-h_{\phi}(x)} = \frac{P_{\theta}(x)}{P_{data}(x)}$$

لگاریتم می‌گیریم:

$$-h_{\phi}(x) = \log \frac{P_{\theta}(x)}{P_{data}(x)}$$

و در نهایت:

$$h_{\phi}(x) = \log \frac{P_{data}(x)}{P_{\theta}(x)}$$

جمع‌بندی:

1. زیرسوال 1: وقتی $D_\phi(G_\theta(z)) \approx 0$ ، گرادیان تابع زیان **Generator** تقریباً صفر

می‌شود.

2. زیرسوال 2: **Discriminator** بهینه نسبت احتمال داده واقعی به مجموع داده واقعی و تولید شده را برمی‌گرداند.

3. زیرسوال 3: خروجی لگیت **Discriminator** معادل لگاریتم نسبت $\frac{P_{data}(x)}{P_\theta(x)}$ است.

بخش دوم - WASSERSTEIN GAN

زیر بخش اول :

با استفاده از فاصله Wasserstein برای حل مشکلات موجود در GAN‌های استاندارد طراحی شده است. این مدل هدف دارد تا پایداری فرآیند آموزش را بهبود داده، مشکل محو شدن گرادیان‌ها را رفع کند و کیفیت بالاتری از نمونه‌های تولیدی ارائه دهد.

مشکلات موجود در GAN‌های استاندارد و راه حل WGAN

مشکلات GAN‌های استاندارد:

1. ناپایداری در فرآیند آموزش:

- در GAN‌های استاندارد، آموزش به شکل یک بازی مینیمم-ماکس بین تولیدکننده و Discriminator انجام می‌شود. این ساختار معمولاً منجر به نوسانات شدید و گرادیان‌های غیرقابل پیش‌بینی می‌شود که گاهی باعث توقف یادگیری می‌شود.

2. فروپاشی حالتها (Mode Collapse)

- تولیدکننده به جای تولید داده‌های متنوع، ممکن است تنها تعداد محدودی از نمونه‌ها را که توسط Discriminator قابل قبول هستند تولید کند. این مشکل باعث کاهش تنوع نمونه‌های تولیدی می‌شود.

3. محو شدن گرادیان‌ها:

- اگر Discriminator بیش از حد قوی شود و داده‌های واقعی و تولیدی را به خوبی تفکیک کند، گرادیان‌های ارائه شده به تولیدکننده بسیار کوچک یا تقریباً صفر می‌شوند. این مسئله منجر به توقف یادگیری تولیدکننده خواهد شد.

4. اشباع شدن تابع هزینه:

- در GAN‌های استاندارد، استفاده از واگرایی JS (Jensen-Shannon) باعث اشباع شدن تابع هزینه می‌شود و اطلاعات مفیدی برای بهبود مدل ارائه نمی‌دهد.

چگونه WGAN این مشکلات را حل می‌کند؟

1. استفاده از فاصله Wasserstein به جای JS:

- فاصله Wasserstein یک معیار قوی‌تر و مشتق‌پذیر برای اندازه‌گیری تفاوت بین توزیع داده‌های واقعی و تولیدی است. این فاصله حتی در شرایطی که دو توزیع همپوشانی نداشته باشند نیز گرادیان‌های معناداری ارائه می‌دهد.

2. ایجاد گرادیان‌های پایدارتر:

- فاصله Wasserstein گرادیان‌هایی ارائه می‌دهد که تغییرات نرم و پیوسته دارند و به تولیدکننده کمک می‌کنند حتی در شرایط پیچیده یادگیری را ادامه دهد.

3. پایداری بیشتر در فرآیند آموزش:

- در WGAN، متمايزکننده (Critic) به جای Discriminator استاندارد استفاده می‌شود. متمايزکننده آموزش دیده تا فاصله Wasserstein را تخمین بزند که منجر به فرآیند آموزش پایدارتر می‌شود.

4. رفع فروپاشی حالت‌ها:

- گرادیان‌های تولید می‌کند که کل فضای داده را پوشش می‌دهند. این موضوع باعث تولید نمونه‌های متنوع‌تر و با کیفیت‌تر می‌شود.
-

زیر بخش دوم:

: نقش متمایزکننده در WGAN و تفاوت با Discriminator در GAN استاندارد نقش متمایزکننده در

در WGAN، متمایزکننده به جای دسته‌بندی داده‌ها به واقعی یا تولیدی، تفاوت توزیع داده‌های واقعی و تولیدی را اندازه‌گیری می‌کند. این تفاوت از طریق تخمین فاصله Wasserstein محاسبه می‌شود. متمایزکننده به حداقل رساندن اختلاف امتیاز داده‌های واقعی و تولیدی می‌پردازد.

تفاوت‌های کلیدی نقش متمایزکننده در WGAN و Discriminator در GAN استاندارد:

1. نوع خروجی:

- در GAN‌های استاندارد، Discriminator یک احتمال بین ۰ و ۱ ارائه می‌دهد که نشان‌دهنده واقعی بودن یا جعلی بودن داده است. این خروجی از تابع سیگموئید استفاده می‌کند.
- در WGAN، متمایزکننده یک مقدار عددی واقعی ارائه می‌دهد که نشان‌دهنده «امتیاز واقعی بودن» داده است و قادر محدودیت بازه‌ای است.

2. تابع هدف:

- در GAN استاندارد، Discriminator سعی دارد داده‌ها را به دو دسته واقعی و تولیدی تقسیم کند. اما در WGAN، متمایزکننده تفاوت میانگین امتیازات داده‌های واقعی و تولیدی را به حداقل می‌رساند.

3. شرط لیپشیتز (Lipschitz Constraint):

- متمایزکننده در WGAN باید یک تابع ۱-لیپشیتز باشد. این شرط تضمین می‌کند که تخمین فاصله Wasserstein به درستی انجام شود. معمولاً این شرط از طریق تکنیک‌هایی مانند «برش وزن» (Weight Clipping) یا «جریمه گرادیان» (Gradient Penalty) اعمال می‌شود.

4. فرآیند آموزش:

در WGAN، متمایزکننده چندین بار نسبت به تولیدکننده به روزرسانی می‌شود تا به دقت بیشتری در تخمین فاصله Wasserstein دست یابد.

مزیت‌های فاصله Wasserstein نسبت به JS و KL

1. پیوستگی و مشتق‌پذیری:

- فاصله Wasserstein حتی در شرایطی که توزیع‌ها همپوشانی نداشته باشند، پیوسته و مشتق‌پذیر است. این ویژگی گرادیان‌های همواره معنادار را تضمین می‌کند.

2. هماهنگی با هندسه داده‌ها:

- فاصله Wasserstein تفاوت‌های واقعی بین توزیع‌ها را بهتر بازتاب می‌دهد و نسبت به هندسه داده‌ها حساس‌تر است.

3. حل مشکل محو شدن گرادیان‌ها:

- با استفاده از Wasserstein، تابع هزینه دچار اشباع نمی‌شود و گرادیان‌ها همواره اطلاعات مفیدی ارائه می‌دهند.

زیر بخش سوم : معیار های ارزیابی :

Kullback-Leibler Divergence KL Divergence .1

یکی از معروف‌ترین روش‌ها برای اندازه‌گیری تفاوت بین دو توزیع است. این معیار نشان می‌دهد که چقدر یک توزیع P (توزیع واقعی) به توزیع Q (توزیع تولیدی) نزدیک است. به عبارت ساده‌تر، KL Divergence می‌گوید که اگر ما بخواهیم از توزیع Q به جای P استفاده کنیم، چقدر اطلاعات را از دست خواهیم داد.

فرمول KL Divergence به این صورت است:

$$D_{KL}(P \parallel Q) = \sum_x P(x) \log \frac{P(x)}{Q(x)}$$

یعنی برای هر داده X میزان تفاوت بین احتمال $P(x)$ و $Q(x)$ را محاسبه می‌کنیم. این جمع برای همه مقادیر X انجام می‌شود.

مزایا:

- این معیار دقیقاً اندازه‌گیری می‌کند که چقدر یک توزیع از دیگری متفاوت است.

محدودیت‌ها:

- **نامتقارن بودن:** یعنی $D_{JS}(Q \parallel P)$ معمولاً با $D_{KL}(P \parallel Q)$ یکسان نیست. این ممکن است باعث شود که مدل نتواند به خوبی تفاوت‌های موجود بین توزیع‌ها را شبیه‌سازی کند.
- **حساسیت به احتمال صفر:** اگر در توزیع Q مقدار صفر برای یک داده خاص داشته باشیم در حالی که در توزیع P مقدار غیر صفر وجود داشته باشد، نتیجه KL Divergence به بینهایت میل می‌کند. این مسئله می‌تواند باعث مشکلاتی در آموزش مدل‌های مولد شود.

(Jensen-Shannon Divergence (JS Divergence .2

JS Divergence یک نسخه اصلاح شده از KL Divergence است. این معیار برای حل مشکلاتی مانند نامتقارن بودن و حساسیت به احتمال صفر طراحی شده است. در JS Divergence، از میانگین دو توزیع PPP و QQQ برای محاسبه استفاده می‌شود تا مشکلات نامتقارن بودن حل شود.

فرمول JS به این صورت است:

که $M = \frac{1}{2}(P + Q)$ است. به عبارت ساده‌تر، این معیار میانگین فاصله KL بین هرکدام از توزیع‌ها و میانگین آن‌ها را محاسبه می‌کند.

مزایا:

- تقارن: برخلاف $D_{JS}(P // Q)$ $D_{JS}(P // Q) = D_{JS}(Q // P)$ یعنی $D_{JS}(P // Q) = D_{JS}(Q // P)$
- حساسیت کمتر به احتمال صفر: به دلیل استفاده از میانگین توزیع‌ها، حساسیت کمتری به احتمال صفر دارد و بنابراین معمولاً پایدارتر است.

محدودیت‌ها:

- گرچه JS نسبت به KL Divergence بهتر عمل می‌کند، هنوز هم ممکن است در برخی از مدل‌های پیچیده و توزیع‌های خاص مانند توزیع‌های بین‌نهایت یا نادر، چالش‌هایی ایجاد کند.

(Wasserstein) فاصله Wasserstein Distance .3

فاصله Wasserstein یا فاصله زمین به طور خاص برای اندازه‌گیری تفاوت بین دو توزیع طراحی شده است و یکی از بهترین گزینه‌ها برای مدل‌های مولد مانند **WGAN** است. این معیار می‌گوید که برای تبدیل یک توزیع P_{PP} به توزیع Q_{QQ} (یعنی برای انتقال داده‌های واقعی به داده‌های تولیدی)، حداقل هزینه‌ای که باید بپردازیم چقدر است.

فرمول فاصله Wasserstein به این صورت است:

$$W(P_r, P_g) = \inf_{\gamma \in \Gamma(P_r, P_g)} \mathbb{E}_{(x,y) \sim \gamma} [\|x - y\|]$$

این فرمول به این معنی است که برای هر جفت داده X از توزیع واقعی P_r و Y از توزیع تولیدی P_g هزینه انتقال آن‌ها محاسبه می‌شود و کمترین این هزینه‌ها در نظر گرفته می‌شود.

مزایای فاصله Wasserstein

- پایداری بهتر در آموزش: فاصله Wasserstein می‌تواند به طور موثری از مشکلاتی مانند "mode collapse" (جایی که مدل فقط نمونه‌هایی از یک دسته خاص تولید می‌کند) جلوگیری کند و فرآیند آموزش را پایدارتر کند.
- عدم حساسیت به احتمال صفر: برخلاف KL Divergence که اگر در توزیع تولیدی احتمال صفر وجود داشته باشد باعث می‌شود خسارت بین‌نهایت شود، فاصله Wasserstein چنین مشکلی ندارد و می‌تواند با داده‌هایی که مقادیر صفر دارند به راحتی کار کند.

- همگرایی بهتر: استفاده از فاصله Wasserstein در آموزش شبکه‌های مولد، ضمانت‌های همگرایی بهتری را فراهم می‌کند، یعنی مدل به تدریج به توزیع واقعی نزدیک‌تر می‌شود.

حدودیت‌ها:

- نیاز به **critic**: برای محاسبه فاصله Wasserstein، به یک "critic" نیاز داریم که مشابه به یک شبکه عصبی است. این critic باید ویژگی‌های خاصی مثل محدود بودن به توابع Lipschitz-1 را داشته باشد.
- پیچیدگی محاسباتی: محاسبه فاصله Wasserstein ممکن است از لحاظ محاسباتی سنگین‌تر از معیارهای دیگر باشد.

چرا Wasserstein در WGAN بهتر از KL و JS عمل می‌کند؟

در **WGAN** (که مخفف Wasserstein GAN است)، از فاصله Wasserstein برای محاسبه تفاوت بین توزیع واقعی و توزیع تولیدی استفاده می‌شود، به دلایل زیر:

1. **پایداری آموزش**: یکی از بزرگترین مشکلات GAN‌های سنتی (که از JS Divergence یا KL استفاده می‌کنند) این است که آموزش آن‌ها می‌تواند ناپایدار باشد و گاهی اوقات به سرعت دچار مشکلاتی مانند "mode collapse" شوند (جایی که مدل فقط نمونه‌هایی از یک دسته خاص تولید می‌کند). استفاده از فاصله Wasserstein این مشکلات را کاهش می‌دهد و فرآیند آموزش را پایدارتر می‌کند.
2. **بهبود همگرایی**: فاصله Wasserstein به مدل این اجازه را می‌دهد که به طور تدریجی به توزیع واقعی همگرا شود. این ویژگی باعث می‌شود که شبکه به آرامی یاد بگیرد که چگونه داده‌هایی واقعی شبیه‌سازی کند و از مشکلاتی مانند عدم همگرایی یا گم شدن در آموزش جلوگیری کند.
3. **عدم حساسیت به نقاط صفر**: یکی از مشکلات استفاده از KL یا JS Divergence این است که اگر توزیع تولیدی QQQ در نقاطی که توزیع واقعی PPP احتمال صفر دارد، مقدار غیر صفر داشته باشد، این معیارها به سمت بینهایت میل می‌کنند و آموزش را مختل می‌کنند. فاصله Wasserstein این مشکل را ندارد و می‌تواند به طور مؤثری این تفاوت‌ها را مدیریت کند.

جمع‌بندی

- **KL Divergence**: دقیق است ولی نامتقارن است و نسبت به احتمال صفر حساس است.
- **JS Divergence**: اصلاح‌شده از KL است و تقارن دارد، اما در برخی شرایط می‌تواند همچنان مشکلاتی داشته باشد.
- **Wasserstein Distance**: به طور مؤثر و پایدارتر از KL و JS عمل می‌کند، باعث همگرایی بهتر می‌شود، و حساسیت کمتری به احتمال صفر دارد.

در نهایت، استفاده از فاصله Wasserstein در WGAN به دلیل پایداری بهتر و همگرایی مؤثرتر نسبت به معیارهای دیگر، باعث شده که این روش در تولید داده‌های واقعی‌تر و با کیفیت‌تر موفق‌تر باشد.

زیر بخش چهارم:

1. مقدار خسارت (Loss) از Critic

در WGAN، به جای استفاده از تابع خسارت معمولی مانند GAN‌های کلاسیک (که معمولاً شامل تابع "binary" یا "log-likelihood" است)، از فاصله Wasserstein بین توزیع‌های واقعی و تولید شده استفاده می‌شود. به این ترتیب، یک critic (که در واقع یک شبکه عصبی است) برای ارزیابی فاصله Wasserstein میان این دو توزیع آموخته داده می‌شود.

- در جای استفاده از تابع خسارت معمولی مانند GAN‌های کلاسیک (که معمولاً شامل تابع "binary" یا "log-likelihood" است)، از فاصله Wasserstein بین توزیع‌های واقعی و تولید شده استفاده می‌شود. به این ترتیب، یک critic (که در واقع یک شبکه عصبی است) برای ارزیابی فاصله Wasserstein میان این دو توزیع آموخته داده می‌شود.

واقعی) و $P_g P_r$ (توزیع داده‌های تولیدی) به این صورت تعریف می‌شود:

$$W(P_r, P_g) = \sup_{\|f\|_{L^1} \leq 1} \mathbb{E}_{x \sim P_r}[f(x)] - \mathbb{E}_{x \sim P_g}[f(x)]$$

- که در آن F یک تابع است که به صورت Lipschitz-1 محدود شده باشد.
- در WGAN، هدف این است که critic خسارت (یا فاصله Wasserstein) را به حداقل برساند. بنابراین، اگر مقدار خسارت critic در حال کاهش باشد و در نهایت به مقدار ثابت نزدیک شود، این نشان‌دهنده این است که شبکه به سمت همگرایی پیش می‌رود. برای ارزیابی همگرایی، معمولاً به مقدار این خسارت در طول زمان توجه می‌شود. کاهش مقدار خسارت به معنای نزدیک‌تر شدن توزیع تولیدی به توزیع واقعی است.

2. چگالی نمونه‌ها (Sample Density)

چگالی داده‌های تولید شده توسط شبکه، یکی از روش‌های ارزیابی کیفیت و همگرایی مدل است. فرض کنید شبکه مولد (Generator) می‌خواهد داده‌هایی شبیه به داده‌های واقعی تولید کند. اگر مدل به درستی آموخته دیده باشد، توزیع داده‌های تولیدی باید مشابه به توزیع داده‌های واقعی باشد.

برای بررسی چگالی نمونه‌ها می‌توانید از روش‌های مختلفی استفاده کنید، از جمله:

- هیستوگرام‌ها یا نمودارهای پراکندگی: شما می‌توانید توزیع داده‌های تولید شده را در مقابل توزیع داده‌های واقعی از طریق نمودارهای هیستوگرام یا پراکندگی (scatter plots) مقایسه کنید. اگر شباهت زیادی بین این دو توزیع مشاهده شود، احتمالاً شبکه به همگرایی نزدیک شده است.

- مقایسه از نظر ویژگی‌ها: اگر داده‌های واقعی و تولیدی به‌طور عمدی مشابهی داشته باشند (مثلاً در ویژگی‌هایی که شبکه استخراج کرده است)، این نیز نشان‌دهنده همگرایی مدل است.

(Visual Evaluation) 3. ارزیابی بصری

در مسائل گرافیکی مانند تولید تصاویر، یکی از بهترین روش‌ها برای ارزیابی همگرایی، مشاهده بصری نمونه‌های تولید شده است. اگر شبکه به خوبی آموزش دیده باشد، تصاویر تولید شده باید شباهت زیادی به داده‌های واقعی داشته باشند و به‌طور کلی باید قابل شناسایی و معنادار باشند.

به‌ویژه در شبکه‌های تولید کننده تصاویر، اگر شبکه به‌طور مؤثری به توزیع داده‌های واقعی همگرا شود، باید ویژگی‌های مشابهی را در تصاویر تولیدی مشاهده کنید. برای مثال:

- در یک شبکه مولد تصویر، اگر تصاویر تولیدی شباهت‌های واضحی به داده‌های واقعی پیدا کنند (مانند اشیاء، رنگ‌ها، بافت‌ها و غیره)، این نشان‌دهنده پیشرفت در همگرایی است.
- Visual Turing Test: یک معیار ارزیابی بصری مشابه "تست تورینگ بصری" است، که در آن انسان‌ها می‌توانند تصاویر تولید شده را از تصاویر واقعی تشخیص دهند یا خیر. اگر انسان‌ها نتوانند تفاوت بین تصاویر واقعی و تولیدی را تشخیص دهند، این یک علامت از همگرایی خوب است.

(FID) 4. استفاده از معیارهای اضافی مانند Frechet Inception Distance

FID یکی از معیارهای معروف در ارزیابی کیفیت تصاویر تولید شده است که توسط *Inception v3* برای استخراج ویژگی‌ها از تصاویر استفاده می‌شود. این معیار، فاصله‌ای بین توزیع ویژگی‌های تصاویر واقعی و تصاویر تولید شده محاسبه می‌کند.

FID به صورت زیر تعریف می‌شود:

$$W(P_r, P_g) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{x \sim P_r}[f(x)] - \mathbb{E}_{x \sim P_g}[f(x)]$$

که در آن:

- μ_r, μ_g میانگین ویژگی‌های استخراج شده از داده‌های واقعی و تولید شده هستند.
- Σ_r, Σ_g ماتریس‌های کوواریانس ویژگی‌های داده‌های واقعی و تولید شده هستند.

کاهش مقدار FID در طول زمان به این معناست که تصاویر تولیدی به ویژگی‌های داده‌های واقعی نزدیک‌تر می‌شوند. مقدار پایین FID (مثلاً زیر 10) نشان‌دهنده این است که مدل به خوبی همگرا شده است و تصاویر تولیدی تقریباً به تصاویر واقعی شباهت دارند.

5. رصد همگرایی در Generator و Critic

در generator و WGAN، critic به طور همزمان آموزش می‌بینند، اما ممکن است این دو شبکه با سرعت‌های متفاوتی همگرا شوند. در بعضی مواقع ممکن است critic سریع‌تر از generator همگرا شود و در این صورت، توصیه می‌شود که تعداد بروزرسانی‌های critic بیشتر از generator باشد.

- اگر critic بیش از حد آموزش ببیند (یعنی تعداد بروزرسانی‌ها برای critic خیلی بیشتر از generator باشد)، ممکن است عملکرد generator ضعیف شود و به همگرایی نرسد. در چنین شرایطی، باید تعادل مناسبی بین تعداد بروزرسانی‌های critic و generator بقرار کرد.
- ناظارت بر همگرایی Critic و Generator: بررسی میزان تغییرات در وزن‌ها و تابع خسارت هر کدام از این دو مدل به شما کمک می‌کند تا متوجه شوید که آیا هر دو شبکه در حال همگرایی هستند یا خیر.

جمع‌بندی:

معیارهای همگرایی در WGAN معمولاً به ارزیابی تطابق توزیع‌های تولیدی و واقعی از طریق خسارت critic، چگالی نمونه‌ها، ارزیابی بصری، معیار FID و رصد رفتار generator و critic بستگی دارند. ترکیب این معیارها می‌تواند کمک کند تا از همگرایی مؤثر مدل اطمینان حاصل شود.

سؤال دوم-پیاده سازی مدل های : GAN

زیر بخش اول:

در این قسمت مدل **GAN** را به این ساختار طراحی کردیم :

```
transform = transforms.Compose([
    transforms.Resize(64),
    transforms.ToTensor(),
    transforms.Normalize([0.5], [0.5])
])
```

1. **تغییر اندازه:** تصاویر ورودی را به اندازه ثابت 64×64 تغییر می‌دهد تا همه تصاویر همان‌اندازه شوند و با مدل سازگار باشند.
2. **تبدیل به تنسور:** تصاویر که به صورت داده‌های خام (مثل آرایه‌های پیکسلی) هستند، به فرمت قابل پردازش برای PyTorch تبدیل می‌شوند.
3. **نرمال‌سازی:** مقادیر پیکسل‌ها را از بازه $[0, 255]$ به بازه $[-1, 1]$ تغییر می‌دهد تا یادگیری مدل سریع‌تر و پایدارتر شود.

```

class Generator(nn.Module):
    def __init__(self, latent_dim=100):
        super(Generator, self).__init__()
        ngf = 64
        self.model = nn.Sequential(
            nn.ConvTranspose2d(latent_dim, ngf * 8, kernel_size=4, stride=1, padding=0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),

            nn.ConvTranspose2d(ngf * 8, ngf * 4, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),

            nn.ConvTranspose2d(ngf * 4, ngf * 2, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),

            nn.ConvTranspose2d(ngf * 2, ngf, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),

            nn.ConvTranspose2d(ngf, 1, kernel_size=4, stride=2, padding=1, bias=False),
            nn.Tanh()
        )

    def forward(self, input):
        return self.model(input)

```

```

class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        ndf = 64
        self.model = nn.Sequential([
            nn.Conv2d(1, ndf, kernel_size=4, stride=2, padding=1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(ndf, ndf * 2, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(ndf * 2, ndf * 4, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(ndf * 4, ndf * 8, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(ndf * 8, 1, kernel_size=4, stride=1, padding=0, bias=False),
            nn.Sigmoid()
        ])

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

```

این کلاس اصلی است:

```
💡
class GAN:
    def __init__(self, latent_dim=100, lr=0.0002):
        self.generator = Generator(latent_dim).to(device)
        self.discriminator = Discriminator().to(device)
        self.latent_dim = latent_dim

        self.criterion = nn.BCELoss()

        self.optimizer_G = torch.optim.Adam(self.generator.parameters(), lr=lr, betas=(0.5, 0.999))
        self.optimizer_D = torch.optim.Adam(self.discriminator.parameters(), lr=lr, betas=(0.5, 0.999))

# Subsection 2: ConvTranspose2d Layer Explanation
```

در این قسمت مدل را train کرده ایم.

```

for epoch in range(epochs):
    for i, (imgs, _) in enumerate(dataloader):
        real_imgs = imgs.to(device)

        valid = torch.ones(real_imgs.size(0), 1, device=device)
        fake = torch.zeros(real_imgs.size(0), 1, device=device)

        gan.optimizer_G.zero_grad()
        z = torch.randn(real_imgs.size(0), latent_dim, 1, 1, device=device)
        gen_imgs = gan.generator(z)
        g_loss = gan.criterion(gan.discriminator(gen_imgs), valid)
        g_loss.backward()
        gan.optimizer_G.step()

        gan.optimizer_D.zero_grad()
        real_loss = gan.criterion(gan.discriminator(real_imgs), valid)
        fake_loss = gan.criterion(gan.discriminator(gen_imgs.detach()), fake)
        d_loss = (real_loss + fake_loss) / 2

        d_loss.backward()
        gan.optimizer_D.step()

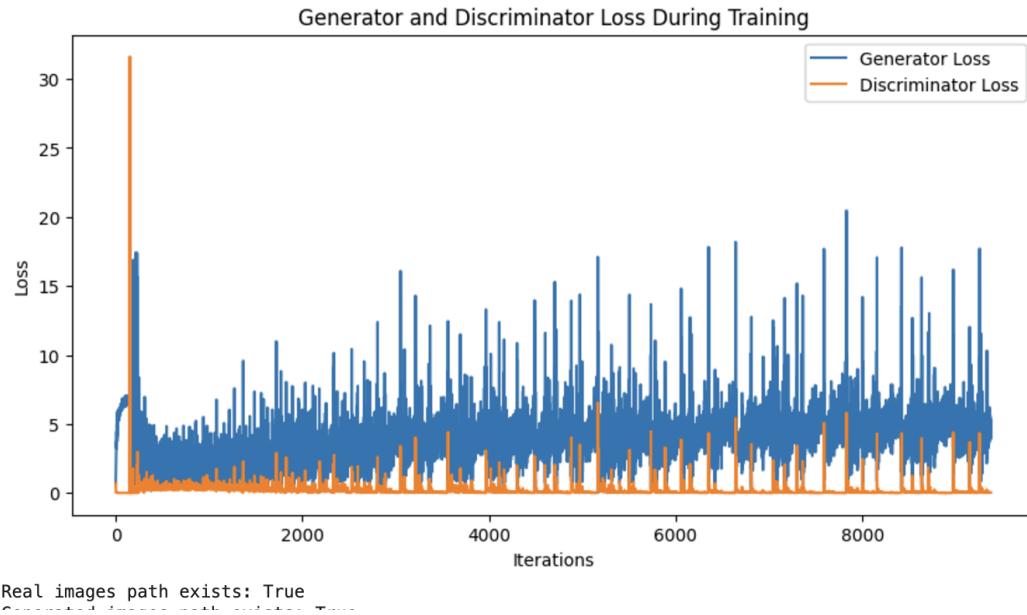
        G_losses.append(g_loss.item())
        D_losses.append(d_loss.item())

    if i % 100 == 0:
        print(f"[Epoch {epoch+1}/{epochs}] [Batch {i}/{len(dataloader)}] "
              f"[D loss: {d_loss.item():.4f}] [G loss: {g_loss.item():.4f}]")

    with torch.no_grad():
        gen_imgs = gan.generator(fixed_noise).cpu()
        grid = torchvision.utils.make_grid(gen_imgs, nrow=5, normalize=True)
        np_grid = grid.numpy()
        plt.figure(figsize=(10, 10))
        plt.imshow(np.transpose(np_grid, (1, 2, 0)))
        plt.title(f'Epoch {epoch+1}')
        plt.axis('off')
        plt.show()

```

مقدار Loss:



بخش: پیاده‌سازی شبکه GAN روی داده‌های Fashion MNIST

1. Generator (مولد):

- **وظیفه:** تولید تصاویر جدید از یک بردار نویز تصادفی.
- **ساختار:** شامل چندین لایه **ConvTranspose2d** است که به تدریج بردار نویز ورودی را بزرگنمایی کرده و به یک تصویر با ابعاد 64×64 times 64×64 تبدیل می‌کند.
- **توابع فعال‌سازی:**
 - از **ReLU** در لایه‌های میانی برای فعال‌سازی استفاده می‌شود.
 - در لایه خروجی، از **Tanh** استفاده می‌شود تا مقادیر پیکسل‌های تصویر تولیدی به بازه $[-1, 1]$ نرمال شوند.

2. Discriminator (تشخیص‌دهنده):

- **وظیفه:** شناسایی تصاویر واقعی از تصاویر تولید شده توسط مولد.
- **ساختار:** شامل چندین لایه **Conv2d** است که تصویر ورودی را به تدریج کوچکتر کرده و ویژگی‌های مهم آن را استخراج می‌کند.

- توابع فعال‌سازی:

- از **LeakyReLU** برای جلوگیری از مشکل ناپدید شدن گرادیان‌ها استفاده می‌شود. این تابع در مقایسه با ReLU اجازه می‌دهد مقدار کمی از گرادیان در بخش منفی عبور کند.
 - در لایه خروجی، از **Sigmoid** استفاده می‌شود تا احتمال واقعی بودن تصویر بین ۰ و ۱ محاسبه شود.
-

GAN .3 (مدیریت مدل):

- **وظیفه:** مدیریت شبکه‌های مولد و تشخیص‌دهنده، تنظیم توابع هزینه و بهینه‌سازی.
 - **جزئیات:**
 - از **Binary Cross-Entropy Loss** برای محاسبه خطاهای استفاده می‌شود.
 - **مولد:** تلاش می‌کند خطای تشخیص‌دهنده را کاهش دهد تا تصاویر تولیدی واقعی‌تر به نظر برسند.
 - **تشخیص‌دهنده:** تلاش می‌کند تصاویر واقعی و جعلی را با دقت بیشتری از هم تشخیص دهد.
 - از بهینه‌ساز **Adam** استفاده شده است که به دلیل تنظیمات خاص خود (مانند نرخ یادگیری تطبیقی) پایداری و سرعت بالایی در آموزش فراهم می‌کند.
-

زیر بخش دوم :

(Generator) در مولد ConvTranspose2d توضیح لایه

1. نقش لایه ConvTranspose2d در شبکه‌های عصبی:

- **کاربرد اصلی:**
- لایه **ConvTranspose2d**، که گاهی با نام "Transposed Convolution" یا "Deconvolution" شناخته می‌شود، برای بزرگنمایی تصاویر یا داده‌ها استفاده می‌شود.
-

• تفاوت با Conv2d :

در حالی که Conv2d برای کاهش ابعاد و استخراج ویژگی‌ها استفاده می‌شود، ConvTranspose2d به طور معکوس عمل کرده و ابعاد داده‌های ورودی را افزایش می‌دهد.

• نقش در GAN‌ها:

در شبکه‌های مولد GAN، این لایه برای تبدیل بردار نویز (بردار تصادفی کوچک) به تصاویر بزرگ‌تر با ابعاد دلخواه (مثل 64×64 پیکسل) به کار می‌رود.

2. تغییر ابعاد داده‌های ورودی توسط ConvTranspose2d :

• نحوه کار:

این لایه از یک هسته کانولوشنی (Kernel) استفاده می‌کند و با گسترش دادن هر موقعیت در ورودی، ابعاد تصویر را افزایش می‌دهد.

• مثال:

ورودی با ابعاد times\ 88×8 times 44×4\ 44×4، سپس times 11×1\ 11×1 (بردار نویز) به ترتیب به 88×8 و در نهایت به times 6464×64\ 6464×64 افزایش می‌یابد.

3. تاثیر پارامترهای ConvTranspose2d بر خروجی:

چند پارامتر مهم این لایه عبارتند از:

1. Kernel Size (اندازه هسته):

تعیین اندازه ناحیه‌ای که در هر گام برای کانولوشن استفاده می‌شود.

2. Stride (گام):

تعیین می‌کند که چقدر لایه ورودی در هر گام جایه‌جا شود. مقدار بزرگ‌تر stride منجر به افزایش سریع‌تر ابعاد خروجی می‌شود.

3. Padding (حاشیه):

اضافه کردن صفر به اطراف داده‌های ورودی برای کنترل ابعاد خروجی.

4. Output Padding (حاشیه خروجی):

برای تنظیم دقیق ابعاد خروجی در صورت نیاز.

4. فرمول محاسبه ابعاد خروجی:

ابعاد خروجی **ConvTranspose2d** برای لایه $W_{out} = W_{in} \times H_{in}$ و $H_{out} = H_{in} \times W_{in}$ به صورت زیر محاسبه می‌شوند:

$$H_{out} = (H_{in} - 1) \times \text{stride} - 2 \times \text{padding} + \text{kernel_size} + \text{output_padding}$$

$$W_{out} = (W_{in} - 1) \times \text{stride} - 2 \times \text{padding} + \text{kernel_size} + \text{output_padding}$$

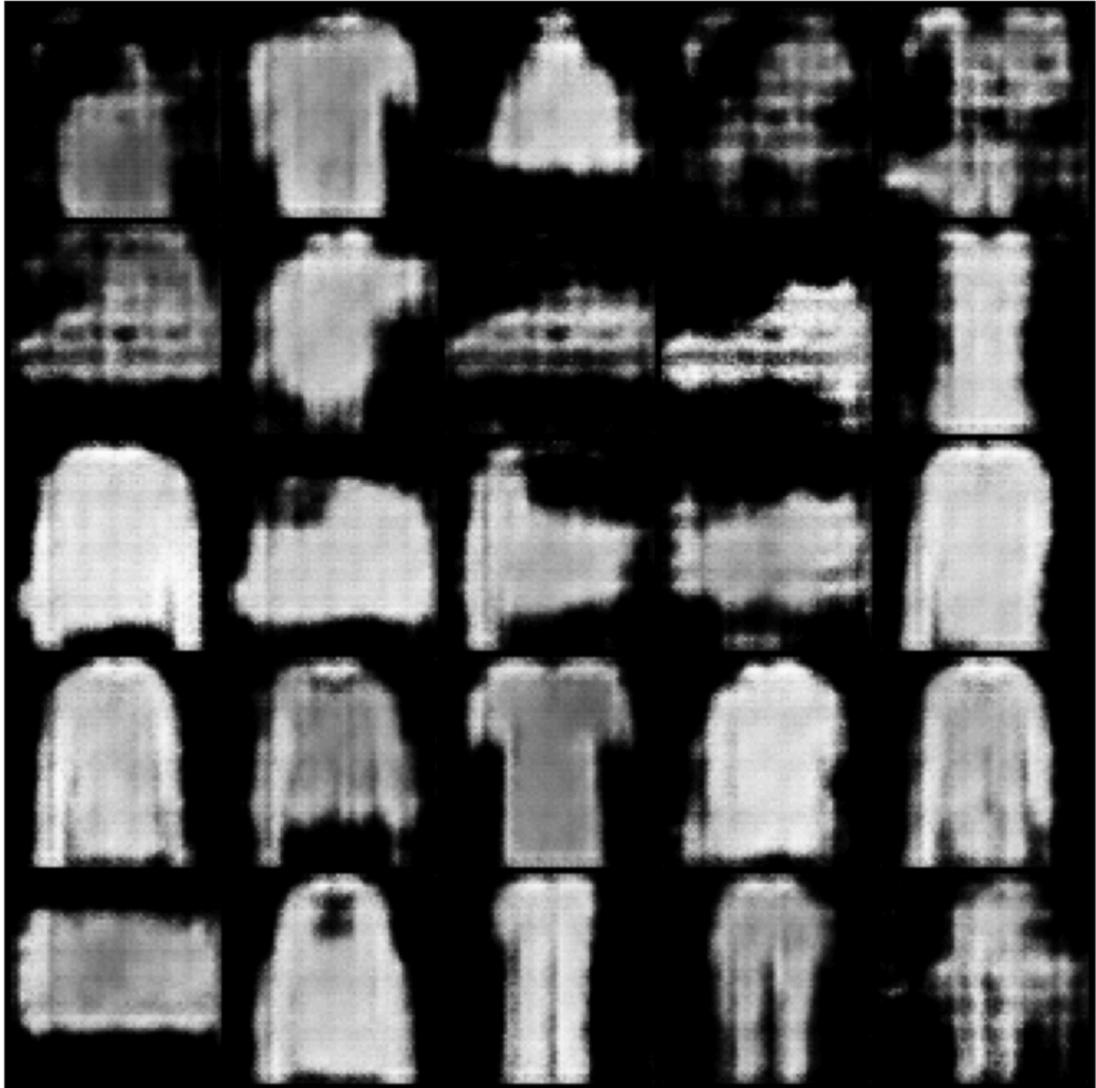
مقادیر:

- H_{in}, W_{in} : ارتفاع و عرض ورودی.
 - stride : گام کانولوشن.
 - padding : حاشیه اضافه شده به ورودی.
 - kernel_size : اندازه هسته.
 - output_padding : حاشیه نهایی برای تنظیم ابعاد خروجی.
-

زیر بخش سوم :

```
[Epoch 1/10] [Batch 0/938] [D loss: 0.6795] [G loss: 0.6267]
[Epoch 1/10] [Batch 100/938] [D loss: 0.0013] [G loss: 6.2292]
[Epoch 1/10] [Batch 200/938] [D loss: 0.0377] [G loss: 13.4333]
[Epoch 1/10] [Batch 300/938] [D loss: 0.0849] [G loss: 2.5734]
[Epoch 1/10] [Batch 400/938] [D loss: 0.3171] [G loss: 1.1969]
[Epoch 1/10] [Batch 500/938] [D loss: 0.6135] [G loss: 0.4932]
[Epoch 1/10] [Batch 600/938] [D loss: 0.2599] [G loss: 1.8977]
[Epoch 1/10] [Batch 700/938] [D loss: 0.3166] [G loss: 2.0946]
[Epoch 1/10] [Batch 800/938] [D loss: 0.3741] [G loss: 1.2295]
[Epoch 1/10] [Batch 900/938] [D loss: 0.4286] [G loss: 3.1867]
```

Epoch 1



```
[Epoch 5/10] [Batch 0/938] [D loss: 0.0929] [G loss: 2.7903]
[Epoch 5/10] [Batch 100/938] [D loss: 0.0644] [G loss: 5.1395]
[Epoch 5/10] [Batch 200/938] [D loss: 0.0356] [G loss: 9.1352]
[Epoch 5/10] [Batch 300/938] [D loss: 0.0611] [G loss: 3.5596]
[Epoch 5/10] [Batch 400/938] [D loss: 0.2820] [G loss: 5.9128]
[Epoch 5/10] [Batch 500/938] [D loss: 0.0286] [G loss: 3.4545]
[Epoch 5/10] [Batch 600/938] [D loss: 0.0779] [G loss: 3.1224]
[Epoch 5/10] [Batch 700/938] [D loss: 0.0149] [G loss: 5.2746]
[Epoch 5/10] [Batch 800/938] [D loss: 0.0673] [G loss: 4.9205]
[Epoch 5/10] [Batch 900/938] [D loss: 0.0418] [G loss: 4.1381]
```

Epoch 5



```
[Epoch 10/10] [Batch 0/938] [D loss: 0.1201] [G loss: 2.8691]
[Epoch 10/10] [Batch 100/938] [D loss: 0.2373] [G loss: 1.9228]
[Epoch 10/10] [Batch 200/938] [D loss: 0.2901] [G loss: 1.7430]
[Epoch 10/10] [Batch 300/938] [D loss: 0.0168] [G loss: 4.3346]
[Epoch 10/10] [Batch 400/938] [D loss: 0.0110] [G loss: 4.9771]
[Epoch 10/10] [Batch 500/938] [D loss: 0.0115] [G loss: 6.4616]
[Epoch 10/10] [Batch 600/938] [D loss: 0.1117] [G loss: 4.1411]
[Epoch 10/10] [Batch 700/938] [D loss: 0.0411] [G loss: 3.6966]
[Epoch 10/10] [Batch 800/938] [D loss: 0.0461] [G loss: 3.6132]
[Epoch 10/10] [Batch 900/938] [D loss: 0.0333] [G loss: 4.4033]
```

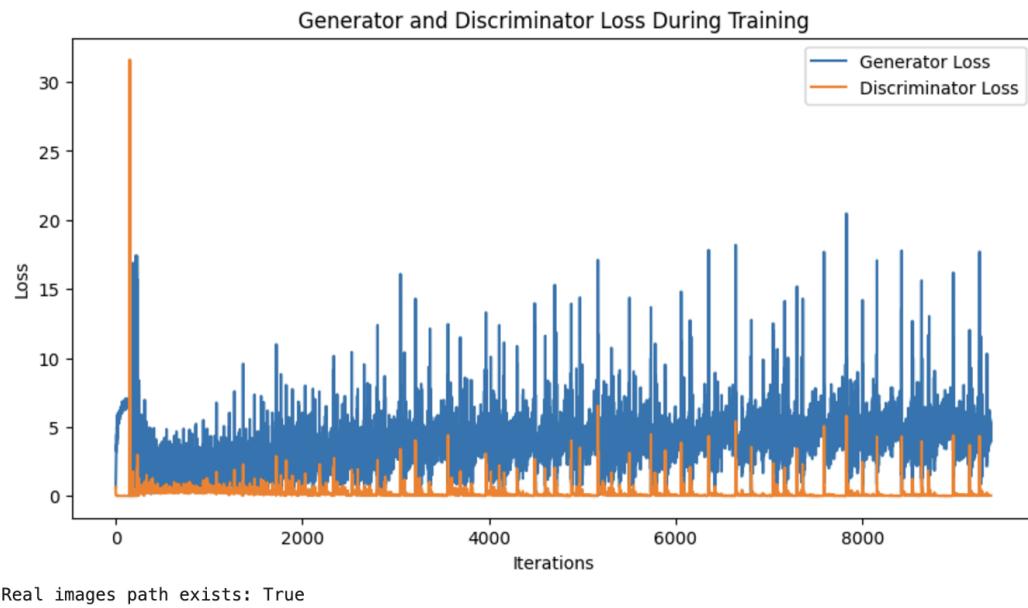
Epoch 10



۲ تولید و نمایش نمونه‌ها:

برای بررسی کیفیت خروجی در طول آموزش، از یک بردار نویز ثابت (**fixed_noise**) استفاده می‌کنیم. این کار امکان مقایسه مستقیم خروجی‌های Generator در دوره‌های مختلف را فراهم می‌کند.

زیربخش چهارم : رسم نمودارهای هزینه Generator و Discriminator



تحلیل نمودارها:

در نمودار میبینیم که در ابتدا در iteration های ابتدایی که در epoch اول رخ داده بود generator loss در حال کاهش یافتن بود که به معنای تولید تصاویر بهتر توسط تولید کننده بود

و همین طور Discriminator در ابتدا لاس زیادی داشت و در هر مرحله داشت کم میشد که این نشان دهنده ای این بود که Discriminator در حال یادگیری برای تشخیص دادن داده های فیک و واقعی میباشد

در ادامه میبینیم که هر دو دارند با هم دیگر رقابت میکنند و در پایان Discriminator لاس پایین تری دارد

و generator لاس بالاتری داشت که این نشان دهنده ای این است که چون داده ها و ساختار خوبی نداریم تقریباً داده های خوبی را تولید نکرده است که البته تصاویر تولیدی برای انسان قابل تشخیص بودند

در حالت کلی هر چه جلو تر رفته ایم Discriminator قوی تر شده و همین طور Generator هم تصاویر بهتری را تولید کرده است اما Discriminator از آن قوی تر شده و بر آن غلبه کرده است .

زیر بخش پنجم : - توضیحات FID

معیار Frechet Inception Distance FID

یکی از معتبرترین معیارها برای ارزیابی کیفیت و تنوع تصاویر تولیدی توسعه شبکه‌های مولد نظری GAN‌ها است. این معیار به طور خاص برای بررسی شباهت میان توزیع تصاویر واقعی و تصاویر تولید شده است و از ویژگی‌های استخراج شده توسط یک مدل پیش‌آموزش داده شده مانند Inception Network استفاده می‌کند.

۱. استفاده از FID در ارزیابی کیفیت تصاویر تولیدی

کاربرد اصلی FID:

- سنجش کیفیت تصاویر تولیدی: FID شباهت توزیع ویژگی‌های تصاویر واقعی و تولیدی را اندازه‌گیری می‌کند. این ویژگی‌ها از لایه‌های میانی یک شبکه عمیق (ممکن‌باشد شبکه Inception) استخراج می‌شوند که اطلاعات سطح بالا مانند ساختار، شکل و بافت تصویر را در خود دارند.
- تشخیص تنوع تصاویر: علاوه بر کیفیت، FID میزان پوشش توزیع داده‌های واقعی را ارزیابی می‌کند. به عبارت دیگر، اگر Generator فقط تعداد محدودی از نمونه‌ها را تولید کند، FID این مشکل را تشخیص می‌دهد.

مزایای FID نسبت به معیارهای دیگر:

۱. تشخیص جزئیات: FID به دلیل استفاده از ویژگی‌های سطح بالای استخراج شده از تصاویر، می‌تواند تفاوت‌های ظریف در کیفیت و ساختار تصاویر را تشخیص دهد.
۲. انعطاف‌پذیری: این معیار برای انواع مختلف مجموعه داده‌ها قابل استفاده است و در برابر تغییر مقیاس یا چرخش تصاویر حساس نیست.
۳. کمتر بودن حساسیت به حالت‌های تکراری یا کپی: اگر تصاویر تکراری تولید کند (مانند مشکل "فروپاشی مد")، FID به وضوح این نقص را نشان می‌دهد.

۲. فرمول ریاضی FID و نحوه عملکرد آن

فرمول FID:

$$FID(P_r, P_g) = \|\mu_r - \mu_g\|^2 + Tr\left(\Sigma_r + \Sigma_g - 2(\Sigma_r \Sigma_g)^{1/2}\right)$$

اجزای فرمول:

$$\mu_r, \mu_g .1$$

- بردارهای میانگین ویژگی‌های استخراج شده از تصاویر واقعی (PrP_rPr) و تولیدی (PgP_gPg).
- این بخش نشان‌دهنده تفاوت میانگین ویژگی‌های تصاویر واقعی و تولیدی است.

$$\Sigma_r, \Sigma_g .2$$

- ماتریس‌های کوواریانس ویژگی‌های تصاویر واقعی و تولیدی.
- این بخش نشان‌دهنده میزان گوناگونی و همبستگی ویژگی‌ها در دو توزیع است.

$$:Tr .3$$

- نشان‌دهنده تریس ماتریس (جمع عناصر قطری ماتریس).

توضیح ریاضی فرمول:

$$.1 \text{. تفاوت میانگین‌ها } (\|\mu_r - \mu_g\|^2)$$

- اولین بخش فرمول فاصله اقلیدسی (Euclidean) بین میانگین‌های ویژگی‌های تصاویر واقعی و تولیدی را اندازه‌گیری می‌کند.
- هرچه این مقدار کمتر باشد، میانگین ویژگی‌ها در دو توزیع به هم نزدیک‌تر است.

$$.2 \text{. تفاوت کوواریانس‌ها } (Tr(\Sigma_r + \Sigma_g - 2(\Sigma_r \Sigma_g)^{1/2}))$$

- این بخش پیچیده‌تر است و تفاوت در ساختار و گوناگونی داده‌های واقعی و تولیدی را بررسی می‌کند.

$$(\Sigma_r \Sigma_g)^{1/2} \text{ نشان‌دهنده میزان همپوشانی بین دو توزیع کوواریانس محاسبه جذر ماتریس است.}$$

- هرچه ماتریس‌های کوواریانس واقعی و تولیدی بیشتر به هم شبیه باشند، این مقدار کمتر خواهد بود.

۳. نحوه عملکرد FID در عمل

مراحل محاسبه :FID

1. استخراج ویژگی‌ها:

- تصاویر واقعی و تولیدی از طریق یک شبکه از پیشآموزش داده شده (مانند v3 Inception) عبور داده می‌شوند.
- ویژگی‌ها معمولاً از یکی از لایه‌های انتهایی (مثلًا قبل از لایه Softmax) استخراج می‌شوند.

2. محاسبه بردار میانگین (μ) و ماتریس کوواریانس (Σ):

- برای هر مجموعه (تصاویر واقعی و تولیدی)، بردار میانگین و ماتریس کوواریانس ویژگی‌ها محاسبه می‌شود.

3. اعمال فرمول FID:

- با استفاده از مقادیر محاسبه شده، مقدار FID نهایی تعیین می‌شود.

۴. تحلیل مفهوم FID

FID کوچکتر:

مقدار کمتر FID نشان‌دهنده شباهت بیشتر بین توزیع ویژگی‌های تصاویر واقعی و تولیدی است. به عبارت دیگر، Generator توانسته تصاویر واقعی‌تر تولید کند.

FID بزرگتر:

مقدار بزرگتر FID به معنای تفاوت بیشتر در کیفیت و تنوع تصاویر تولیدی نسبت به تصاویر واقعی است. این می‌تواند به دلیل مشکلاتی مانند "فروپاشی مد" یا تولید تصاویر بی‌کیفیت باشد.

۶. کاربرد FID در GAN‌ها:

1. ارزیابی کیفیت در مراحل مختلف آموزش:

- در مراحل ابتدایی، FID معمولاً بالا است چون Generator تصاویر با کیفیت پایین تولید می‌کند.
- با پیشرفت آموزش، مقدار FID کاهش می‌یابد که نشان‌دهنده بهبود کیفیت تصاویر تولیدی است.

2. مقایسه مدل‌ها:

FID به عنوان معیاری استاندارد برای مقایسه کیفیت تصاویر تولیدی در معماری‌های مختلف به کار می‌رود.

زیر بخش ششم :

در این قسمت برای هر کدام از epoch‌ها FID مربوطه به آن را محاسبه خواهیم کرد :

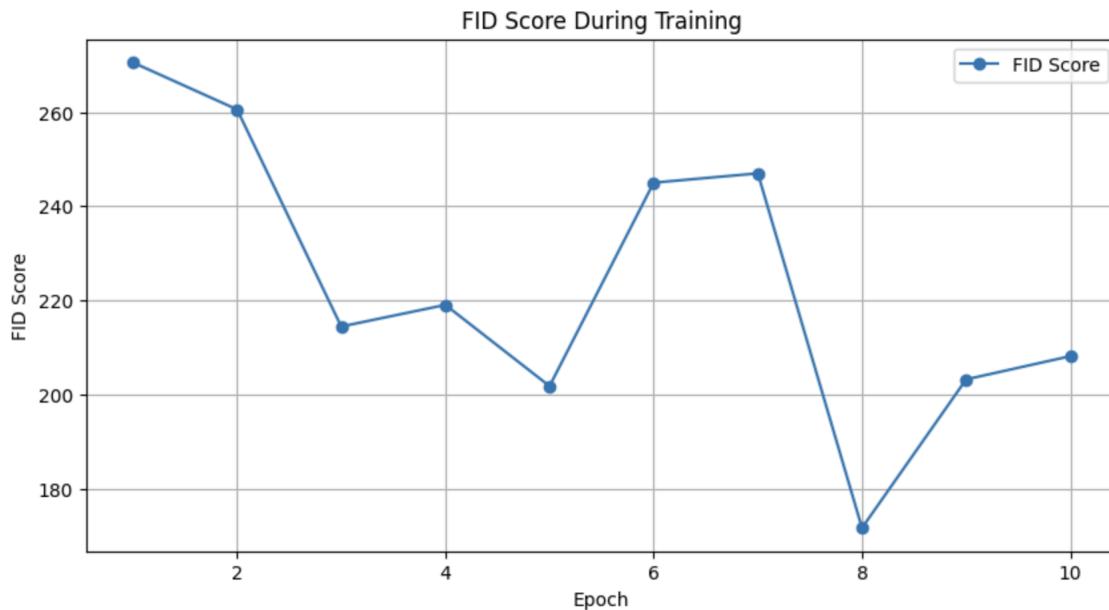
```
# calculate FID score for the epoch
fid_value = fid_score.calculate_fid_given_paths([real_path, generated_epoch_dir], batch_size, device, dims=2048)
FID_scores.append(fid_value)
print(f"FID Score at Epoch {epoch+1}: {fid_value}")
```

قبل از آن باید تعدادی تصویر برای FID ساخته شود.

```
generated_epoch_dir = os.path.join(generated_path_epoch, f'epoch_{epoch+1}')
os.makedirs(generated_epoch_dir, exist_ok=True)
with torch.no_grad():
    for j in range(0, len(dataloader.dataset), batch_size):
        noise = torch.randn(batch_size, latent_dim, 1, 1, device=device)
        gen_imgs = gan.generator(noise).cpu()
        for k, img in enumerate(gen_imgs):
            torchvision.utils.save_image(img, os.path.join(generated_epoch_dir, f'img_{j+k}.png'), normalize=True)
```

و در پایان آن را نمایش داده ایم .

```
plt.figure(figsize=(10, 5))
plt.title("FID Score During Training")
plt.plot(range(1, epochs+1), FID_scores, marker='o', label="FID Score")
plt.xlabel("Epoch")
plt.ylabel("FID Score")
plt.legend()
plt.grid()
plt.show()
```



نمودار FID نوسان دارد که احتمالاً به خاطر ماهیت MinMax بودن مدل های GAN میباشد .
 ما روی ۱۶ تصویر از هر کدام از دو توزیع اصلی و fake هر بار fid را محاسبه کرده بودیم که به همین
 خاطر احتمالاً نتوانسته مقدار FID دقیق تر را تخمین بزند
 که اگر بر روی تعداد بیشتری از داده ها FID را اعمال کرده بودیم به نتایجه ی بهتری میرسیدیم که زمان
 خیلی زیاد تری هم میبرد.
 چون زمان بسیار زیادی میبرد صرفا در هر ایپاک ۱۶ داده را با هم مقایسه کردیم.

مراجع

- Goodfellow, I., et al. (2014). Generative Adversarial Nets. *Advances in Neural Information Processing Systems*, 27, 2672-2680.
- Salimans, T., et al. (2016). Improved Techniques for Training GANs. *Advances in Neural Information Processing Systems*, 29, 2234-2242.
- Arjovsky, M., Chintala, S., & Bottou, L. (2017). Wasserstein GAN. arXiv preprint arXiv:1701.07875.
- Behrmann, J., Grathwohl, W., Chen, R. T. Q., Duvenaud, D., Jacobsen, J.-H. (2019). **Invertible Residual Networks**. *International Conference on Machine Learning (ICML)*.
- Chen, R. T. Q., Behrmann, J., Duvenaud, D., Jacobsen, J.-H. (2019). **Residual Flows for Invertible Generative Modeling**. *Advances in Neural Information Processing Systems (NeurIPS)*.
- Hutchinson, M. F. (1990). **A Stochastic Estimator of the Trace of the Influence Matrix for Laplacian Smoothing Splines**. *Communications in Statistics - Simulation and Computation*.