به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

مدل‌های مولد عمیق

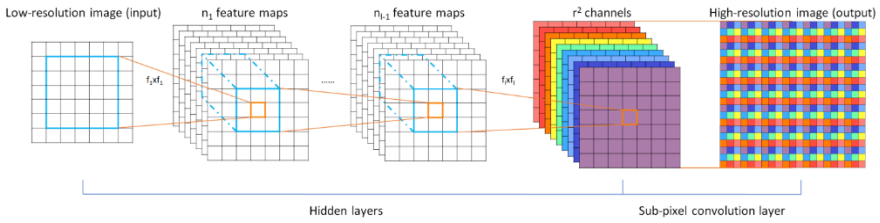
تمرین شماره دوم – قسمت دوم

نام و نام خانوادگی	کیانا هوشانفر
شماره دانشجویی	۸۱۰۱۰۱۳۶۱
تاریخ ارسال گزارش	

فهرست گزارش سوالات

سوال ۱ – Generative Adversarial Networks (GANs)

A.



شکل ۱ - **PixelShuffle**

PixelShuffle اغلب با وظایف با وضوح فوق العاده همراه است که شامل افزایش وضوح تصویر است. این مفهوم برای مقابله با چالش ارتقاء تصاویر با وضوح پایین و در عین حال حفظ یا بهبود جزئیات معرفی شد. هدف اصلی PixelShuffle افزایش وضوح مکانی یک تصویر با ترتیب مجدد و ترکیب اطلاعات پیکسل های مجاور است.

- تانسور ورودی: فرض کنید یک تانسور دارید که تصویری با وضوح پایین را نشان می دهد.
- تقسیم کانال ها: تانسور به گروه هایی از کانال ها تقسیم می شود.
- مرتب سازی مجدد پیکسل ها: پیکسل های هر گروه برای ایجاد یک تانسور خروجی با وضوح بالاتر مرتب می شوند.
- ترکیب اطلاعات: اطلاعات از تصویر اصلی با وضوح پایین به روشی ساختاریافته ترکیب می‌شوند تا تصویری با وضوح بالاتر تشکیل دهند.

ایده اصلی پشت PixelShuffle استفاده از اطلاعات موجود در تصویر با وضوح پایین و بازسازی هوشمندانه یک نسخه با وضوح بالاتر است. این به ویژه در برنامه‌هایی مانند وضوح فوق‌العاده تک تصویر، که در آن افزایش کیفیت یک تصویر منفرد ضروری است، مفید است.

در مورد تأثیر عملگر PixelShuffle بر روی یک شبکه عصبی، می‌تواند در کارهایی که وضوح فضایی افزایش یافته مورد نیاز است، مفید باشد. با گنجاندن PixelShuffle در یک معماری شبکه، مدل می‌تواند یاد بگیرد که وضوح فوق‌العاده را به طور مؤثرتری انجام دهد. این می تواند منجر به بهبود عملکرد در کارهایی مانند بازسازی تصویر شود، جایی که شبکه نیاز به تولید تصاویر با وضوح بالا از ورودی های با وضوح پایین دارد.

- PixelShuffle چگونه کار می کند:

PixelShuffle شکلی از sub-pixel convolution است و با مرتب کردن مجدد عناصر یک تنسور عمل می کند. به طور خاص، تصویری با وضوح پایین می گیرد و وضوح فضایی آن را با چیدمان مجدد عناصر در هر کانال برای تشکیل تصویر با وضوح بالاتر افزایش می دهد. این عملیات معمولاً به عنوان بخشی از معماری شبکه عصبی، اغلب در شبکه مولد GAN ها استفاده می شود

- نحوه عملکرد PixelShuffle:

- تنسور ورودی:

تانسور ورودی دارای شکل (اندازه_ بچ، ارتفاع، عرض، کانال) است.

- بازآرایی/Rearrangement :

تنسور برای افزایش وضوح فضایی آن تغییر شکل و مرتب شده است. به عنوان مثال، اگر هدف افزایش مقیاس با ضریب ۲ باشد، هر کانال به پیکسل‌های فرعی غیر همپوشانی تقسیم می‌شود و این پیکسل‌های فرعی برای تشکیل تصویر بزرگ‌تر بازآرایی می‌شوند.

- تنسور خروجی:

تانسور خروجی دارای شکل (اندازه_ بچ ، ۲ * ارتفاع، ۲ * عرض، کانال ها / ۴) است.

- هدف و مبدا:

انگیزه اصلی پشت PixelShuffle، رسیدگی به چالش تولید تصاویر با وضوح بالا در مدل‌های تولیدی است. این به مدل ها اجازه می دهد تا وضوح فضایی را بدون وارد کردن هزینه محاسباتی قابل توجهی افزایش دهند.

PixelShuffle به عنوان بخشی از معماری های یادگیری عمیق برای افزایش قابلیت های مدل های تولیدی معرفی شد. با استفاده از این عملیات، مدل‌ها می‌توانند تصاویر دقیق‌تر و واقعی‌تری تولید کنند، که در برنامه‌هایی مانند وضوح تصویر فوق‌العاده، انتقال سبک، و سایر کارهایی که نیاز به سنتز تصویر با کیفیت بالا است، بسیار مهم است.

- تأثیر بر GAN ها:

در زمینه GAN ها، PixelShuffle اغلب در شبکه generator استفاده می شود تا feature maps با وضوح پایین تولید شده توسط لایه های اولیه را ارتقا دهد. این به ویژه در کارهایی مانند -image

to-image translation یا تولید تصاویر واقعی با وضوح بالا از ورودی های با وضوح پایین مفید است.

اثر PixelShuffle در GAN ها:

- افزایش وضوح:

PixelShuffle به generator اجازه می‌دهد تا وضوح فضایی feature maps را به طور موثر افزایش دهد و به GAN کمک می‌کند تا تصاویر دقیق‌تر و از نظر بصری جذاب‌تری تولید کند.

- کاهش پیچیدگی محاسباتی:

در مقایسه با روش‌های درون‌یابی سنتی، PixelShuffle یک روش کارآمد برای ارتقاء تصاویر بدون افزایش قابل توجهی پیچیدگی محاسباتی مدل ارائه می‌کند.

- واقع گرایی بهبود یافته:

feature map با وضوح بالاتر، generator را قادر می‌سازد تا جزئیات دقیق‌تری را در تصاویر تولید شده ثبت کند و به خروجی واقعی‌تر و از نظر بصری متقاعدکننده‌تر کمک کند.

.B

❖ پیاده سازی Generator:

ژنراتور یک بردار نویز تصادفی را به یک تصویر synthetic از طریق یک سری لایه‌های کاملاً متصل و کانولوشن تبدیل می‌کند، که شامل نرمال‌سازی دسته‌ای و فعال‌سازی ReLU برای استخراج ویژگی و افزایش وضوح فضایی است. هدف این معماری ثبت و تولید الگوهای پیچیده، تولید تصاویری با ظاهر واقعی در طول آموزش در یک سناریوی مدل‌سازی مولد است.

```
class Generator(torch.nn.Module):
    def __init__(self, z_dim=64, num_channels=1):
        super().__init__()
        self.z_dim = z_dim
        # YOUR CODE STARTS HERE
        self.layers = torch.nn.Sequential(
            torch.nn.Linear(z_dim, 512),
            torch.nn.BatchNorm1d(512),
            ReLU(),

            torch.nn.Linear(512, 64 * 7 * 7),
            torch.nn.BatchNorm1d(64 * 7 * 7),
            ReLU(),
```

```
            Reshape(64, 7, 7),

            torch.nn.PixelShuffle(2),

            torch.nn.Conv2d(16 , 32, kernel_size=3, padding=1),
            torch.nn.BatchNorm2d(32),
            ReLU(),

            torch.nn.PixelShuffle(2),

            torch.nn.Conv2d(8 , num_channels, kernel_size=3, padding=1)
        )
        # YOUR CODE ENDS HERE
    def forward(self, z):
        return self.layers(z)
```

این کد برای تولید تصاویر در شبکه GAN طراحی شده است. ژنراتور یک بردار نویز تصادفی z را به عنوان ورودی می گیرد که معمولاً از یک توزیع نرمال با بعد z_dim گرفته می شود. معماری شامل چندین لایه است که به صورت متوالی سازماندهی شده اند. (معماری با توجه به ساختارهای داده شده پیاده سازی شده است.)

بخش اول شامل لایه های کاملاً متصل با نرمال سازی دسته ای و توابع فعال سازی ReLU است. این لایه ها بردار نویز ورودی را به نمایشی با ابعاد بالاتر تبدیل می کنند. لایه Reshape خروجی را به یک تنسور سه بعدی با ابعاد ۶۴ (کانال)، ۷ (ارتفاع) و ۷ (عرض) تغییر شکل می دهد. متعاقباً، یک عملیات ترکیبی پیکسل اعمال می شود که به طور موثر وضوح فضایی را افزایش می دهد.

پس از نمونه برداری، دو لایه کانولوشن با نرمال سازی دسته ای و فعال سازی ReLU به کار گرفته می شوند. این لایه ها بیشتر ویژگی های فضایی داده ها را اصلاح می کنند. لایه کانولوشنال نهایی تصویر خروجی را با تعداد مشخص کانال (معمولاً ۱ برای مقیاس خاکستری یا ۳ برای RGB) تولید می کند.

- ❖ پیاده سازی Discriminator:

هدف این شبکه طبقه‌بندی تصاویر ورودی به عنوان واقعی یا تولید شده با تبدیل آنها از طریق یک سری لایه‌های کانولوشنال و fully connected با فعال‌سازی ReLU است. این معماری به گونه‌ای طراحی شده است که ویژگی‌های سلسله مراتبی و نمایش‌های فضایی را به تصویر بکشد، و Discriminator را قادر می‌سازد تا به طور مؤثر بین تصاویر واقعی و تولید شده در طول آموزش تمایز قائل شود.

```
class Discriminator(torch.nn.Module):
    def __init__(self, num_channels=1):
        super().__init__()
        # YOUR CODE STARTS HERE
        self.layers = torch.nn.Sequential(
            torch.nn.Conv2d(1, 32, kernel_size=4, stride=2, padding=1),
            ReLU(),

            torch.nn.Conv2d(32, 64, kernel_size=4, stride=2, padding=1),
            ReLU(),

            Reshape(64 * 7 * 7),
            torch.nn.Linear(64 * 7 * 7, 512),
            ReLU(),

            torch.nn.Linear(512, 1),
            Reshape()
        )
        # YOUR CODE ENDS HERE
    def forward(self, x):
        return self.layers(x)
```

کد ارائه شده برای آموزش Discriminator، استفاده می شود. Discriminator برای تمایز بین تصاویر واقعی و تولید شده طراحی شده است. معماری شامل یک سری لایه های کانولوشن و کاملاً متصل است که به صورت متوالی سازماندهی شده اند.

اولین لایه کانولوشن یک تنسور تصویر ورودی x با یک کانال (مقیاس خاکستری) می گیرد و یک کانولوشن دوبعدی با ۳۲ فیلتر، اندازه ۴ در ۴، گام ۲ و stride ۱ را اعمال می کند. پس از آن ReLU به عنوان تابع فعال سازی. لایه کانولوشن دوم به طور مشابه یک convolution دوبعدی با ۶۴ فیلتر را اعمال می کند و مجدداً با یک فعال سازی ReLU دنبال می شود. این لایه های کانولوشن، استخراج ویژگی را انجام می دهند و ابعاد فضایی را پایین می آورند.

پس از لایه های کانولوشن، یک لایه Reshape خروجی را به یک تنسور یک بعدی مسطح می کند و دو لایه fully connected به دنبال آن می آیند. اولین لایه fully connected، تنسور مسطح را به یک فضای ویژگی ۵۱۲ بعدی نگاشت می‌کند و یک فعال‌سازی ReLU اعمال می‌شود. لایه نهایی ابعاد را به ۱ کاهش می دهد و یک خروجی اسکالر تولید می کند که نشان دهنده اطمینان Discriminator به اینکه ورودی یک تصویر واقعی است. یک لایه Reshape نهایی اعمال می شود تا اطمینان حاصل شود که خروجی شکل مناسبی دارد. **(قسمت هایی که در سوال با ؟! نوشته شده بود در کد بالا با رنگ زرد مشخص شده است.)**

.C

$$L_{\text{discriminator}}(\phi;\theta) \approx -\frac{1}{m}\sum_{i=1}^{m}\log D_\phi(\mathbf{x}^{(i)}) - \frac{1}{m}\sum_{i=1}^{m}\log\left(1 - D_\phi(G_\theta(\mathbf{z}^{(i)}))\right)$$

$$L^{\text{ns}}_{\text{generator}}(\phi;\theta) \approx -\frac{1}{m}\sum_{i=1}^{m}\log D_\phi(G_\theta(\mathbf{z}^{(i)}))$$

for batch-size m, and batches of *real-data* $\mathbf{x}^{(i)} \sim p_{\text{data}}(\mathbf{x})$ and *fake-data* $\mathbf{z}^{(i)} \sim \mathcal{N}(0, I)$

شکل ۲ – تابع **loss**

تابع loss را با توجه به روابط بالا به شکل زیر پیاده سازی میکنیم:

```
    def loss_nonsaturating(self, x_real, *, device):
        # Generate random noise vector
        batch_size = x_real.shape[0]
        z = torch.randn(batch_size, self.g.z_dim, device=device)

        # Generate fake images using the generator
        x_fake = self.g(z)

        # Compute discriminator outputs for real and fake images
        d_real = self.d(x_real)
        d_fake = self.d(x_fake)

        # Clone and detach fake images for non-training use
        x_fake_no_train = x_fake.clone().detach()
        d_fake_no_train = self.d(x_fake_no_train)

        # Calculate nonsaturating GAN losses
        # Discriminator loss penalizing real images being classified as fake and
vice versa
        d_loss = -torch.mean(torch.nn.functional.logsigmoid(d_real)) -
torch.mean(torch.nn.functional.logsigmoid(-d_fake_no_train))

        # Generator loss encouraging the discriminator to classify fake images as
real
        g_loss = -torch.mean(torch.nn.functional.logsigmoid(d_fake))

        return d_loss, g_loss
```

کد شامل یک مولد self.g و یک Discriminator، self.d است. GAN برای آموزش داده های تصویری طراحی شده است، و به طور خاص، روش loss_nonssaturating اتلاف GAN غیراشباع را محاسبه می

کند، که معمولا برای آموزش GAN ها استفاده می شود. در این روش، دسته‌ای از تصاویر واقعی x_real با یک بردار نویز به‌طور تصادفی تولید شده z جفت می‌شوند که به عنوان ورودی به generator عمل می‌کند. مولد تصاویر مصنوعی x_fake تولید می کند، و هر دو تصاویر واقعی و جعلی برای بدست آوردن خروجی های مربوطه به Discriminator وارد می شوند. سپس خروجی های Discriminator برای تصاویر واقعی d_real و تصاویر fake شبیه سازی شده و جدا شده d_fake_no_train برای محاسبه loss تفکیک کننده استفاده می شود. به طور همزمان، تلفات مولد بر اساس خروجی Discriminator برای تصاویر تولید شده d_fake محاسبه می شود. هدف آموزش مولد برای تولید تصاویر واقع گرایانه است که می تواند Discriminator را فریب دهد، در حالی که هدف Discriminator تشخیص تصاویر واقعی و تولید شده است. تلفات Discriminator و مولد محاسبه‌شده برگردانده می‌شوند و مبنایی برای به‌روزرسانی پارامترهای GAN در طول آموزش فراهم می‌کنند. استفاده از log sigmoid در GAN ها متداول است زیرا با نگاشت خروجی های Discriminatorها در محدوده بین ۰ و ۱ به تثبیت آموزش کمک می کند و همچنین به حل مشکل ناپدید شدن گرادیان کمک می کند.

.D

کد ترین را به شکل زیر تکمیل می‌کنیم:

```
    def gan_step(self, x_real, y_real):
        assert len(self.optimizers) == 2

        generator, discriminator = self.model.g, self.model.d
        g_optimizer, d_optimizer = self.optimizers

        # Compute discriminator and generator losses
        discriminator_loss, generator_loss =
self.model.loss_nonsaturating(x_real, device=self.device)

        # Update generator
        g_optimizer.zero_grad()
        generator_loss.backward(retain_graph=True)
        g_optimizer.step()

        # Update discriminator
        d_optimizer.zero_grad()
        discriminator_loss.backward()
        d_optimizer.step()

        # Update the optimizers in the class
        self.optimizers = [g_optimizer, d_optimizer]
```

```
        # Append losses to the lists
        self.D_losses.append(discriminator_loss.item())
        self.G_losses.append(generator_loss.item())

        return {"discriminator_loss": discriminator_loss, "generator_loss":
generator_loss}

    def train(self, train_loader, reinit=False):
        global_step = 0

        # train model from scratch
        if reinit:
            # OPTIONAL: Initialize your model if needed
            pass

        # train models for multiple epochs
        with tqdm(total=int(self.iter_max)) as pbar:
            for epoch in range(self.iter_max):
                for batch_idx, (x, y) in enumerate(train_loader):
                    x_real = x.to(self.device)
                    y_real = y.to(self.device)

                    loss, summaries = self.gan_step(x_real, y_real)
                    global_step += 1
                    pbar.update(1)
                    self.checkpoint_and_log(global_step, loss, summaries)

                    if global_step >= self.iter_max:
                        break
```

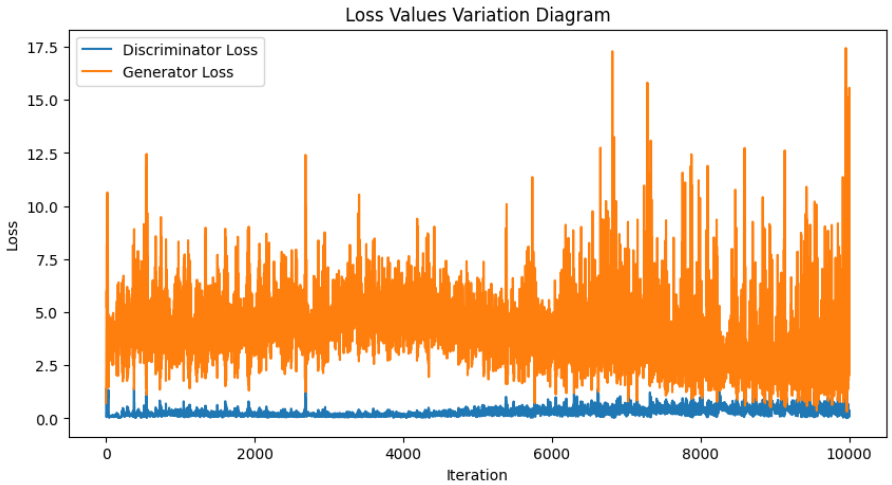
حلقه آموزشی از طریق دسته‌ای از داده‌ها از train_loader مشخص شده برای حداکثر تعداد تکرار iter_max تکرار می‌شود. فرآیند آموزش شامل به روز رسانی پارامترهای مولد و discriminator بر اساس تلفات محاسبه شده در طول هر تکرار است.

تابع gan_step یک مرحله از فرآیند آموزش GAN است. مجموعه ای از داده های واقعی x_real و y_real را به عنوان ورودی می گیرد. مدل های مولد و discriminator از مدل GAN به دست می آیند و بهینه سازهای مربوطه آنها بازیابی می شوند. سپس تلفات GAN که در بالا آن را پیاده سازی کردیم با استفاده از روش « loss_nonsaturating » مدل GAN محاسبه می‌شود. پارامترهای مولد و discriminator با استفاده از مراحل پس انتشار و بهینه ساز به روز می شوند. تلفات در لیست‌ها ثبت می‌شوند.

در قسمت "Train" کل فرآیند آموزش را هماهنگ می کند داده‌های آموزشی تکرار می‌کند تا زمانی که به تعداد مشخصی از تکرارها iter_max برسد. برای هر دسته از داده ها، تابع gan_step برای به روز رسانی پارامترهای مدل فراخوانی می شود. حلقه آموزش تا زمانی ادامه می یابد که به حداکثر تعداد تکرار برسد یا کاربر آموزش را قطع کند.

.E

.a



شکل ۳ – نمودار تغییرات **loss**

از نمودار بالا متوجه می شویم که در مراحل اولیه آموزش، تلفات generator زیاد است زیرا generator هنوز در حال یادگیری تولید نمونه های واقعی است. از سوی دیگر، تلفات discriminator در ابتدا نسبتاً کم است زیرا به راحتی می توان بین نمونه های تولید شده ضعیف و داده های واقعی تمایز قائل شد. در ادامه روند آموزش، تلفات مولد کاهش می یابد، و تلفات discriminator نیز کاهش می یابد، که نشان می دهد discriminator در تفکیک نمونه های واقعی و تولید شده مهارت بیشتری پیدا می کند. در آخر مرحله training هم تلفات مولد و discriminator تثبیت می‌شوند. همگرایی نشان می‌دهد که مولد نمونه‌هایی را ایجاد می‌کند که به اندازه کافی واقع‌بینانه هستند تا discriminator را فریب دهند، و discriminator در تمایز بین نمونه‌های واقعی و تولید شده بسیار دقیق است.

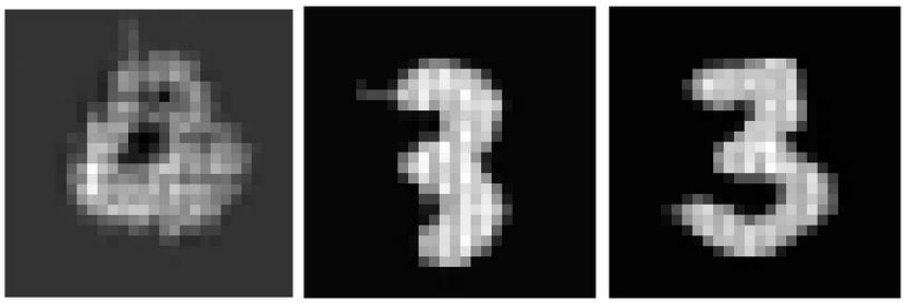
.b

خروجی نهایی مدل:

شکل ۴ - نمونه خروجی تولیدی به وسیله شبکه **GAN**

خروجی مدل به ازای ۳ ایپاک:

شکل ۵ - خروجی مدل به ازای ۳ ایپاک ابتدایی میانی و نهایی (از سمت چپ به ترتیب)



شکل ۶ - خروجی مدل به ازای ۳ ایپاک ابتدایی میانی و نهایی

مشاهده می کنیم که مدل به خوبی داده ها را تولید کرده است.

.c

FID (Fréchet Inception Distance) معیاری است که معمولاً برای ارزیابی کیفیت تصاویر تولید شده، به‌ویژه در زمینه شبکه‌های GAN استفاده می‌شود. شباهت بین دو مجموعه داده از تصاویر را با مقایسه آمار نمایش ویژگی های آنها اندازه گیری می کند.

$$FID = \|\mu_r - \mu_g\|^2 + T_r(\Sigma_r + \Sigma_g - 2(\Sigma_r\Sigma_g)^{1/2})$$

شکل ۷ – فرمول محاسبه **fid**

۱. Fréchet Distance: معیاری برای تشابه بین دو توزیع احتمال در یک فضای متریک است. در مورد FID، بین نمایش ویژگی های تصاویر واقعی و تولید شده محاسبه می شود.

۲. Inception Score: امتیاز Inception برای ارزیابی کیفیت و تنوع تصاویر تولید شده استفاده می شود. بر اساس این ایده است که تصاویر تولید شده خوب باید هم واقعی (با کیفیت بالا) و هم متنوع باشند. Inception Score از شبکه عصبی Inception v3 برای استخراج ویژگی ها از تصاویر استفاده می کند و سپس امتیاز را بر اساس توزیع این ویژگی ها محاسبه می کند.

امتیاز FID این دو جنبه را ترکیب می کند تا معیار جامع تری از کیفیت تصویر را در مقایسه با استفاده از تفاوت های پیکسلی ارائه دهد. **امتیاز FID پایین تر نشان دهنده شباهت بهتر بین تصاویر واقعی و تولید شده است که نشان دهنده کیفیت و واقع گرایی بالاتر است.**

برای پیاده سازی آن از کتابخانه pytorch-fid استفاده میکنیم.

```
!pip install --q pytorch-fid
```

در ادامه ۱۰۰۰۰ سمپل generate میکنیم و آن ها را ذخیره میکنیم.

```
num_samples = 10000
num_latents = 64
batch_size = 2048  #the only number that have reasonable output
# Loop through batches
for batch_idx in range(num_samples // batch_size):
    # Generate random noise for the generator
    z_fake = torch.randn(batch_size, num_latents, device=device)
    # Generate fake images using the generator model
    with torch.no_grad():
        fake_images = model.g(z_fake).detach().cpu()
    # Save each generated image to the output directory
    for i in range(batch_size):
        # Construct the path for saving the image
```

```
        image_path = os.path.join(output_directory, f"generated_image_{batch_idx
* batch_size + i + 1}.png")

        # Save the generated image
        save_image(fake_images[i], image_path)

print("Done")
```

همچنین داده های تست mnist را نیز در فولدری ذخیره میکنیم.

```
# Directory to save real images
mnist_directory = "mnist_images"
os.makedirs(mnist_directory, exist_ok=True)
# Load real images from the MNIST dataset
mnist_loader = torch.utils.data.DataLoader(
    datasets.MNIST(root="./data", train=False, download=True,
transform=transforms.ToTensor()),
    batch_size=10000,  # there are 10,000 images in the test set
    shuffle=True)
# Get a batch of real images
real_images, _ = next(iter(mnist_loader))
real_images = real_images.to(device)
# Loop through the real images and save each one
for i in range(num_samples):
    # Construct the path for saving the real image
    image_path = os.path.join(mnist_directory, f"real_image_{i + 1}.png")

    # Save the real image
    save_image(real_images[i], image_path)
print("Done")
```

با اجرای دستورات زیر مقدار fid را بدست می آوریم:

```
from pytorch_fid.fid_score import calculate_fid_given_paths

# Paths to generated images and real images
path_to_generated_images = '/content/generated_images'
path_to_real_images = '/content/mnist_images'

# Calculate the FID score
fid_value = calculate_fid_given_paths([path_to_generated_images,
path_to_real_images],
                                      batch_size=50,
                                      device='cuda',
                                      dims=2048)
print(f'FID score: {fid_value}')
```

خروجی:

FID score: 45.657116388698

هرچقدر این مقدار کمتر باشد تصاویر واقعی و تولید شده شباهت بیشتری دارند.

راه دیگر محاسبه fid نوشتن مراحل آن از پایه است:

```
import torch
from torchvision.models import inception_v3
from torchvision import transforms
from scipy.linalg import sqrtm

def calculate_fid(generator, x_real, device, num_samples=10, batch_size=100):
    # Load InceptionV3 model
    inception_model = inception_v3(pretrained=True,
transform_input=False).to(device).eval()

    # Calculate the number of batches
    num_batches = num_samples // batch_size

    # Initialize an empty list to store fake activations
    fake_activations_list = []

    # Generate fake images in batches
    for _ in range(num_batches):
        with torch.no_grad():
            generator.eval()
            fake_images = generator(torch.randn(batch_size, generator.z_dim,
device=device))
        fake_activations_list.append(calculate_inception_activations(fake_images,
inception_model, device))

    # Concatenate activations from all batches
    fake_activations = np.concatenate(fake_activations_list, axis=0)

    # Preprocess and calculate Inception activations for real images
    real_activations = calculate_inception_activations(x_real, inception_model,
device)

    # Calculate FID score
    fid_score = calculate_fid_score(real_activations, fake_activations)

    return fid_score
```

```
def calculate_inception_activations(images, inception_model, device):
    # Resize images for InceptionV3
    transform = transforms.Compose([
        transforms.ToPILImage(),
        transforms.Resize((299, 299)),
        transforms.Grayscale(num_output_channels=3),  # Convert to RGB if working
with grayscale images
        transforms.ToTensor(),
    ])

    # Duplicate the single channel to create an RGB-like tensor
    images = torch.cat([transform(img).unsqueeze(0) for img in images], dim=0)  #
Add a batch dimension
    images = images.to(device)

    # Get InceptionV3 activations
    activations = inception_model(images)

    # Detach the tensor before calling numpy()
    return activations.detach().cpu().numpy()


def calculate_fid_score(real_activations, fake_activations):
    # Calculate mean and covariance for real and fake activations
    mu_real, sigma_real = real_activations.mean(axis=0), np.cov(real_activations,
rowvar=False)
    mu_fake, sigma_fake = fake_activations.mean(axis=0), np.cov(fake_activations,
rowvar=False)

    # Calculate FID score
    diff = mu_real - mu_fake
    cov_mean = sqrtm(sigma_real.dot(sigma_fake))
    fid_score = np.trace(sigma_real + sigma_fake - 2 * cov_mean) + np.dot(diff,
diff.T)

    return fid_score
```

این تابع یک مدل مولد ("مولد")، مجموعه ای از تصاویر واقعی x_real ، و پارامترهای اختیاری برای تعداد نمونه های تولید شده و اندازه دسته ای را می گیرد. از مدل InceptionV3 از کتابخانه Torchvision برای استخراج فعال‌سازی ویژگی‌ها از تصاویر واقعی و تولید شده استفاده می‌کند. تصاویر تولید شده به صورت دسته‌ای تولید می‌شوند و فعال‌سازی‌های آن‌ها به هم متصل می‌شوند. سپس امتیاز FID بر اساس میانگین و کوواریانس این فعال‌سازی‌ها محاسبه می‌شود. علاوه بر این، توابع کمکی

calculate_inception_activations و calculate_fid_score به ترتیب برای پیش پردازش تصاویر و محاسبه امتیازات FID تعریف شده اند. امتیاز FID عدم تشابه بین توزیع ویژگی‌های تصویر واقعی و تولید شده را کمیت می‌دهد و معیار کمی از کیفیت تصاویر تولید شده را ارائه می‌دهد.

در اینجا از روش اول fid محاسبه شده است ولی روش دوم نیز همان نتیجه روش اول را می دهد ولی رم بیشتری را مصرف میکند و بهینه نیست.

.F

جدول ۱ – مقایسه بین مدل های موجود

نام مدل GAN	چه مشکلی را برطرف میکند؟	چگونه این مشکل را مرتفع میکند؟
Wasserstein GAN (WGAN)	آموزش GAN های استاندارد ممکن است دشوار باشد و از مشکلاتی مانند mode collapse و بی ثباتی آموزش رنج ببرند. vanishing gradients	WGAN از یک تابع هزینه متفاوت استفاده می کند که فرآیند آموزشی پایدارتری را ارائه می دهد. به جای Jensen-Shannon divergence یا Kullback-Leibler divergence که در GAN های معمولی استفاده می شود، WGAN فاصله Wasserstein را به حداقل می رساند. این تغییر منجر به بهبود همگرایی و شیب معنی‌دارتر در طول آموزش می‌شود و مسائلی مانند mode collapse را کاهش می‌دهد. Wasserstein به عنوان فاصله Earth-Mover's نیز شناخته می شود، به عنوان معیاری معنادارتر از تفاوت بین توزیع های تولید شده و واقعی است. Discriminator در WGAN برای به حداقل رساندن فاصله Wasserstein بین توزیع داده های واقعی و تولید شده آموزش دیده است. مولد برای به حداکثر رساندن فاصله Wasserstein بین توزیع داده واقعی و توزیع داده تولید شده آموزش دیده است.
PG-GAN	آموزش تصاویر با وضوح بالا با GAN ها می تواند چالش برانگیز باشد و تصاویر تولید شده ممکن است فاقد جزئیات و انسجام باشند. این مدل بر بهبود کیفیت و تنوع تصاویر تولید شده تمرکز دارد. (تصاویر واضح تر و واقعی تر با طیف وسیع)	آموزش را با وضوح تصویر کوچک آغاز می کند و به تدریج آنها را در طول زمان افزایش می دهد. این به مدل اجازه می دهد ابتدا ویژگی های درشت را یاد بگیرد و سپس به تدریج جزئیات را اصلاح کند. این رشد تدریجی به تثبیت تمرین کمک می کند، از mode collapse جلوگیری می کند و منجر به تولید تصویر با کیفیت و وضوح بالا می شود. مولد را برای تولید تصاویر با جزئیات بیشتر آموزش می دهد. هر مرحله آموزش، لایه‌های جدیدی را به شبکه‌های مولد و discriminator معرفی می‌کند و به آنها اجازه می‌دهد تا ویژگی‌های پیچیده‌تری را ثبت کنند.

Big GAN	موضوع بی ثباتی آموزش در GAN ها و دستیابی دشوار به همگرایی و عملکرد پایدار تولید تصاویر با کیفیت بالا با GAN ها اغلب به مقدار زیادی از منابع محاسباتی نیاز دارد، که آن را برای محققان با دسترسی محدود به سخت افزار قدرتمند چالش برانگیز می کند.	اندازه و پیچیدگی مدل را افزایش می دهد و به آن اجازه می دهد تا الگوها و جزئیات پیچیده تری را در داده ها ثبت کند. با استفاده از سخت افزار قدرتمندتر یا محاسبات توزیع شده، Big GAN به بهترین نتایج از نظر کیفیت تصویر دست می یابد. از یک clipped gradient penalty, شده استفاده می کند که gradientsها را در حین آموزش محدود می کند و از قدرتمند شدن بیش از discriminator و بی ثبات کردن شبکه جلوگیری می کند. BigGAN همچنین استفاده از mini-batch discrimination را معرفی می‌کند، که در آن از چندین mini-batch از داده‌ها برای به‌روزرسانی discrimination استفاده می‌شود و احتمال mode collapse را کاهش می‌دهد.
Style GAN	GANهای معمولی ممکن است کنترل دقیقی بر سبک‌های تصاویر تولید شده ارائه نکنند، و دستکاری ویژگی‌های بصری خاص را به چالش می‌کشد.	مفهوم بازنمایی‌های disentangled را معرفی می‌کند، که در آن سبک (مانند رنگ، بافت) و محتوا (مانند ساختار، اشیاء) یک تصویر به طور مستقل دستکاری می‌شوند. این به کاربران اجازه می دهد تا سبک بصری محتوای تولید شده را با دقت بیشتری کنترل کنند و رویکردی شهودی و انعطاف پذیرتر برای ترکیب تصویر ارائه دهند. StyleGAN مفهوم استفاده از " style vectors " را برای کنترل ظاهر تصاویر تولید شده معرفی می کند. تصاویر را به عنوان ترکیبی از بردارهای style جداگانه نشان می دهد که هر یک جنبه خاصی از سبک تصویر را کنترل می کند. مولد یاد می گیرد که این بردارها را به تصاویر واقعی نگاشت کند، که امکان کنترل بیشتر بر ظاهر تصویر را فراهم می کند. همچنین از یک تکنیک جدید به نام " attention mapping" استفاده می کند که به مولد اجازه می دهد هنگام ایجاد جزئیات روی مناطق خاصی از تصویر تمرکز کند.

.G

.a

هدف در آموزش WGAN ها به حداقل رساندن این تابع هزینه است. مهم است که توجه داشته باشید که خروجی discriminator مانند GAN های معمولی بین ۰ و ۱ محدود نمی شود. در عوض، می تواند هر ارزش واقعی را داشته باشد.

```
class WGAN(nn.Module):
    def __init__(self, z_dim=2):
        super().__init__()
        self.z_dim = z_dim
        self.g = Generator(z_dim=z_dim)
        self.d = Discriminator()

    def loss_wasserstein_gp(self, x_real, *, device):
        '''
        Input Arguments:

        - x_real (torch.Tensor): training data samples (batch_size, 1, 28, 28)
        - device (torch.device): 'cpu' by default
        Returns:
        - d_loss (torch.Tensor): Wasserstein GAN discriminator loss
        - g_loss (torch.Tensor): Wasserstein GAN generator loss
        '''
        # Generate random noise vector
        batch_size = x_real.shape[0]  #64 according to the algorithm
        z = torch.randn(batch_size, self.g.z_dim, device=device)
        # Generate fake images using the generator
        x_fake = self.g(z)

        # Compute discriminator outputs for real and fake images
        d_real = self.d(x_real)
        d_fake = self.d(x_fake)
        d_fake_clone =  self.d(x_fake.clone().detach())


        # Wasserstein GAN discriminator loss
        d_loss = d_fake_clone.mean() - d_real.mean()

        # Wasserstein GAN generator loss
        g_loss = -d_fake.mean()

        return d_loss, g_loss
```

شبکه Wasserstein (WGAN) از یک تابع هزینه متفاوت در مقایسه با GANهای معمولی استفاده می‌کنند. نوآوری کلیدی در WGAN ها استفاده از فاصله Wasserstein به عنوان معیاری برای تفاوت بین توزیع نمونه های تولید شده و توزیع داده های واقعی است.

	Discriminator/Critic	Generator
GAN	$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(x^{(i)}\right) + \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right)\right]$	$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(D\left(G\left(z^{(i)}\right)\right)\right)$
WGAN	$\nabla_w \frac{1}{m} \sum_{i=1}^{m} \left[f\left(x^{(i)}\right) - f\left(G\left(z^{(i)}\right)\right)\right]$	$\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} f\left(G\left(z^{(i)}\right)\right)$

شکل ۸ – تابع های هزینه در دو مدل GAN

Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

Require: : α, the learning rate. c, the clipping parameter. m, the batch size. n_{critic}, the number of iterations of the critic per generator iteration.
Require: : w_0, initial critic parameters. θ_0, initial generator's parameters.

while θ has not converged **do**
  for $t = 0, \ldots, n_{\text{critic}}$ **do**
    Sample $\{x^{(i)}\}_{i=1}^{m} \sim \mathbb{P}_r$ a batch from the real data.
    Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
    $g_w \leftarrow \nabla_w \left[\frac{1}{m}\sum_{i=1}^{m} f_w(x^{(i)}) - \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))\right]$
    $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$
    $w \leftarrow \text{clip}(w, -c, c)$
  end for
  Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
  $g_\theta \leftarrow -\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))$
  $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$
end while

شکل ۹ – الگوریتم WGAN

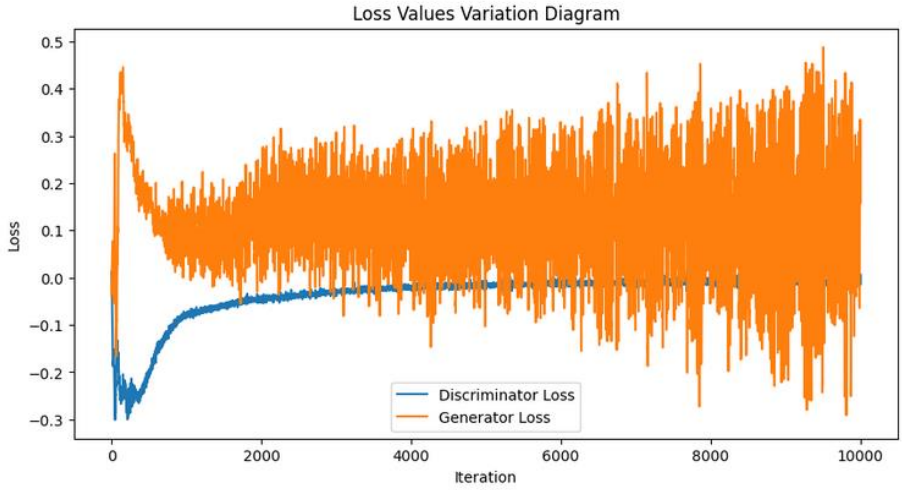
با توجه به الگوریتم بالا تغییرات زیر را در کد قسمت ابتدایی سوال انجام میدهیم.

برای اعمال تداوم Lipschitz از discriminator، WGAN ها یک تکنیک weight clipping را معرفی می کنند. به طور خاص، وزن های discriminator پس از هر مرحله تمرین به مقدار ثابت کوچکی بریده می شود. این برش کمک می کند تا اطمینان حاصل شود که هنجار گرادیان تمایز کننده محدود است. برای پیادا سازی این قسمت در تابع gan_step کد زیر را اضافه میکنیم:

```
# Clip discriminator weights
for p in discriminator.parameters():
p.data.clamp_(-0.01, 0.01)
```

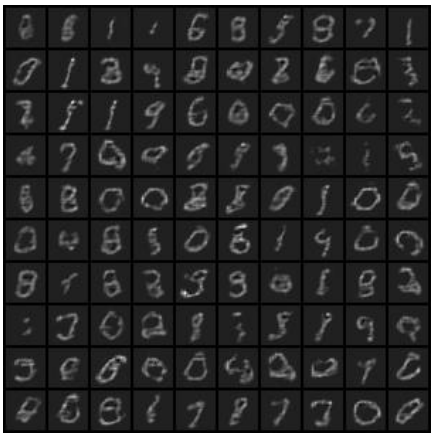
از RMSprop برای آموزش مدل استفاده میکنیم:

```
g_opt = torch.optim.RMSprop(model.g.parameters(), lr=0.00005)
d_opt = torch.optim.RMSprop(model.d.parameters(), lr=0.00005)
```



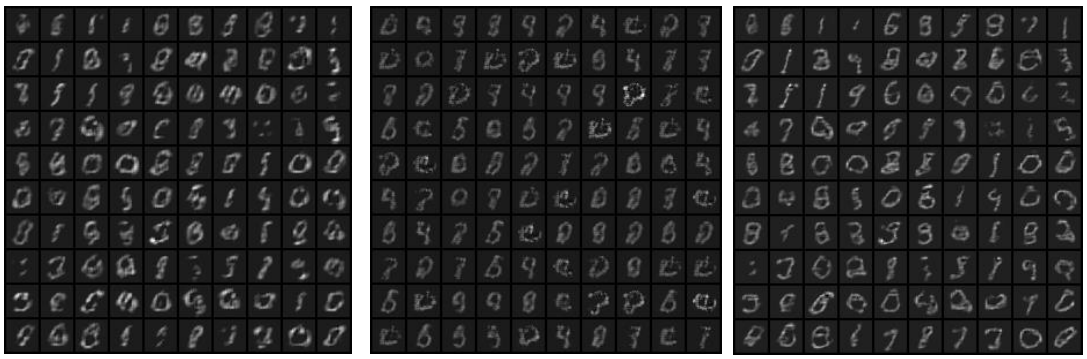
شکل ۱۰- نمودار تغییرات **loss**

خروجی نهایی مدل:

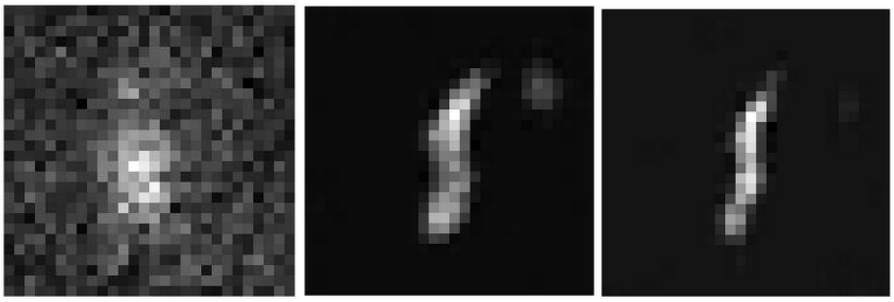


شکل ۱۱- نمونه خروجی تولیدی به وسیله شبکه **GAN**

خروجی مدل به ازای ۳ ایپاک:



شکل ۱۲ - خروجی مدل به ازای ۳ ایپاک ابتدایی میانی و نهایی (از سمت چپ به ترتیب)



شکل ۱۳ - خروجی مدل به ازای ۳ ایپاک ابتدایی میانی و نهایی

مشاهده می کنیم که مدل به خوبی داده ها را تولید کرده است ولی هنوز دقت خوبی ندارد، برای افزایش دقت مدل با ساختاری که در ابتدا تعریف شده است را برای ۵۰ ایپاک به شکل زیر آموزش میدهیم:

```
# Hyperparameters
batch_size = 64
z_dim = 100
lr = 0.00005
epochs = 50

# Data loading and preprocessing
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

train_dataset = datasets.MNIST(root='./data', train=True, download=True,
transform=transform)
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True,
num_workers=4, pin_memory=True)

# Initialize models and optimizers
generator = Generator(z_dim, num_channels)
discriminator = Discriminator(num_channels)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
generator.to(device)
discriminator.to(device)

optimizer_G = optim.RMSprop(generator.parameters(), lr=lr)
optimizer_D = optim.RMSprop(discriminator.parameters(), lr=lr)

# Lists to store the losses
losses_G = []
```

```
losses_D = []

# WGAN training loop
for epoch in range(epochs):
    for real_images, _ in tqdm(train_loader, desc=f"Epoch {epoch+1}/{epochs}"):

        # Training Discriminator
        real_images = real_images.to(device)
        optimizer_D.zero_grad()

        # Generate fake images
        z = torch.randn(batch_size, z_dim).to(device)
        fake_images = generator(z)

        # Discriminator predictions
        real_preds = discriminator(real_images)
        fake_preds = discriminator(fake_images.detach())

        # WGAN loss
        loss_D = -torch.mean(real_preds) + torch.mean(fake_preds)

        # Update Discriminator
        loss_D.backward()
        optimizer_D.step()

        # Clip discriminator weights
        for p in discriminator.parameters():
            p.data.clamp_(-0.01, 0.01)

        # Training Generator
        optimizer_G.zero_grad()

        # Generate fake images
        z = torch.randn(batch_size, z_dim).to(device)
        fake_images = generator(z)

        # Discriminator predictions on fake images
        fake_preds = discriminator(fake_images)

        # WGAN loss for the generator
        loss_G = -torch.mean(fake_preds)

        # Update Generator
        loss_G.backward()
        optimizer_G.step()

    # Save losses
    losses_G.append(loss_G.item())
```

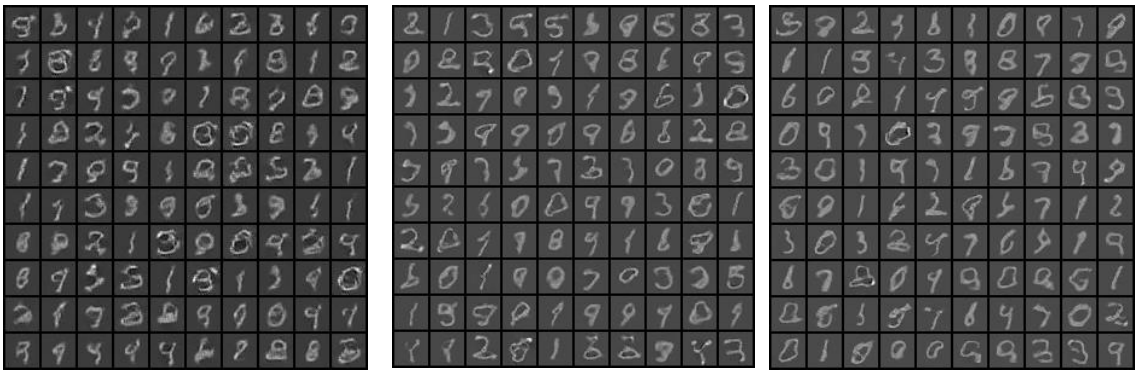
```
    losses_D.append(loss_D.item())

    # Print and save generated images (10x10 grid)
    if (epoch + 1) % 10 == 0:
        with torch.no_grad():
            fake_samples = generator(torch.randn(100, z_dim).to(device)).cpu()  # 
10x10 grid, 100 samples
        torchvision.utils.save_image(fake_samples.view(100, 1, 28, 28), 
f"gan_generated_epoch_{epoch + 1}.png", normalize=True, nrow=10)
```

در این حالت پارامترها باتوجه به الگوریتم توضیح داده شده پیاده سازی شده است.

در این حالت خروجی به شکل زیر است:

شکل ۱۴- نمودار تغییرات **loss**

خروجی مدل به ازای ۳ ایپاک:



شکل ۱۵خروجی مدل به ازای ۳ ایپاک ابتدایی میانی و نهایی (از سمت چپ به ترتیب)

مشاهده می کنیم که مدل بهتر از حالت قبل داده ها را تولید کرده است

Fid در این حالت **FID score: 46**.32213344221503 بدست می آید.

b.

ناپایداری تمرین: اگرچه هدف WGAN ها رسیدگی به مسئله mode collapse مشاهده شده در GAN های معمولی است، اما همچنان می توانند از training instability رنج ببرند. دستیابی به تعادل بین مولد و تمایز می تواند چالش برانگیز باشد و یافتن فراپارامترهای مناسب بسیار مهم است.

انتخاب معماری و هایپرپارامترها: عملکرد WGAN ها به انتخاب معماری شبکه عصبی و هایپرپارامترها حساس است. تنظیمات نامناسب می تواند منجر به همگرایی کند یا شکست آموزش شود. **(این مسئله را در روش اول دیدیم)**

مشکل در ارزیابی همگرایی: WGAN ها معیار مشخصی برای ارزیابی همگرایی در طول آموزش ارائه نمی دهند. برخلاف GANهای معمولی که از تلفات discriminator به عنوان معیار همگرایی استفاده می‌کنند، WGANها بر فاصله Wasserstein تکیه می‌کنند که تفسیر آن در طول آموزش چندان شهودی نیست.

هزینه محاسباتی: محاسبه فاصله Wasserstein شامل حل یک مسئله برنامه ریزی خطی است که می تواند از نظر محاسباتی گران باشد. این هزینه محاسباتی اضافی ممکن است WGAN ها را در برخی موارد کارآمدتر از GAN های معمولی کند.

مشکل در اجرای پیوستگی Lipschitz: WGAN ها نیاز دارند که discriminator ، Lipschitz پیوسته باشد، به این معنی که نرم گرادیان آن محدود است. در حالی که برش وزن معمولاً برای اعمال این محدودیت استفاده می شود، می تواند منجر به مشکلاتی مانند از بین رفتن گرادیان ها و مشکلات در یادگیری شود.

mode collapseدر سناریوهای خاص: در حالی که WGAN ها قصد کاهش mode collapse را دارند، هیچ تضمینی وجود ندارد که آنها این مشکل را به طور کامل از بین ببرند. در برخی موارد، WGAN ها همچنان ممکن است mode collapse را نشان دهند، جایی که مولد بر تولید مجموعه محدودی از نمونه ها تمرکز می کند و تنوع موجود در توزیع داده های واقعی را نادیده می گیرد.

برای حل مشکلات بالا میتوانیم راهکارهای زیر را امتحان کنیم:

- با نرخ های مختلف یادگیری برای مولد و متمایز کننده آزمایش کنیم.
- از بهینه سازهای تطبیقی مانند Adam استفاده کنیم.

- تکنیک هایی مانند پنالتی گرادیان WGAN-GP را به عنوان جایگزینی برای برش وزن برای اعمال تداوم لیپشیتز در نظر بگیریم.
- از تکنیک هایی مانند بهینه سازی هایپرپارامتر برای خودکار کردن فرآیند جستجو استفاده کنیم.
- افزایش ظرفیت مولد برای ایجاد تنوع در نمونه های تولید شده.
- با فرمول‌های جایگزین فاصله Wasserstein که از نظر محاسباتی کارآمدتر هستند، آزمایش کنیم.

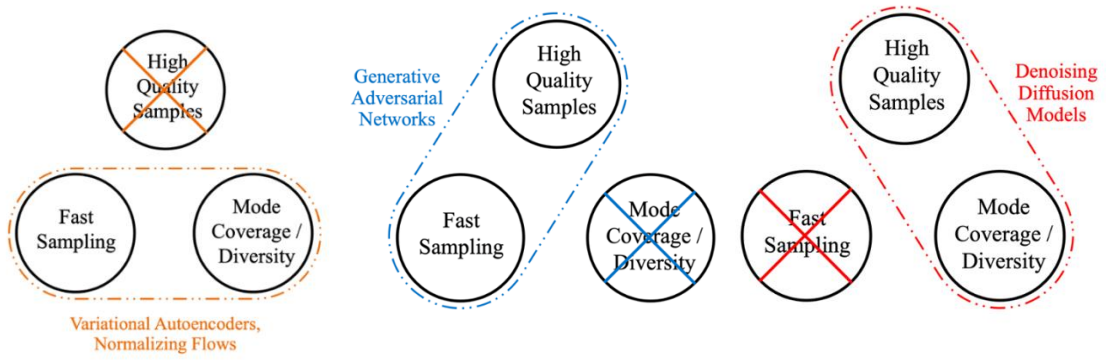
سوال ۲ – Diffusion Model

سوالات تئوری:

(سوال ۱)

Paradigm	Quality	Diversity	Speed
VAE	✗	✓	✓
GAN	✓	✗	✓
Diffusion	✓	✓	✗

شکل ۱۶ – جدول مقایسه مدل ها



شکل ۱۷ - **The Generative Learning Trilemma**

کیفیت:

- Normalizing Flow: Normalizing Flow آنها می توانند توزیع های پیچیده را مدل کنند و الگوهای پیچیده ای را در داده ها ثبت کنند. ولی تصاویر تولید شده همانند VAE کیفیت خوبی ندارند.
- GAN: GAN ها همچنین به دلیل تولید نمونه های با کیفیت بالا مشهور هستند. فرآیند آموزش adversarial به GAN ها کمک می کند تا تصاویر واقعی و واضح تولید کنند.
- VAE: VAE ها معمولاً نمونه های جذاب بصری کمتری نسبت به GAN ها و Normalizing Flow تولید می کنند. بازسازی‌ها ممکن است کمتر واضح باشند، و ممکن است مشکلاتی با mode collapse وجود داشته باشد.

- شبکه‌های Diffusion-based: این مدل ها، از جمله مدل‌هایی مانند DDPM می‌توانند نمونه‌هایی با کیفیت بالا، به‌ویژه در وظایف تولید تصویر تولید کنند. آنها از رویکرد متفاوتی در مقایسه با مدل‌های مولد دیگر استفاده می‌کنند.

تنوع:

- Normalizing Flow: Normalizing Flow تمایل به گرفتن حالت های مختلف در توزیع داده ها دارد. آنها قادر به تولید نمونه از بخش های مختلف توزیع هستند.
- GAN: GAN ها ممکن است از mode collapse رنج ببرند، جایی که مولد یک سمپلی تولید میکند که discriminator قادر به تشخیص واقعی یا فیک بودن آن نیست و نتواند تشخیص دهد، از آن پس مولد همان sample را تولید میکند.
- VAE : در حالی که VAE ها می توانند با چالش هایی مانند mode collapse مواجه شوند و ممکن است با کیفیت نمونه برخی از مدل های دیگر مطابقت نداشته باشند، ولی آنها به طور کلی از نظر تنوع نمونه برتر هستند. این باعث می‌شود زمانی که تنوع یک عامل حیاتی در کار مدل‌سازی مولد است، انتخاب خوبی باشند.
- شبکه‌های Diffusion-based: مدل‌های Diffusion-based، با طراحی، می‌توانند حالت‌های متنوعی را در توزیع داده‌ها ثبت کنند و در نتیجه نمونه‌های متنوع‌تری به دست آورند.

سرعت generation داده ها/ سرعت نمونه برداری:

- Normalizing Flow: استنتاج در Normalizing Flow می تواند از نظر محاسباتی گران باشد، به ویژه برای مدل های پیچیده و داده های با ابعاد بالا. سرعت نمونه برداری ممکن است در مقایسه با برخی روش های دیگر کندتر باشد.
- GAN: GAN ها می توانند نمونه ها را نسبتاً سریع در طول استنتاج پس از آموزش مدل تولید کنند. عبور رو به جلو از مولد معمولاً از نظر محاسباتی کارآمد است.
- VAE: VAE ها عموماً سریعتر از Normalizing Flow هستند، اما ممکن است در تولید نمونه ها به سرعت GAN ها نباشند.
- شبکه‌های Diffusion-based: نمونه‌برداری در مدل‌های Diffusion-based می‌تواند از نظر محاسباتی سخت باشد، و تولید نمونه‌های با کیفیت بالا ممکن است در مقایسه با GAN یا VAE به زمان بیشتری نیاز داشته باشد. چون باید مثلا ۱۰۰۰ بار denoise کند تا به یک sample برسیم.

(سوال ۲)

در مسیر forward داریم

$$
\begin{aligned}
&x_0 \\
&x_1 = \sqrt{\alpha_1}x_0 + \sqrt{1-\alpha_1}\varepsilon_1 \quad 0 \le \alpha_1 \le 1, \varepsilon_1 \sim N(0, I) \\
&x_2 = \sqrt{\alpha_2}x_1 + \sqrt{1-\alpha_2}\varepsilon_2 \\
&. \\
&. \\
&x_T
\end{aligned}
$$

برای اینکه با یک مرحله بازنمایی به هرکدام از بازنمایی های میانی برسیم روابط بالا را به شکل زیر مینویسیم:

$$
\begin{aligned}
&x_0 \\
&x_1 = \sqrt{\alpha_1}x_0 + \sqrt{1-\alpha_1}\varepsilon_1 \quad 0 \le \alpha_1 \le 1, \varepsilon_1 \sim N(0, I) \\
&x_2 = \sqrt{\alpha_2}x_1 + \sqrt{1-\alpha_2}\varepsilon_2 = \sqrt{\alpha_1\alpha_2}x_0 + \sqrt{\alpha_2(1-\alpha_1)}\varepsilon_1 + \sqrt{1-\alpha_2}\varepsilon_2
\end{aligned}
$$

ε_2 و ε_1 دارای توزیع نرمال هستند و از هم مستقل هستند، با توجه به راهنمایی صورت گرفته در صورت سوال آنها را به شکل زیر مینویسیم:

$$
\begin{aligned}
&\text{when we merge two gaussian with different var iance}: \sqrt{1-\alpha_2+\alpha_2(1-\alpha_1)} = \sqrt{1-\alpha_1\alpha_2} \\
&\rightarrow \tilde{\varepsilon} \sim N(0, 1-\alpha_1\alpha_2) \\
&\sqrt{1-\alpha_1\alpha_2}\tilde{\varepsilon} \rightarrow \tilde{\varepsilon} \sim N(0, I)
\end{aligned}
$$

مطابق بالا بقیه x_t ها را نیز محاسبه میکنیم:

$$
\begin{aligned}
&x_2 = \sqrt{\alpha_1\alpha_2}x_0 + \sqrt{1-\alpha_1\alpha_2}\varepsilon \\
&x_3 = \sqrt{\alpha_1\alpha_2\alpha_3}x_0 + \sqrt{1-\alpha_1\alpha_2\alpha_3}\varepsilon \\
&\text{in paper we have}: \bar{\alpha}_t = \alpha_1\alpha_2...\alpha_t \\
&\Rightarrow x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\varepsilon \\
&\text{In DDPM we want to predict } \varepsilon \\
&q(x_t \mid x_0) = N(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t)I)
\end{aligned}
$$

در حالت کلی داریم: $x_t = \sqrt{\bar{\alpha}_{t-1}}x_{t-1} + \sqrt{1-\bar{\alpha}_{t-1}}\varepsilon$

(سوال ۳)

درفرآیند معکوس که از $q(x_{t-1}|x_t)$ نمونه برداری می کنیم، می توانیم نمونه واقعی را از ورودی نویز گاوسی، $x_T \sim N(0,I)$ دوباره ایجاد کنیم. **اگر β_t به اندازه کافی کوچک باشد، $q(x_{t-1}|x_t)$ نیز گوسی خواهد بود.** اما ما نمی‌توانیم به راحتی $q(x_{t-1}|x_t)$ را تخمین بزنیم زیرا نیاز به استفاده از کل مجموعه داده دارد و فرم بسته‌ای نخواهد داشت،

$$q(x_{t-1} \mid x_t) = \frac{q(x_t \mid x_{t-1})q(x_{t-1})}{q(x_t)}$$

$$p_\theta(x_{0:T}) = p_\theta(x_T) \prod_{t=1}^{T} p_\theta(x_{t-1}|x_t)$$

$$p_\theta(x_{t-1}|x_t) = \mathcal{N}\big(x_{t-1}; \mu_\theta(x_t, t), \Sigma_\theta(x_t, t)\big)$$

بنابراین ما باید یک مدل p_θ (روابط مشخص شده در بالا) را برای تقریب این احتمالات شرطی به منظور اجرای فرآیند reverse diffusion یاد بگیریم و احتمال شرطی معکوس روی x0 شرطی می شود و خواهیم داشت:

$$q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) = \mathcal{N}(\mathbf{x}_{t-1}; \tilde{\boldsymbol{\mu}}_t(\mathbf{x}_t, \mathbf{x}_0), \tilde{\beta}_t \mathbf{I}),$$

$$\text{where} \quad \tilde{\boldsymbol{\mu}}_t(\mathbf{x}_t, \mathbf{x}_0) := \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}\mathbf{x}_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\mathbf{x}_t \quad \text{and} \quad \tilde{\beta}_t := \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$$

تابع هزینه به شکل زیر می شود:

$$\mathbb{E}_q \Big[\underbrace{D_{\mathrm{KL}}(q(\mathbf{x}_T|\mathbf{x}_0) \parallel p(\mathbf{x}_T))}_{L_T} + \sum_{t>1} \underbrace{D_{\mathrm{KL}}(q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) \parallel p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t))}_{L_{t-1}} \underbrace{- \log p_\theta(\mathbf{x}_0|\mathbf{x}_1)}_{L_0} \Big]$$

در نتیجه، تمام واگرایی های KL در معادله بالا مقایسه‌ای بین گوسی‌ها هستند، بنابراین می‌توان آن‌ها را به روش Rao-Blackwellized با عبارات شکل بسته به جای تخمین‌های مونت کارلو با واریانس بالا محاسبه کرد.

در کل انتخاب توزیع گاوسی به چند دلیل انجام می شود:

۱. سادگی تحلیل: توزیع های گوسی از نظر تحلیلی راحت و رفتار خوبی دارند. شرطی‌سازی در یک توزیع گوسی از نظر محاسباتی کارآمد است و اغلب به راه‌حل‌های بسته منجر می‌شود و فرآیند مدل‌سازی و آموزش کلی را ساده می‌کند.

۲. بازده محاسباتی: نمونه برداری از یک توزیع گوسی از نظر محاسباتی کارآمد است و روش های به خوبی تثبیت شده برای نمونه برداری از توزیع گوسی وجود دارد.

۳. Expressiveness: علیرغم سادگی یک توزیع گوسی، می تواند به اندازه کافی گویا باشد که طیف وسیعی از توزیع های داده را به تصویر بکشد. انعطاف‌پذیری توزیع‌های گوسی در نمایش اشکال و ساختارهای مختلف، آنها را به گزینه‌ای مناسب در بسیاری از سناریوهای مدل‌سازی احتمالی تبدیل می‌کند.

۴. استقلال آماری: در زمینه انتشار denoising diffusion، فرض توزیع های گوسی کمک می کند تا اطمینان حاصل شود که فرآیند نویز زدایی استقلال آماری را بین مراحل مختلف زمانی حفظ می کند. این برای ثبات کلی و اثربخشی مدل مهم است.

(سوال ۴)

$$L_{VLB} = L_T + \sum_{t=1}^{T-1} L_t + L_0 \quad where \quad \begin{cases} L_T = D_{KL}\big(q(x_T|x_0) \parallel p_\theta(x_T)\big) \\ L_t = D_{KL}\big(q(x_{t-1}|x_t, x_0) \parallel p_\theta(x_{t-1}|x_t)\big) \quad 1 \le t < T \\ L_0 = -\log p_\theta(x_0|x_1) \end{cases}$$

Forward process and LT:

شامل احتمال ورود داده ها تحت تاثیر prior است. این ترم نشان میدهد که مدل چقدر می‌تواند داده‌هایی تولید کند که متناسب با توزیع قبلی مشخص شده باشد.

Reverse process and L1: T −1 :

نشاندهنده ی واگرایی Kullback-Leibler KL است و تفاوت بین توزیع واقعی و توزیع تخمینی مدل را اندازه می‌گیرد. ما در فرایند یادگیری این مقدار نشان میدهد که توزیع‌های تخمینی مدل در هر مرحله با توزیع‌های واقعی همسو هستند.

Data scaling, reverse process decoder, and L0 :

این عبارت شامل احتمال گزارش داده اصلی x_0 با توجه به اولین مشاهده نویزدار x_1 است.که توانایی مدل را برای بازسازی داده های اصلی از ورودی نویزدار اولیه ارزیابی می کند.

(سوال ۵)

$$L_{VLB} = L_T + \sum_{t=1}^{T-1} L_t + L_0 \quad where \quad \begin{cases} L_T = D_{KL}\big(q(x_T|x_0) \parallel p_\theta(x_T)\big) \\ L_t = D_{KL}\big(q(x_{t-1}|x_t, x_0) \parallel p_\theta(x_{t-1}|x_t)\big) \quad 1 \le t < T \\ L_0 = -\log p_\theta(x_0|x_1) \end{cases}$$

شکل ۱۸ – تابع هزینه

ترم L_t در مقاله درنظر گرفته نشده.

LT -واگرایی KL بین توزیع متغیر نهفته نهایی در فرآیند رو به جلو و اولین توزیع متغیر پنهان در فرآیند معکوس. این عبارت در تابع ضرر گنجانده نشده است زیرا هیچ پارامتر شبکه عصبی وجود ندارد که بتوان آن را کنترل کرد. در عوض، نویسندگان برای اطمینان از اینکه توزیع‌های متغیر پنهان به خوبی تطبیق دارند، بر یک زمان‌بندی واریانس تکیه می‌کنند.L_T ظاهر نمی شود زیرا واریانس های فرآیند رو به جلو β_t ثابت هستند. درنتیجه L_T در طول تمرین ثابت است و می توان آن را نادیده گرفت.

دلیل اصلی حذف شدن ، کارایی محاسباتی است. محاسبه این ترم، می تواند برای مدل های پیچیده و مجموعه داده های بزرگ بسیار پرهزینه باشد..

L0 - عبارت likelihood، که احتمال تصویر مشاهده شده را تحت توزیع مدل اندازه گیری می کند.

به طور معمول برای یادگیری تابع حذف نویز مهم است و نادیده گرفته نمی شود. این اندازه گیری می کند که چگونه مدل می تواند داده های اصلی را از داده های نویزدار بازسازی کند. این عبارت برای آموزش مدل برای تولید نمونه ضروری است و در تابع هدف نگهداری می شود. در مقاله نشان داده شده است که آموزش با L0 منجر به آموزش ناپایدار می شود و کیفیت تصاویر تولید شده را بهبود نمی بخشد و در تابع نهایی آن را حذف کرده است.

(سوال ۶)

تأثیر استفاده از توزیع پیچیده برای مسیر رو به عقب به جای توزیع گاوسی در مقاله DDPM، معرفی غیرخطی بودن در محاسبه تابع هزینه خواهد بود. این غیر خطی بودن به طور بالقوه می‌تواند عملکرد مدل را تحت تاثیر قرار بدهد. با این حال، می تواند آموزش مدل را دشوارتر و پایدارتر کند.

هنگامی که از یک توزیع پیچیده برای مسیر عقب استفاده می کنیم، پارامترهای اضافی را به مدل وارد می کنیم. از این پارامترها می توان برای ثبت روابط پیچیده تری بین تصویر ورودی و latent استفاده کرد. با این حال، این همچنین پیچیدگی بیشتری را در محاسبه واگرایی KL وارد می کند. این پیچیدگی می

تواند یادگیری پارامترهای بهینه را برای مدل دشوارتر کند و همچنین می تواند مدل را در طول آموزش ناپایدارتر کند.

محاسبه مسیر معکوس شامل نمونه برداری از توزیع شرطی $q(x_{t-1} \mid x_t)$ اگر این توزیع پیچیده باشد، نمونه‌برداری ممکن است از نظر محاسباتی سخت‌تر باشد. در مقابل، سادگی توزیع گاوسی، نمونه برداری را ساده می کند و فرآیند بهینه سازی ممکن است کندتر همگرا شود. همچنین نیاز به حافظه بیشتری داریم.

(سوال ۷)

عبارت L_t برای به حداقل رساندن اختلاف از $\tilde{\mu}$ پارامتر می شود:

$$
\begin{aligned}
L_t &= \mathbb{E}_{\mathbf{x}_0,\epsilon}\left[\frac{1}{2\|\boldsymbol{\Sigma}_\theta(\mathbf{x}_t,t)\|_2^2}\|\tilde{\boldsymbol{\mu}}_t(\mathbf{x}_t,\mathbf{x}_0)-\boldsymbol{\mu}_\theta(\mathbf{x}_t,t)\|^2\right] \\
&= \mathbb{E}_{\mathbf{x}_0,\epsilon}\left[\frac{1}{2\|\boldsymbol{\Sigma}_\theta\|_2^2}\left\|\frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t-\frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_t\right)-\frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t-\frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t,t)\right)\right\|^2\right] \\
&= \mathbb{E}_{\mathbf{x}_0,\epsilon}\left[\frac{(1-\alpha_t)^2}{2\alpha_t(1-\bar{\alpha}_t)\|\boldsymbol{\Sigma}_\theta\|_2^2}\|\boldsymbol{\epsilon}_t-\boldsymbol{\epsilon}_\theta(\mathbf{x}_t,t)\|^2\right] \\
&= \mathbb{E}_{\mathbf{x}_0,\epsilon}\left[\frac{(1-\alpha_t)^2}{2\alpha_t(1-\bar{\alpha}_t)\|\boldsymbol{\Sigma}_\theta\|_2^2}\|\boldsymbol{\epsilon}_t-\boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0+\sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}_t,t)\|^2\right]
\end{aligned}
$$

تابع تلفات اولیه که با L_t مشخص می شود، شامل امید ریاضی با توجه به شرایط اولیه x_0 و شرایط نویز epsilon می شود. Sigma_theta و mu_theta به ترتیب نشان دهنده ماتریس کوواریانس diffusion و میانگین در یک زمان معین t هستند. با در نظر گرفتن ساختار کوواریانس diffusion، loss به عنوان معیاری برای اختلاف بین مقادیر پیش‌بینی‌شده و واقعی فرمول‌بندی می‌شود. عبارت وزن دهی شامل alpha_t بخشی از طراحی مدل برای تنظیم سهم عبارات مختلف در ضرر کلی است. با این حال، در مقاله پیشنهاد می کند که یک نسخه ساده شده از تابع هزینه منجر به نتایج آموزشی بهتری می شود. این تابع تلفات ساده، عبارت وزن‌دهی را نادیده می‌گیرد و صرفاً بر اختلاف بین شرایط نویز پیش‌بینی‌شده و واقعی تمرکز می‌کند. expectation طی مراحل زمانی t و شرایط اولیه x_0 انجام می شود. سادگی این تابع، به دلیل حذف پیچیدگی‌های معینی که توسط عبارت وزن‌دهی اصلی معرفی شده است، فرآیند تمرین را مؤثرتر می‌کند.

$$
\begin{aligned}
L_t^{\text{simple}} &= \mathbb{E}_{t\sim[1.T],\mathbf{x}_0,\epsilon_t}\left[\|\epsilon_t-\epsilon_\theta(\mathbf{x}_t,t)\|^2\right] \\
&= \mathbb{E}_{t\sim[1.T],\mathbf{x}_0,\epsilon_t}\left[\left\|\epsilon_t-\boldsymbol{\epsilon}_\theta\left(\sqrt{\bar{\alpha}_t}\mathbf{x}_0+\sqrt{1-\bar{\alpha}_t}\epsilon_t,t\right)\right\|^2\right]
\end{aligned}
\quad . L_{simple} = L_t^{simple} + C
$$

معادله بالا شبیه تطبیق امتیاز حذف نویز در مقیاس های نویز چندگانه است که با t نمایه شده است. این معادله برابر است با (یک جمله) کران تغییرات برای فرآیند معکوس Langevin-like، همچنین در مقاله اشاره شده است که بهینه سازی یک هدف شبیه به تطبیق امتیاز حذف نویز معادل استفاده از variational inference برای برازش حاشیه زمان محدود یک زنجیره نمونه‌برداری شبیه دینامیک Langevin.

در مقاله نشان داده شده است که از آنجایی که واریانس را ثابت نگه می دارند، فقط باید میانگین توزیع را پیش بینی کنند و فقط نویز ε را پیش‌بینی می کنند که از توزیع نرمال نمونه‌برداری شده و از طریق ترفند reparameterization به تصویر اضافه شده است. همچنین نشان دادند که پیش‌بینی نویز پایدارتر است. از آنجایی که فقط باید نویز اضافه شده را پیش بینی کنند، می توان از تلفات MSE بین نویز پیش بینی شده و نویز واقعی اضافه شده به تصویر استفاده کرد.

(سوال ۸)

در این شبکه ورودی تصویر در زمان t و خروجی نویز درون آن تصویر است. علاوه بر این، در هر لایه در شبکه، اطلاعات زمانی را اضافه می کنند تا به مدل کمک کنند تا بداند در کجای فرآیند diffusion قرار دارد. $p(x_{t-1} \mid x_t, t)$

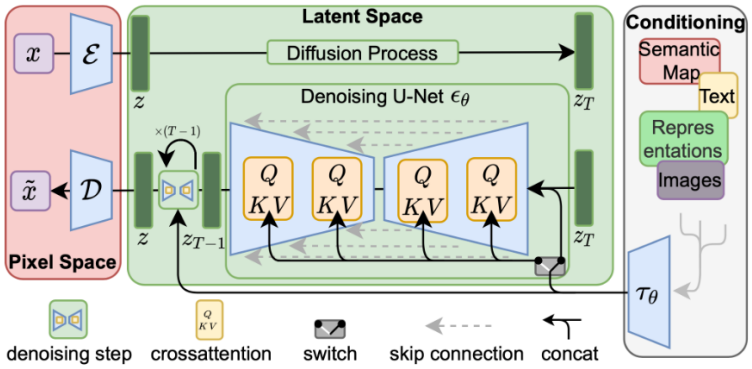
پارامتر زمان گسسته است برای اینکه بتوانیم از آن در این مدل استفاده کنیم، از positional encodings استفاده میکنیم. به جای رمزگذاری مکان در دنباله، می‌توان embeddingها را به‌عنوان بردارهای timestep در نظر گرفت که در آن یک بردار، یک مرحله زمانی را نشان می‌دهد. همچنین می‌توان بردار زمان را به تعداد کانال‌ها منتقل کرد تا دو بردار ایجاد شود، یکی برای جابجایی رمزگذاری‌های تصویر میانی و دیگری برای مقیاس‌بندی رمزگذاری‌های تصویر میانی.

(سوال ۹)

Latent Diffusion (LDM) این موضوع را با معرفی یک نمایش نهفته (به جای فضای پیکسلی) ، که تعبیه‌ای با ابعاد پایین‌تر از تصویر است، برطرف می‌کند. این نمایش نهفته برای هدایت فرآیند Diffusion استفاده می شود و امکان تولید کارآمدتر تصاویر با وضوح بالا را فراهم می کند. اکثر بیت‌های تصویر به جزئیات ادراکی کمک می‌کنند و ترکیب معنایی و مفهومی همچنان پس از فشرده‌سازی باقی می‌ماند. مدل LD همچنین دارای لایه‌های cross-attention است که آن را قادر می‌سازد اطلاعات شرطی اضافی، مانند

متن یا جعبه‌های محدودکننده را در خود جای دهد. این کار هزینه آموزش را کاهش می دهد و سرعت استنتاج را سریعتر می کند. LDM فشرده‌سازی ادراکی و فشرده‌سازی معنایی را با یادگیری مدل‌سازی مولد، ابتدا با حذف افزونگی در سطح پیکسل با autoencoder و سپس تولید با فرآیند diffision در latent، تجزیه می‌کند.

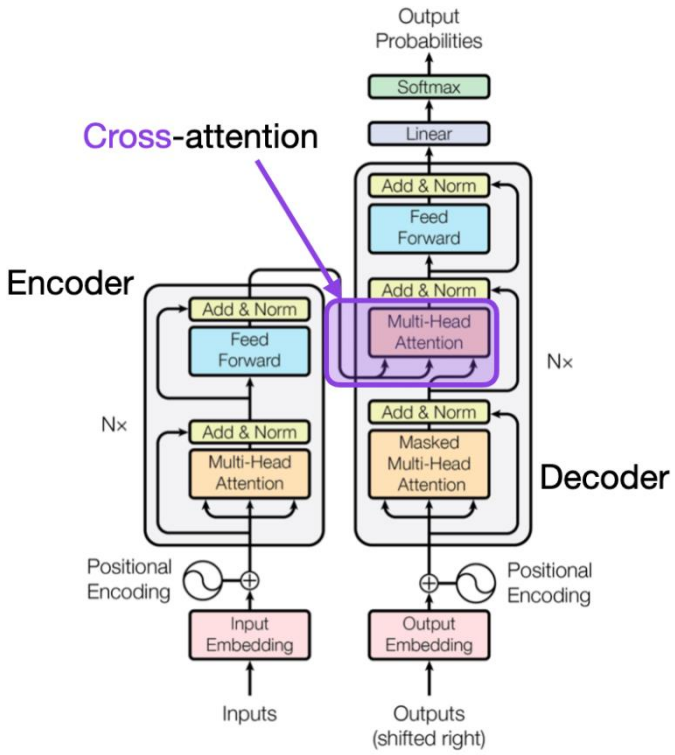
$$\text{Attention}(\mathbf{Q},\mathbf{K},\mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right)\cdot\mathbf{V}$$
$$\text{where } \mathbf{Q} = \mathbf{W}_Q^{(i)}\cdot\varphi_i(\mathbf{z}_i),\ \mathbf{K} = \mathbf{W}_K^{(i)}\cdot\tau_\theta(y),\ \mathbf{V} = \mathbf{W}_V^{(i)}\cdot\tau_\theta(y)$$
$$\text{and } \mathbf{W}_Q^{(i)}\in\mathbb{R}^{d\times d_\epsilon^i},\ \mathbf{W}_K^{(i)},\mathbf{W}_V^{(i)}\in\mathbb{R}^{d\times d_\tau},\ \varphi_i(\mathbf{z}_i)\in\mathbb{R}^{N\times d_\epsilon^i},\ \tau_\theta(y)\in\mathbb{R}^{M\times d_\tau}$$



شکل ۱۹- معماری LDM

فرآیندهای diffusion و حذف نویز در بردار پنهان z اتفاق می افتد. مدل حذف نویز یک U-Net شرطی شده با زمان است که با cross-attention برای کنترل اطلاعات شرطی شده برای تولید تصویر تقویت شده است. طراحی معادل نمایش fuse representationهای مختلف در مدل با cross-attention است. هر نوع اطلاعات شرطی‌سازی با یک رمزگذار مخصوص جفت می‌شود تا ورودی شرطی‌سازی y را به یک نمایش که می‌تواند در cross-attention نگاشت شود، ارائه می‌کند.

Cross-attention، همچنین به عنوان multi-head attention شناخته می شود، مکانیزمی است که در شبکه های عصبی، به ویژه در معماری ترانسفورماتور، برای ثبت روابط بین عناصر مختلف در یک دنباله استفاده می شود. در زمینه پردازش زبان طبیعی، مانند ترجمه ماشینی یا خلاصه‌سازی متن، Cross-attention به مدل اجازه می‌دهد تا هنگام تولید هر عنصر از دنباله خروجی، بر بخش‌های مختلف دنباله ورودی تمرکز کند. این شامل محاسبه وزن توجه برای هر عنصر در دنباله خروجی با توجه به همه عناصر در دنباله ورودی است، و مدل را قادر می‌سازد تا اهمیت نشانه‌های ورودی مختلف را برای هر نشانه خروجی متفاوت ارزیابی کند. این کار توانایی مدل را برای گرفتن وابستگی‌ها و روابط دوربرد در کل توالی ورودی تسهیل می‌کند و عملکرد آن را در درک و تولید توالی‌های منسجم از اطلاعات بهبود می‌بخشد.



Source: “Attention Is All You Need” (https://arxiv.org/abs/1706.03762)

شکل ۲۰ - **Cross-attention**

(سوال ۱۰)

DDIM روشی را برای سرعت بخشیدن به تولید تصویر با کاهش کیفیت تصویر معرفی کرد. این کار را با تعریف مجدد فرآیند diffusion به عنوان یک فرآیند non-Markovian انجام می دهد و در مسیر برگشت به جای اینکه دونه دونه عقب برویم، چند خط درمیان عقب میاییم که باعث سریعتر شدن می شود.DDIM یک توزیع احتمالی را پیشنهاد می دهد که وقتی loss/elbo را مینویسیم، بدون هیچ تغییری به پیزی که در DDPM بود، میرسیم. دراینجا مسیر رفت فرق دارد ولی loss همانی است که در DDPM داشتیم.

DDIM قادر است:

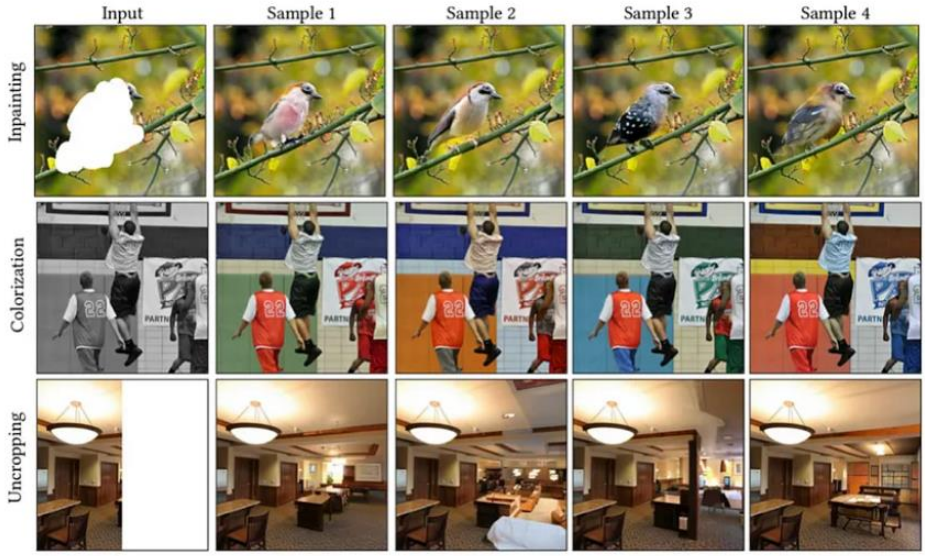
- با استفاده از تعداد بسیار کمتری از مراحل، نمونه های با کیفیت بالاتر تولید کنید.
- دارای ویژگی consistency هستند زیرا فرآیند تولیدی قطعی است، به این معنی که چندین نمونه مشروط بر یک متغیر پنهان باید ویژگی های سطح بالا مشابهی داشته باشند.
- به دلیل سازگاری، DDIM می تواند درون یابی معنایی معنادار را در متغیر latent انجام دهد.

$$\sigma_{\tau_i}(\eta) = \eta\sqrt{\frac{1-\alpha_{\tau_{i-1}}}{1-\alpha_{\tau_i}}}\sqrt{1-\frac{\alpha_{\tau_i}}{\alpha_{\tau_{i-1}}}}$$

شکل ۲۱ - **DDIM variance**

مدل یک DDIM است وقتیکه η=0 چون نویز وجود ندارد و یک DDPM اصلی وقتی η=1 است. هر η بین ۰ و ۱ درون یابی ای بین DDIM و DDPM است.

(سوال ۱۱)



شکل ۲۲ – نمونه ای از خروجی مدل ها - **Palette**

مدل‌های مبتنی Diffusion به‌عنوان یک رویکرد امیدوارکننده برای semantic segmentation پدیدار شده‌اند. آنها چندین مزیت برای این کار ارائه می دهند، از جمله:

تقویت داده ها: این مدل ها را می توان برای تولید داده های مصنوعی برای آموزش شبکه ها استفاده کرد و نیاز به برچسب گذاری دستی را کاهش داد. این داده‌های مصنوعی می‌توانند تغییرات متنوعی را در ظاهر و نور شیء ثبت کنند و توانایی تعمیم مدل segmentation را افزایش دهند.

Few-Shot Learning: مدل‌ها را می‌توان در وظایف few-shot semantic segmentation، که در آن تنها چند نمونه برچسب‌گذاری شده برای هر کلاس شی در دسترس است، استفاده کرد. با تولید تصاویر مصنوعی از این نمونه‌های برچسب‌گذاری شده، مدل‌های diffusion می‌توانند داده‌های آموزشی را گسترش دهند و عملکرد segmentation را برای کلاس‌های جدید بهبود بخشند.

Label Fusion: مدل‌های Diffusion را می‌توان برای ترکیب ماسک‌های segmentationچندگانه از منابع مختلف، مانند چسب‌گذاری انسانی یا سایر ماسک‌های تولید شده توسط ماشین استفاده کرد. این ادغام می تواند دقت و استحکام بخش بندی را بهبود بخشد.

تخمین عدم قطعیت: این مدل‌ها می‌توانند تخمین‌های عدم قطعیت را برای ماسک‌های segmentation تولید شده خود ارائه دهند. این اطلاعات عدم قطعیت می‌تواند برای اصلاح نتایج segmentationیا شناسایی مناطقی که مدل در آن‌ها اعتماد کمتری دارد، استفاده شود.

سوالات پیاده‌سازی:

(سوال ۱۲)

$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}) \quad q(\mathbf{x}_{1:T}|\mathbf{x}_0) = \prod_{t=1}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1})$$

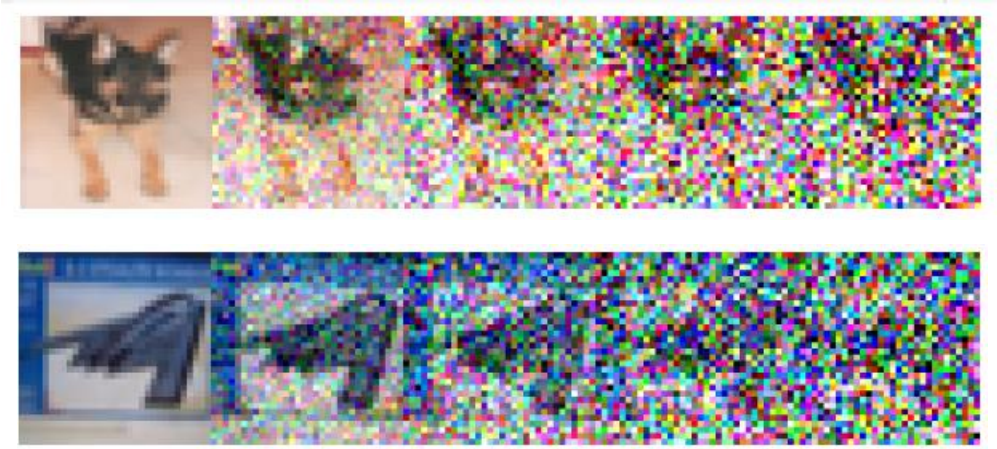
در این قسمت معادله بالا را به شکل زیر پیاده سازی میکنیم:

```
def q_xt_xtminus1(xtm1, t):
    # Parameters
    sqrt_one_minus_beta_t = (1. - beta[t]) ** 0.5
    sqrt_beta_t = beta[t] ** 0.5
    # Compute mean and variance
    mean = sqrt_one_minus_beta_t * xtm1  # sqrt(1-beta*t)*xtm1
    var = sqrt_beta_t  # 1-beta*t I
    # Generate noise shaped like xtm1
    eps = torch.randn_like(xtm1)
    # Combine mean and noise to get the final result
    result = mean + (var * eps)

    return result
```

این کد یک فرآیند تصادفی را با استفاده از یک نوع از دینامیک گرادیان تصادفی Langevin مدل می کند. این تابع دو پارامتر ورودی را می گیرد، xtm1 نماینده وضعیت قبلی فرآیند و t نماینده مرحله زمانی فعلی. میانگین و واریانس حالت بعدی را بر اساس پارامترهای داده شده بتا محاسبه می‌کند و نویز تصادفی eps را به شکل حالت قبلی با استفاده از تابع randn_like تولید می‌کند. نتیجه نهایی با ترکیب نویز متوسط و مقیاس شده به دست می آید.

خروجی در این حالت:

مشاهده میکنیم که در هر مرحله مقداری نویز به عکس اضافه شده است.



شکل ۲۳ - **Adding Noise**

(سوال ۱۳)

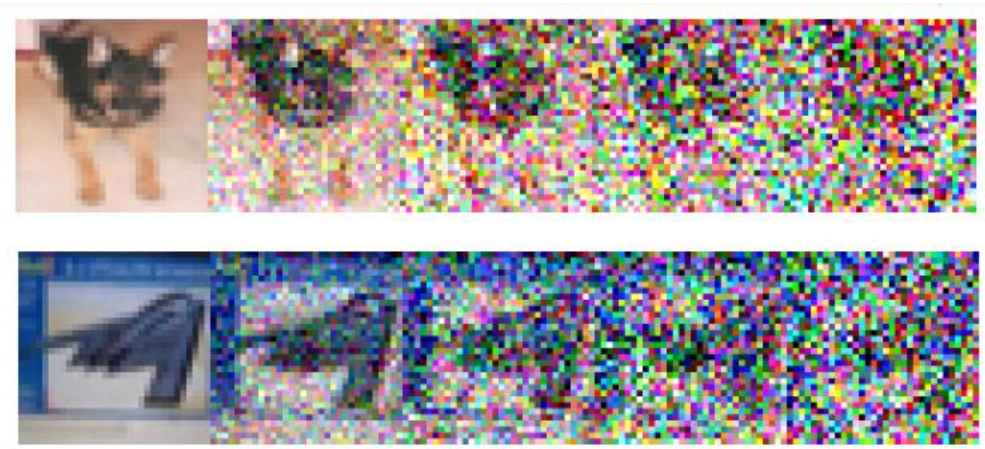
$$q(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I}) \text{ where } \bar{\alpha}_t = \prod_{i=1}^{T} \alpha_i$$

در این قسمت معادله بالا را به شکل زیر پیاده سازی میکنیم:

```
def q_xt_x0(x0, t):

    # Parameters
    sqrt_alpha_bar_t = alpha_bar[t] ** 0.5
    one_minus_alpha_bar_t = 1. - alpha_bar[t]
    # Compute mean and variance
    mean = sqrt_alpha_bar_t * x0  # now alpha_bar
    var = one_minus_alpha_bar_t  # (1-alpha_bar)
    # Generate noise shaped like x0
    eps = torch.randn_like(x0)
    # Combine mean and noise to get the final result
    result = mean + (var ** 0.5) * eps

    return result
```

همانطور که در سوال ۲ توضیح دادیم در مسیر رو به جلو نیازی به اضافه کردن نویز به صورت تکرار شونده نیست، در اینجا به ازای گام های ۰, ۲۰, ۴۰, ۶۰, ۸۰ خروجی را مشاهده می کنیم.

خروجی:



شکل ۲۴ - **model at different time steps**

(سوال ۱۴)

```
n_steps = 1000
```

تنها تفاوت آن با قسمت قبل این است که تابع خود نویز را هم بر میگرداند.

```
def q_xt_x0(x0, t):

    # Parameters
    sqrt_alpha_bar_t = alpha_bar[t] ** 0.5
    one_minus_alpha_bar_t = 1. - alpha_bar[t]
    # Compute mean and variance
    mean = sqrt_alpha_bar_t * x0  # now alpha_bar
    var = one_minus_alpha_bar_t  # (1-alpha_bar)
    # Generate noise shaped like x0
    eps = torch.randn_like(x0).to(x0.device)
    # Combine mean and noise to get the final result
    result = mean + (var ** 0.5) * eps

    return result , eps
```

ابرپارامترهایی که برای train استفاده شده است:

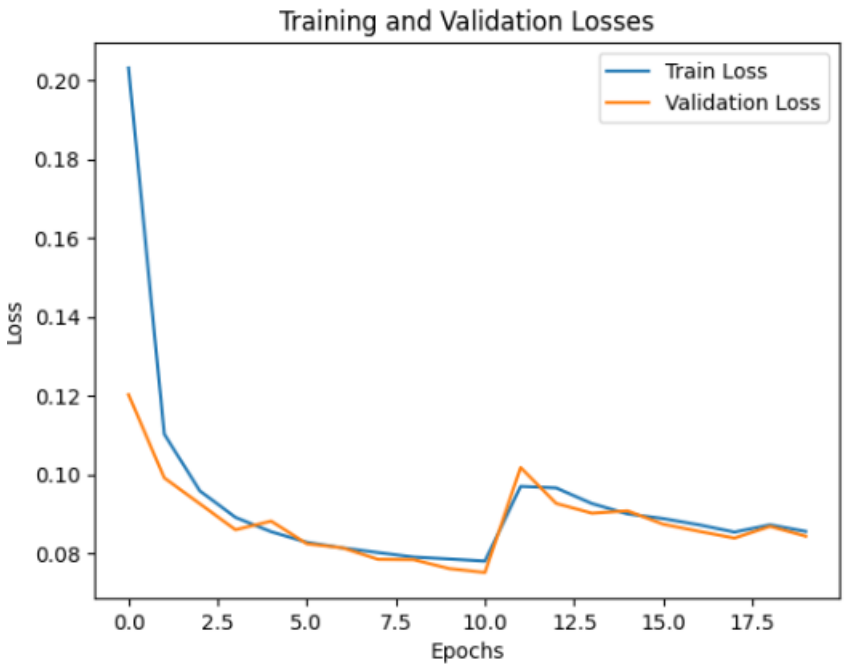
```
batch_size = 128
lr = 2e-4
num_epochs = 20
optimizer = torch.optim.AdamW(unet.parameters(), lr=lr)
```

train loop به شکل زیر است:

مدل بر روی مجموعه داده cifar10 با استفاده از یک تابع از loss جدید شامل نویز آموزش داده شده است. در طول هر تکرار ، شاخص‌های تصادفی برای تعیین یک گام زمانی گسسته t برای هر دسته از تصاویر ورودی ims انتخاب می‌شوند. سپس یک تابع q_xt_x0 برای تولید یک نمایش xt latent از تصاویر ورودی، همراه با نویز مرتبط استفاده می شود. U-Net این نمایش را برای پیش بینی سیگنال نویز pred_noise پردازش می کند. loss به عنوان میانگین مجذور خطا بین نویز پیش‌بینی‌شده و نویز واقعی محاسبه می‌شود و backpropagation برای بهینه‌سازی پارامترهای U-Net استفاده می‌شود. میانگین تلفات آموزش و اعتبارسنجی در طول batch ها ثبت می شود.

```
# Loss function container
train_losses = []
val_losses = []

# Training loop
for epoch in range(num_epochs):
    # Training phase
    unet.train()
    epoch_train_losses = []
    for i in tqdm(range(0, len(cifar10['train']) - batch_size, batch_size)):
        ims = [cifar10['train'][idx]['img'] for idx in range(i, i + batch_size)]
        tims = [img_to_tensor(im).cuda() for im in ims]
        x0 = torch.cat(tims)
        t = torch.randint(0, n_steps, (batch_size,), dtype=torch.long).cuda()
        xt, noise = q_xt_x0(x0, t)
        pred_noise = unet(xt.float(), t)
        loss = F.mse_loss(noise.float(), pred_noise)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        epoch_train_losses.append(loss.item())

    # Validation phase
    unet.eval()
    epoch_val_losses = []
    with torch.no_grad():
        for i in tqdm(range(0, len(cifar10['test']) - batch_size, batch_size)):
            ims_val = [cifar10['test'][idx]['img'] for idx in range(i, i +
batch_size)]
            tims_val = [img_to_tensor(im).cuda() for im in ims_val]
            x0_val = torch.cat(tims_val)
            t_val = torch.randint(0, n_steps, (batch_size,),
dtype=torch.long).cuda()
```

```
            xt_val, noise_val = q_xt_x0(x0_val, t_val)
            pred_noise_val = unet(xt_val.float(), t_val)
            loss_val = F.mse_loss(noise_val.float(), pred_noise_val)
            epoch_val_losses.append(loss_val.item())


    # Calculate average losses for the epoch
    avg_train_loss = sum(epoch_train_losses) / len(epoch_train_losses)
    avg_val_loss = sum(epoch_val_losses) / len(epoch_val_losses)

    # Append losses to the lists
    train_losses.append(avg_train_loss)
    val_losses.append(avg_val_loss)

    # Print or log the losses for monitoring
    print(f'Epoch {epoch + 1}/{num_epochs}: Train Loss: {avg_train_loss:.4f}, Val
Loss: {avg_val_loss:.4f}')
```



شکل ۲۵ – نمودار تابع خطا **loss,validation**

(سوال ۱۵)

```
def p_xt(xt, noise, t):

    # Extract alpha_t, alpha_bar_t, and beta values using direct indexing
    alpha_t = alpha[t]
    alpha_bar_t = alpha_bar[t]
    # Compute epsilon coefficient for mean calculation
    eps_coef = (1 - alpha_t) / (1 - alpha_bar_t) ** 0.5
    # Calculate the mean using the given formula (note the minus sign)
    mean = xt / alpha_t ** 0.5 - eps_coef * noise / alpha_t ** 0.5
    # Extract the variance (beta) using direct indexing
    var = beta[t]
    # Generate random noise with the same shape as xt
    eps = torch.randn_like(xt)
    # Compute the final value using the mean and scaled variance
    return mean + var ** 0.5 * eps
```

فرآیند تولید را برای یک دنباله زمانی از داده ها نشان می دهد. با توجه به حالت فعلی xt ، نویز و مرحله زمانی t، تابع میانگین و واریانس حالت بعدی را محاسبه می‌کند.

reverse step loop:

```
# Iterate over the range of n_steps
for i in range(n_steps):
    # Calculate the time step in reverse order
    t = torch.tensor(n_steps - i - 1, dtype=torch.long).cuda()

    # Use unet to predict noise for the current time step
    with torch.no_grad():
        pred_noise = unet(x.float(), t.unsqueeze(0))

    # Update the input data x using the predicted noise and time step
    x = p_xt(x, pred_noise, t.unsqueeze(0))

    # Append the image representation of x to ims every 24 steps
    if i % 24 == 0:
        ims.append(tensor_to_image(x.cpu()))
```

مرحله زمانی را به ترتیب معکوس محاسبه کنید.

از unet برای پیش بینی نویز برای مرحله زمانی فعلی استفاده می کند.

داده های ورودی x را با استفاده از نویز و مرحله زمانی پیش بینی شده به روز می کند.

خروجی:

شکل ۲۶ - **Reverse Step**

مشاهده می کنیم که در هر مرحله تصویر denoise شده.

تصاویر تولید شده:

شکل ۲۷ - تصاویر تولید شده

(سوال ۱۶)

همانند سوال اول sampleهای اصلی و generate شده را در فولدری ذخیره می کنیم و با دستور زیر fid را محاسبه می کنیم.

```
os.makedirs('generated_images', exist_ok=True)
# Generate and save 3000 samples
for k in tqdm(range(3000)):
    x = torch.randn(1, 3, 32, 32).cuda() # Start with random noise
    for i in range(n_steps):
        t = torch.tensor(n_steps - i - 1, dtype=torch.long).cuda()
        with torch.no_grad():
            pred_noise = unet(x.float(), t.unsqueeze(0))
            x = p_xt(x, pred_noise, t.unsqueeze(0))

    # Save each generated image with a unique filename
    save_image(x, f'content/generated_images/sample_{k}.png')
os.makedirs('cifar10_real_images', exist_ok=True)

# Load CIFAR-10 dataset
transform = transforms.Compose([
    transforms.ToTensor()
])
cifar10_dataset = torchvision.datasets.CIFAR10(root='./data', train=True,
download=True, transform=transform)

# and save 3000 real samples
for i, (image, _) in enumerate(cifar10_dataset):
    torchvision.utils.save_image(image,
f'cifar10_real_images/cifar10_{i:04d}.png')
    if i >= 2999:
        break
```

Compute the FID Score

```
from pytorch_fid.fid_score import calculate_fid_given_paths

# Paths to the generated images and real images
path_to_generated_images = 'generated_images'
path_to_real_images = 'cifar10_real_images'

# Calculate the FID score
fid_value = calculate_fid_given_paths([path_to_generated_images,
path_to_real_images],
                                      batch_size=50,
                                      device='cuda',
                                      dims=2048)
```

```
print(f'FID score: {fid_value}')
```

FID score: 167.20400019331666