به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

مدل‌های مولد عمیق

تمرین شماره ۲ بخش ۲

نام و نام خانوادگی	علیرضا حسینی
شماره دانشجویی	۸۱۰۱۰۱۱۴۲
تاریخ ارسال گزارش	۱۴۰۲/۰۹/۲۵

فهرست گزارش سوالات

سوال 1 – GAN

A) عملگر Pixel Shuffle

PixelShuffle تکنیکی است که در شبکه‌های متخاصم مولد (GAN) و سایر مدل‌های یادگیری عمیق برای ارتقاء یا افزایش وضوح فضایی یک تصویر استفاده می‌شود. این اولین بار در مقاله ای توسط شی و همکاران در سال ۲۰۱۶ معرفی شد که جهت انجام عملیات super resolution بوده است.[1]

عملکرد اصلی nn.PixelShuffle (k) این است که تصاویر را با ضریب k در هر بعد افزایش می دهد و در نتیجه وضوح فضایی آنها را افزایش می دهد. این عملکرد را با مرتب کردن مجدد پیکسل ها در مجموعه ای از کانال ها در یک تصویر با وضوح پایین انجام می دهد تا تصویری با وضوح بالاتر تولید کند.

در ادامه در مورد PixelShuffle و اثرات آن آمده است:

- Upscaling: PixelShuffle برای افزایش مقیاس یک تصویر با ضریب k در هر بعد استفاده می شود که به طور موثر وضوح فضایی را تا k به توان ۲ برابر افزایش می دهد.
- عملیات: به عنوان مثال در شرایط k=2 این لایه یک تصویر با وضوح پایین با ابعاد (batch size ، channel، height، width)، که در آن کانال‌ها مضرب 4 است را به عنوان ورودی می‌گیرد و یک تصویر با ابعاد (batch size، channel/4، 2*height، 2*width) را خروجی می‌دهد. . پیکسل ها را در هر کانال مرتب می کند تا تصویری با وضوح بالاتر تولید کند.
- استفاده در GAN ها: جدای از کاربرد اصلی آن ها در تسک های super resolution در GAN ها، PixelShuffle اغلب در شبکه ژنراتور، به ویژه در لایه خروجی استفاده می شود. این به تولید تصاویر با وضوح بالا از ورودی های با وضوح پایین کمک می کند. برای مثال، اگر GAN وظیفه تولید چهره‌های با وضوح بالا از ورودی‌های با وضوح پایین را داشته باشد، PixelShuffle می‌تواند برای ارتقای ویژگی‌های آموخته‌شده توسط شبکه استفاده شود و در نتیجه خروجی دقیق‌تر و از نظر بصری دلپذیرتر به دست آید.
- مزایا: مزیت اصلی استفاده از PixelShuffle این است که از نظر محاسباتی در مقایسه با تکنیک‌های افزایش مقیاس سنتی مانند درون یابی دو خطی یا غیره کارآمد است. این به شبکه

[1] Shi, Wenzhe, et al. "Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016.

اجازه می‌دهد تا فرآیند نمونه‌برداری را مستقیماً از داده‌ها یاد بگیرد، که می‌تواند منجر به نتایج بهتر شود.

B) تکمیل ساختار Generator و discriminator

با توجه به ساختار داده شده زیر میتوان کد بخش Generator را به صورت زیر پیاده سازی کرد.

	Sequential Blocks	In_Channels	Out_Channels	Batch Norm., Stride,Padding
1	Linear	64	512	BN
	ReLU	-	-	-
2	Linear	512	?!	BN
	ReLU	-	-	-
3	PixelShuffle	-	-	-
4	Conv 3*3	16	32	BN , p=1
	ReLU	-	-	-
5	PixelShuffle	-	-	-
6	Conv 3*3	8	?!	p=1

شکل 1: معماری **Generator**

با توجه به معماری فوق برای علامت سوال اول بلوک دوم و z-dim برابر ۶۴ و تصاویر دیتاست که 28*28 میباشد باید خروجی آن 64*(28/4)*(28/4) باشد و برای علامت سوال دوم نیز خروجی num_channel میباشد.

بدین ترتیب داریم :

```
class Generator(nn.Module):
    def __init__(self, z_dim=64, num_channels=1):
        super().__init__()
        self.z_dim = z_dim

        # Define the generator layers
        self.layers = nn.Sequential(
            # Block 1: Linear -> BatchNorm
            nn.Linear(z_dim, 512),
            nn.BatchNorm1d(512),
            ReLU(),

            # Block 2: Linear -> BatchNorm
```

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

            # Block 3: Pixel Shuffle
            Reshape(64, 7, 7),  # Reshape to (16, 7, 7) before Pixel Shuffle
            nn.PixelShuffle(2),

            # Block 4: Conv -> BatchNorm
            nn.Conv2d(16, 32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            ReLU(),

            # Block 5: Pixel Shuffle
            nn.PixelShuffle(2),

            # Block 6: Conv
            nn.Conv2d(8, num_channels, kernel_size=3, padding=1)
        )

    def forward(self, x):
        return self.layers(x)
```

به همین ترتیب برای discriminator نیز با توجه به تصوی پر واضح است که علامت سوال همان 7*7*64 میشود.

	Sequential Blocks	In_Channels	Out_Channels	Stride & Padding
1	Conv 4*4	1	32	S=2 , p=1
	ReLU	-	-	-
2	Conv 4*4	32	64	S=2 , p=1
	ReLU	-	-	-
3	Linear	?!	512	-
	ReLU	-	-	-
4	Linear	512	1	-

شکل 2 : معماری **discriminator**

کد پایتون پیاده سازی آن به شرح زیر میباشد.

```
class Discriminator(nn.Module):
    def __init__(self, num_channels=1):
        super().__init__()

        # Define the discriminator layers
        self.layers = nn.Sequential(
            # Block 1: Conv -> ReLU
            nn.Conv2d(num_channels, 32, kernel_size=4, stride=2, padding=1),
            ReLU(),

            # Block 2: Conv -> ReLU
            nn.Conv2d(32, 64, kernel_size=4, stride=2, padding=1),
            ReLU(),

            # Flatten the output
            nn.Flatten(),

            # Block 3: Linear -> ReLU
            nn.Linear(7 * 7 * 64, 512),
            ReLU(),

            # Block 4: Linear
            nn.Linear(512, 1)
        )

    def forward(self, x):
        return self.layers(x)
```

لازم به ذکر که است در ۲ کلاس فوق از ۲ تابع reshape و ReLU داده شده در صورت سوال که به شرح زیر میباشد استفاده شده است.

```
class Reshape(torch.nn.Module):
    def __init__(self, *shape):
        super().__init__()
        self.shape = shape
    def forward(self, x):
        return x.reshape(x.size(0), *self.shape)


class ReLU(torch.nn.Module):
    def __init__(self):
        super().__init__()
    def forward(self, x):
        return torch.maximum(x, torch.zeros_like(x))
```

C) پیاده سازی تابع خطای GAN

با توجه به توضیحات صورت سوال باید در این بخش تابع خطای زیر پیاده سازی شود.

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

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

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 3 : تابع خطای **GAN**

اگر بخواهیم تابع خطا را به صورت دستی پیاده سازی کنیم باید گفت که با توجه به فرمول داده شده تابع خطا را میشود با توابع mean و log و ... پیاده کرد.

اما باید توجه داشت که عبارت فوق همان BCE یا binary cross entropy میباشد که برای پیاده سازی با آن ۲ رویکرد داریم یکی اینکه از BCE استفاده کنیم ولی باید در آخر هر ۲ مدل یک nn.sigmoid() قرار دهیم تا خروجی را احتمالاتی بدهد یا از BCE with logits استفاده کنیم که عملا همان کار را میکند و خودش sigmoid را اضافه میکند.

با توضیحات فوق کلاس GAN را تکمیل میکنیم. عملیات به شرح زیر میباشد.

```
class GAN(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_nonsaturating(self, x_real, *, device):
        '''
        Input Arguments:
        - x_real (torch.Tensor): training data samples (64, 1, 28, 28)
        - device (torch.device): 'cpu' by default

        Returns:
        - d_loss (torch.Tensor): nonsaturating discriminator loss
        - g_loss (torch.Tensor): nonsaturating generator loss
        '''

        # Generate fake data
```

```
        z = torch.randn((x_real.size(0), self.z_dim), device=device)
        x_fake = self.g(z)

        # Discriminator loss
        logits_real = self.d(x_real)
        logits fake = self.d(x fake.detach())

        d_loss_real = F.binary_cross_entropy_with_logits(logits_real,
torch.ones_like(logits_real))
        d_loss_fake = F.binary_cross_entropy_with_logits(logits_fake,
torch.zeros_like(logits_fake))
        d_loss = d_loss_real + d_loss_fake

        # Generator loss
        logits_fake = self.d(x_fake)
        g_loss = F.binary_cross_entropy_with_logits(logits_fake,
torch.ones_like(logits_fake))

        return d_loss, g_loss
```

در کلاس فوق ،

- داده های fake (x_fake) با نویز تصادفی (z) از طریق ژنراتور (self.g) تولید میشود.
- (logits_real) برای داده‌های واقعی (x_real) و (logits_fake) برای داده‌های جعلی تولید شده (x_fake) محاسبه می‌شود.
- (d_loss) را به عنوان مجموع دو تلفات متقابل آنتروپی باینری محاسبه می‌شود: یکی برای داده‌های واقعی (d_loss_real) و دیگری برای داده‌های جعلی (d_loss_fake). هدف تمایزگر تشخیص داده های واقعی و جعلی است.
- اتلفات مولد (g_loss) را به عنوان تلفات آنتروپی متقاطع باینری بین داده‌های جعلی تولید شده و برچسب هدف محاسبه می‌شود که روی همه آنها تنظیم شده است. هدف مولد تولید داده‌هایی است که متمایزکننده نتواند آن‌ها را از داده‌های واقعی تشخیص دهد.
- در نهایت، هم تابع هر 2 لاس را برمی گرداند.

D) آموزش مدل

قبل از آموزش مدل نیاز است تا دیتاست مورد نظر لود شود که در اینجا از دیتاست MNIST استفاده شده است که دانلود و پیش‌پردازش و لود کردن آن به کمک کد زیر انجام شده است.

```
preprocess = transforms.ToTensor()
train_loader = torch.utils.data.DataLoader(
```

```
            datasets.MNIST(root='./data', train=True, download=True, transform=preprocess),
            batch_size=128,
            shuffle=True
        )
test_loader = torch.utils.data.DataLoader(
            datasets.MNIST(root='./data', train=False, download=True,
transform=preprocess),
            batch_size=128,
            shuffle=True
        )

# Create pre-processed training and test sets
X_train = train_loader.dataset.data.to(device).reshape(-1, 784).float() / 255
y_train = train_loader.dataset.data.to(device)
X_test = test_loader.dataset.data.to(device).reshape(-1, 784).float() / 255
y_test = test_loader.dataset.data.to(device)
```

مورد بعدی تنظیم device مطمئن شدن از وجود GPU میباشد که به کمک کد های ابتدا این کار انجام شده است.

در این مسئله از سیستم با مشخصات زیر استفاده شده است که از GPU دوم آن یعنی cuda:1 در تمامی بخش ها استفاده شده است و در صورتی که بخواهید آن را بر روی سیستم دیگری که تنها یک GPU دارد ران کنید باید تمامی 1 ها تبدیل به صفر شود.

برای حلقه ترین کد زیر استفاده شده است که توضیحات آن در ادامه آمده است.

```
# Lists to keep track of progress
img list = []
G_losses = []
D_losses = []
iters = 0

class Trainer:
    def __init__(self,
                 model,
                 optimizers,
                 device="cuda:1",
                 iter_max=10000,
                 iter_save=1000,
                 num_latents=100,
                 out_dir=""
             ):
        self.model = model
        self.optimizers = optimizers
```

```
        self.device = device
        self.iter_save = iter_save
        self.iter_max = iter_max
        self.out_dir = out_dir
        self.num_latents = num_latents

        self.G_losses = []  # Store generator losses
        self.D_losses = []  # Store discriminator losses
        self.img_list = []  # Store generated images during training

        # fix visualization latents
        self.z_test = torch.randn(100, self.num_latents).to(device)

    def viz(self, global_step=1):
        with torch.no_grad():
            generator = self.model.g
            generator.eval()
            fake = ((generator(self.z_test) + 1) / 2.)
            self.img_list.append(fake)
            generator.train()

            # Show generated images
            vutils.save_image(fake, '%s/fake_%04d.png' % (self.out_dir, global_step),
nrow=10, padding=2, normalize=True)


    def checkpoint_and_log(self, global_step, loss, summaries):
        if global_step % self.iter_save == 0:
            with torch.no_grad():
                self.viz(global_step)
                torch.save((self.model.g, self.model.d), '%s/model_%04d.pt' %
(self.out_dir, global_step))

        # Print generator and discriminator losses
        if global_step % self.iter_save == 0:
            print(f"Iteration {global_step}:")
            print(f"Generator Loss: {loss['generator_loss']:.4f}")
            print(f"Discriminator Loss: {loss['discriminator_loss']:.4f}")


    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}, None

    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):
            while global_step < 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

    def generate_fake_images(self, num_images):
```

```
        z = torch.randn(num_images, self.model.z_dim).to(self.device)
        with torch.no_grad():
            fake_images = self.model.g(z)
        return fake_images



    def evaluate_fid(self, test_loader):
        # Load the Inception model
        inception_model = inception_v3(pretrained=True,
transform_input=False).to(self.device)
        inception_model.eval()

        # Generate a batch of fake images
        fake_images = self.generate_fake_images(num_images=128)  # Adjust the batch size as
needed

        # Get a batch of real images
        real_images, _ = next(iter(test_loader))
        real_images = real_images.to(self.device)

        # Resize images for Inception model
        real_images = F.interpolate(real_images, size=(299, 299), mode='bilinear',
align_corners=False)
        fake_images = F.interpolate(fake_images, size=(299, 299), mode='bilinear',
align_corners=False)

        # Compute FID score
        fid_value = calculate_fretchet(real_images, fake_images, inception_model)
        print(f"FID Score: {fid_value}")
        return fid_value

    def plot_losses(self):
        plt.figure(figsize=(15, 5))  # Increase the figure size for three subplots
        iterations = list(range(0, len(self.D_losses) * self.iter_save, self.iter_save))

        # Subplot 1: Generator Loss
        plt.subplot(1, 3, 1)
        plt.plot(iterations, self.G_losses, label="Generator Loss", marker='o',
linestyle='-')
        plt.xlabel("Iteration")
        plt.ylabel("Loss")
        plt.title("Generator Loss")
        plt.grid(True)
        plt.legend()

        # Subplot 2: Discriminator Loss
        plt.subplot(1, 3, 2)
```

```
        plt.plot(iterations, self.D_losses, label="Discriminator Loss", marker='o',
linestyle='-')
        plt.xlabel("Iteration")
        plt.ylabel("Loss")
        plt.title("Discriminator Loss")
        plt.grid(True)
        plt.legend()

        # Subplot 3: Both Losses
        plt.subplot(1, 3, 3)
        plt.plot(iterations, self.G_losses, label="Generator Loss", marker='o',
linestyle='-', color='blue')
        plt.plot(iterations, self.D_losses, label="Discriminator Loss", marker='o',
linestyle='-', color='red')
        plt.xlabel("Iteration")
        plt.ylabel("Loss")
        plt.title("Generator and Discriminator Losses")
        plt.grid(True)
        plt.legend()

        plt.tight_layout()
        plt.show()
```

در کد فوق،

__init__(...): Trainer را با پارامترهای مختلف، از جمله مدل GAN، بهینه سازها، دستگاه، تنظیمات تکرار آموزش و دایرکتوری خروجی، مقداردهی اولیه می کند. همچنین لیست ها را برای لاس مولد و متمایز کننده (self.G_losses و self.D_losses) و لیستی برای ذخیره تصاویر تولید شده (self.img_list) مقداردهی می کند.

viz(...): با استفاده از مولد تصاویر جعلی تولید می کند و آنها را در لیست مشخص شده ذخیره می کند. از مجموعه ثابتی از نویز تصادفی (self.z_test) برای تولید تصاویر استفاده می کند.

checkpoint_and_log(...): این تابع وظیفه چک پوینت مدل و ثبت لاس در حین آموزش را بر عهده دارد. مدل را ذخیره می کند و تصاویر تولید شده را در فواصل زمانی مشخص (self.iter_save) رسم می کند. همچنین تلفات مولد و تفکیک کننده را پرینت میکند.

gan_step(...): یک مرحله از فرآیند آموزش GAN را پیاده سازی می کند. این تلفات متمایز کننده و مولد را با استفاده از روش loss_nonssaturating مدل GAN محاسبه می کند. سپس، وزن مولد و تفکیک

کننده را با استفاده از بهینه سازهای مربوطه به روز می کند. لاس های نیز به لیست های لاس هاAppend می شوند.

train(...): حلقه آموزشی اصلی در این تابع قرار دارد. روی داده های آموزشی (train_loader) تکرار می شود و gan_step را برای هر دسته فراخوانی می کند. این مرحله (global_step) را پیگیری می کند و نوار پیشرفت را به روز می کند. آموزش تا رسیدن به تکرار self.iter_max ادامه دارد.

plot_losses(...): این تابع تلفات مولد و متمایز را در طول تکرار ترسیم و نمایش می دهد. از Matplotlib برای ایجاد سه قطعه فرعی استفاده می‌کند: یکی برای لاس مولد، یکی برای لاس متمایزکننده و دیگری برای هر دو تلفات با هم.

در نهایت نیز به کمک کد زیر optimizer ها تعریف و مدل ساخته شده و فرایند آموزش انجام میشود.

```
def build model(device='cpu', num latents=64):
    model = GAN(z_dim=num_latents)
    return model.to(device)


def build_optimizers(model):
    g_opt = torch.optim.Adam(model.g.parameters(), lr=1e-3)
    d_opt = torch.optim.Adam(model.d.parameters(), lr=1e-3)
    optimizers = [g_opt, d_opt]
    return optimizers


num_latents = 64
device = torch.device('cuda:1' if torch.cuda.is_available() else 'cpu')
model = build_model(device, num_latents=num_latents)
optimizers = build_optimizers(model)


trainer = Trainer(model, optimizers,
                    device=device,
                    iter_max=10000,
                    num_latents=num_latents,
                    out_dir='./result_gan'
                  )

trainer.train(train_loader)

trainer.viz()
```

تصویر زیر مشخصات سیستم استفاده شده جهت آموزش شبکه میباشد.

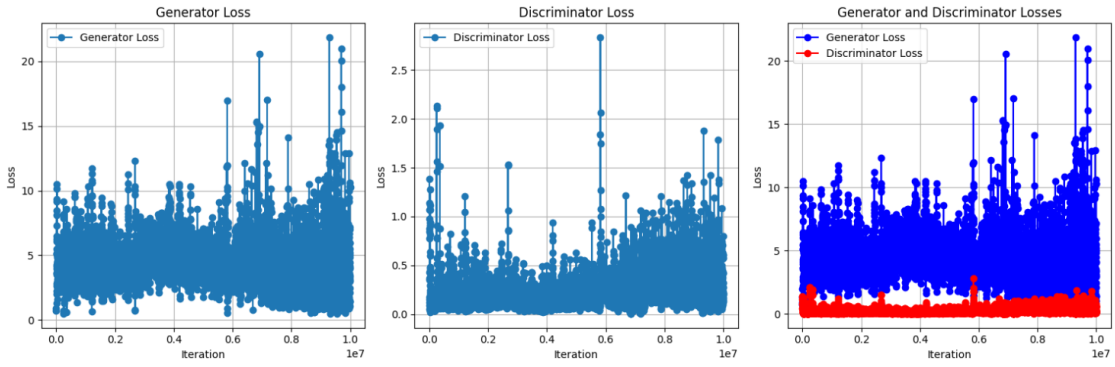
```
              total        used        free      shared  buff/cache   available
Mem:          64129        7087        4638          18       52403       56311
Swap:          8191        1092        7099
Filesystem      Size  Used Avail Use% Mounted on
overlay         916G  874G     0 100% /
tmpfs            64M     0   64M   0% /dev
tmpfs            32G     0   32G   0% /sys/fs/cgroup
shm              64M     0   64M   0% /dev/shm
/dev/sda1       3.7T  2.1T  1.7T  55% /code/Datasets
/dev/nvme0n1p2  916G  874G     0 100% /code/content
tmpfs            32G   12K   32G   1% /proc/driver/nvidia
tmpfs           6.3G  2.8M  6.3G   1% /run/nvidia-persistenced/socket
udev             32G     0   32G   0% /dev/nvidia0
tmpfs            32G     0   32G   0% /proc/asound
tmpfs            32G     0   32G   0% /proc/acpi
tmpfs            32G     0   32G   0% /proc/scsi
tmpfs            32G     0   32G   0% /sys/firmware
Fri Dec 15 10:18:54 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.12             Driver Version: 535.104.12   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA GeForce RTX 3090        Off | 00000000:01:00.0 Off |                  N/A |
| 38%   66C    P2             117W / 370W |   4820MiB / 24576MiB |      3%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+
|   1  NVIDIA GeForce RTX 3090        Off | 00000000:03:00.0 Off |                  N/A |
| 36%   44C    P8              18W / 370W |      2MiB / 24576MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
```

شکل 4 : مشخصات سیستم استفاده شده برای این سوال

E) ارزیابی مدل آموزش داده شده

ابتدا به کمک تابع پلات که در ترینر قرار داده شده است مقادیر لاس برای مولد و متمایزکننده را رسم میکنیم.

```
trainer.plot_losses()
```



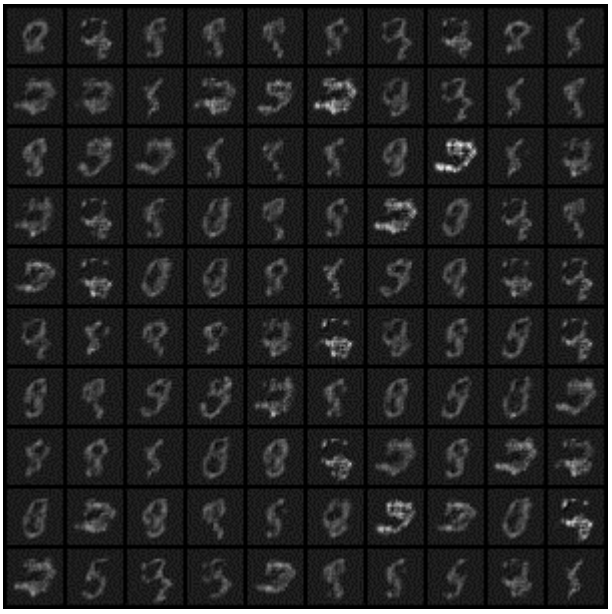
شکل 5 : محنی لاس برای مولد و متمایزکننده پس از آموزش **GAN**

همانطور که میدانید GAN ها برخلاف سایر شبکه هات بدان صورت نیستند که یک لاس کاملا تمیز نزولی داشته باشند و اینگونه نوسانات بدلیل رد و بدل شدن بین مولد و متمایز کننده تا هرکدام نسبت به دیگری خود را قوی تر کنند کاملا طبیعی میباشد و در سایر شبکه های GAN هم چنین منحنی های loss طبیعی میباشد.

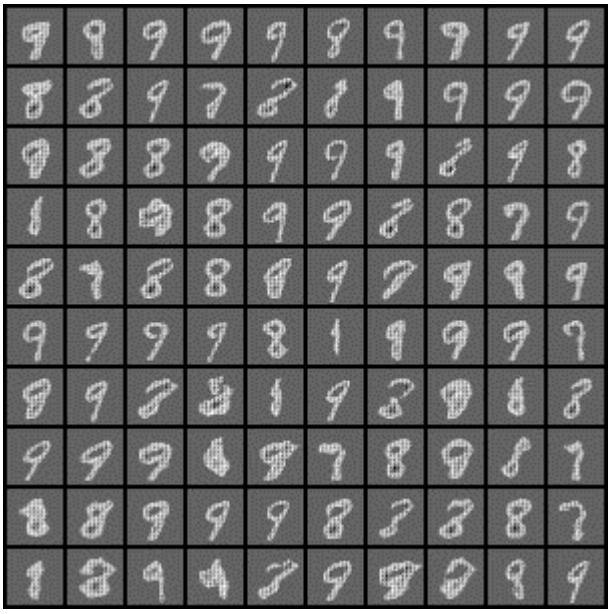
یک نمونه خروجی مدل آموزش داده شده پس از epoch 10000 به صورت زیر میباشد که نشان میدهد شبکه به خوبی آموزش داده شده است.

شکل 6 : خروجی شبکه آموزش داده شده **GAN** در ایپاک 10000

به همین ترتیب برای مشاهده روند خروجی های شکل های زیر به ترتیب خروجی در ایپاک های 1000 و 5000 که ابتدا و میانه های آموزش میباشد را نشان میدهد.



شکل 7 : خروجی شبکه **GAN** پس از 1000 ایپاک



شکل 8 : خروجی شکه **GAN** پس از 5000 ایپاک

E-2) **معیار** FID

E2-1 تعریف

FID (Fréchet Inception Distance) معیاری است که برای ارزیابی کیفیت تصاویر تولید شده توسط شبکه های متخاصم (GAN) استفاده می شود. توسعه یافته توسط هوسل و همکاران. در مقاله 2017 خود، شباهت بین توزیع تصاویر تولید شده و تصاویر واقعی را اندازه گیری می کند. در اینجا نحوه کار آن آمده است:

- استخراج ویژگی: FID از یک مدل یادگیری عمیق به نام InceptionV3 برای استخراج ویژگی ها از تصاویر تولید شده و واقعی استفاده می کند. این مدل در ImageNet آموزش داده شده است و می تواند جنبه های مختلف تصاویر مانند بافت ها، اشیاء و اشکال را شناسایی کند.
- مقایسه آماری: ویژگی های استخراج شده برای محاسبه میانگین و کوواریانس برای هر دو تصویر واقعی و تولید شده استفاده می شود. این معیارهای آماری یک توزیع گاوسی چند بعدی را برای هر مجموعه ای از تصاویر تشکیل می دهند.
- محاسبه فاصله: امتیاز FID سپس با استفاده از فاصله Fréchet (همچنین به عنوان فاصله Wasserstein-2 شناخته می شود) بین این دو توزیع گاوسی محاسبه می شود. این فاصله اندازه گیری می کند که دو توزیع در فضای ویژگی چقدر از هم فاصله دارند.

هر چه امتیاز FID کمتر باشد، توزیع تصاویر تولید شده با تصاویر واقعی مشابه تر است که نشان دهنده کیفیت بهتر تصاویر تولید شده است. امتیاز FID بالا نشان می دهد که تصاویر تولید شده از کیفیت پایین تری برخوردار هستند، یا به این دلیل که شباهت زیادی به تصاویر واقعی ندارند یا به دلیل عدم تنوع.

E2-2 پیاده سازی

رویکرد های متفاوتی برای محاسبه میتوان در نظر گرفت که در اینجا ۳ رویکرد مطرح شده است.

E2-2-1 استفاده از کتاب خانه آماده FID TORCH

این رویکرد میتواند ساده ترین رویکرد ممکن باشد بدین صورت که ابتدا تعدادی تصاویر توسط مدل تولید میشود و تصاویر اصلی هم در فولدری ذخیره میشود و به کمک دستورات زیر توسط پکیج های آماده محاسبه FID انجام میشود.[1] (جهت تنوع در سوال ۱ از این روش و در سوال ۲ از رویکرد سوم که پیاده سازی از scratch میباشد استفاده شده است اما نتایج با هم یکی بوده و فرقی ندارد)

```
!pip install pytorch-fid
!python -m pytorch_fid  'Real_Image_folder_path'  'generated_images_folder_path' --device
cuda:1
```

E2-2-2 استفاده از کتاب خانه آماده Torch-Fidelity

رویکرد دوم مانند رویکرد اول بوده و از کتابخانه های آماده استفاده میکند و کافیست ۲ مسیر را به کد زیر بدهیم.[2]

[1] https://pypi.org/project/pytorch-fid/

[2] https://github.com/toshas/torch-fidelity

E2-2-3 پیاده سازی از ابتدا

در این بخش هدف پیاده سازی کل الگوریتم FID میباشد که در ادامه توضیحات آن آمده شده است.

(برای تنوع در سوال ۲ از این روش استفاده شده است)

```
import torch
from torchvision.models import inception_v3
import torchvision.transforms as transforms
from scipy.linalg import sqrtm
import numpy as np
import glob
from PIL import Image

# Function to load images and convert to Inception-friendly format
def load_images_for_inception(path):
    transform = transforms.Compose([
        transforms.Resize((299, 299)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
    ])
    images = []
    for filename in glob.glob(f'{path}/*.png'):
        with open(filename, 'rb') as f:
            image = Image.open(f)
            image = transform(image).unsqueeze(0)
            images.append(image)
    return torch.cat(images, dim=0)

# Function to calculate activations using InceptionV3
def get_activations(images, model):
    with torch.no_grad():
        pred = model(images)
    return pred.detach().cpu().numpy()

# Load pretrained InceptionV3 model
inception_model = inception_v3(pretrained=True, transform_input=False)
inception_model.eval()
print("model loaded")

def calculate_fid(act1, act2):
    # Calculate mean and covariance
    mu1, sigma1 = act1.mean(axis=0), np.cov(act1, rowvar=False)
    mu2, sigma2 = act2.mean(axis=0), np.cov(act2, rowvar=False)

    # Calculate sum squared difference in means
```

```
    ssdiff = np.sum((mu1 - mu2)**2.0)

    # Calculate sqrt of product between cov
    covmean = sqrtm(sigma1.dot(sigma2))

    # Check and correct imaginary numbers from sqrt
    if np.iscomplexobj(covmean):
        covmean = covmean.real

    # Calculate score
    fid = ssdiff + np.trace(sigma1 + sigma2 - 2.0 * covmean)
    return fid

# Load and process images
real_images = load_images_for_inception('cifar10_images')
generated_images = load_images_for_inception('generated_images')

# Get activations
act_real = get_activations(real_images, inception_model)
act_generated = get_activations(generated_images, inception_model)

# Calculate FID
fid_score = calculate_fid(act_real, act_generated)
print('FID score:', fid_score)
```

در کد فوق،

- تابع load_images_for_inception(path):

هدف: تصاویر را از یک دایرکتوری مشخص بارگیری می کند، آنها را برای مطابقت با الزامات ورودی مدل InceptionV3 پیش پردازش می کند و تانسوری از این تصاویر را برمی گرداند.
مراحل: اندازه تصاویر را به ۲۹۹*۲۹۹ تغییر می دهد (اندازه ورودی InceptionV3)، آنها را به تانسور PyTorch تبدیل می کند و با استفاده از مقادیر میانگین از پیش تعریف شده و انحراف استاندارد آنها را نرمال می کند.

- تابع get_activations (تصاویر، مدل):

هدف: محاسبه فعال‌سازی (ویژگی‌ها) از دسته‌ای از تصاویر با استفاده از مدل InceptionV3.
روند: این تابع تصاویر را در یک زمینه بدون گرادیان (برای صرفه جویی در حافظه و محاسبات) از طریق مدل عبور می دهد و فعال سازی های خروجی را به صورت آرایه NumPy برمی گرداند.

- بارگیری مدل InceptionV3:

مدل InceptionV3 با وزن های از پیش آموزش دیده بارگذاری شده و روی حالت ارزیابی تنظیم شده است. این مدل برای استخراج ویژگی ها از تصاویر استفاده می شود.

تابع account_fid(act1, act2):

هدف: امتیاز FID را بین دو مجموعه ویژگی را محاسبه می کند.

روند:

میانگین و کوواریانس هر مجموعه از فیچر ها را محاسبه می کند.

مجذور اختلاف میانگین ها و اثر مجموع کوواریانس ها را محاسبه می کند.

جذر حاصل ضرب دو ماتریس کوواریانس را محاسبه می کند.

اگر نتیجه شامل اعداد مختلط باشد، فقط از قسمت واقعی استفاده می شود.

امتیاز FID مجموع مجذور اختلاف میانگین ها و ردیابی ماتریس های کوواریانس است که با حاصلضرب کوواریانس ها تنظیم می شود.

- بارگیری و پردازش تصاویر:
- تصاویر واقعی و تولید شده را با استفاده از تابع load_images_for_inception بارگیری و پردازش می کند.
- به دست آوردن فیچر:

با استفاده از تابع get_activations فیچر ها برای تصاویر واقعی و تولید شده استخراج میشود.

- محاسبه امتیاز FID:

در نهایت، امتیاز FID با استفاده از تابع account_fid با فیچر های تصاویر واقعی و تولید شده محاسبه می شود.

به طور کلی، این کد به طور موثر تصاویر را بارگیری و پردازش می کند، ویژگی های لازم را با استفاده از یک مدل InceptionV3 از پیش آموزش دیده استخراج می کند و امتیاز FID را محاسبه می کند که یک معیار پذیرفته شده برای ارزیابی کیفیت تصاویر تولید شده توسط GAN ها است.

E2-3 ارزیابی مدل با FID

پس از توضیحات فوق حال میتوان معیار FID را برای مدل GAN آموزش دیده به دست آورد. همانطور که در هر ۳ رویکرد دیدیم در وهله اول باید از مدل خروجی گرفته و در فولدر هایی این خروجی ها ذخیره

شود. با توجه به 10k بودن دیتای تست MNIST به کمک کد زیر 10k داده توسط مدل تولید و خود MNIST در فولدر های مربوطه ذخیره میشود.

```
# Set device
device = torch.device("cuda:1" if torch.cuda.is_available() else "cpu")

# Set model to evaluation mode
model.eval()

# Generate fake images
num_samples = 10000
num_latents = 64
output_directory = "generated_images_gan"
os.makedirs(output_directory, exist_ok=True)

batch_size = 2048
for batch_idx in range(num_samples // batch_size):
    # Generate random latent vectors for each batch
    z_fake = torch.randn(batch_size, num_latents, device=device)

    # Generate fake images using the generator
    with torch.no_grad():
        fake_images = model.g(z_fake).detach().cpu()

    # Save generated images for each batch
    for i in range(batch_size):
        image_path = os.path.join(output_directory, f"fake_image_{batch_idx * batch_size +
i + 1}.png")
        save_image(fake_images[i], image_path)

    #print(f"Generated and saved {batch_size} fake images for batch {batch_idx + 1}.")

print(f"Generated and saved a total of {num_samples} fake images to {output_directory}.")

# Create a directory to save the MNIST images
mnist_directory = "mnist_images_gan"
os.makedirs(mnist_directory, exist_ok=True)

# MNIST data loader
mnist_loader = torch.utils.data.DataLoader(
    datasets.MNIST(root="./data", train=False, download=True,
transform=transforms.ToTensor()),
    batch size=10000,  # Use the full test set batch size
    shuffle=True,
)

# Get the full batch of MNIST test images
```

```
mnist_images, _ = next(iter(mnist_loader))
mnist_images = mnist_images.to(device)

# Save MNIST images
for i in range(num_samples):
    image path = os.path.join(mnist directory, f"mnist image {i + 1}.png")
    save_image(mnist_images[i], image_path)

print(f"Saved MNIST Test images to {mnist_directory}.")
```

پس از آن نیز از هر ۳ رویکرد میتوان FID را محاسبه کرد که در اینجا از رویکرد اول به کمک کد زیر میتوان FID را محاسبه کرد.

```
!python -m pytorch_fid  'mnist_images_gan'  'generated_images_gan' --device cuda:1
100%|████████████████████████████████████████| 200/200 [00:10<00:00, 18.88it/s]
100%|████████████████████████████████████████| 164/164 [00:07<00:00, 21.40it/s]
FID:  33.873376711831554
```

مشاهده میشود که مقدار FID برابر ۳۳ میشود.

با توجه به اینکه مدل Generative Adversarial Network (GAN) امتیاز FID= 33 را در مجموعه داده MNIST به دست آورد میتوان این نتیجه را به صورت زیر تحلیل کرد:

مجموعه داده MNIST شامل تصاویرgray Scale از ارقام دست نویس با وضوح 28*28 پیکسل است. این یک مجموعه داده نسبتا ساده و به خوبی مطالعه شده در زمینه یادگیری ماشین است. بنابراین، انتظارات برای نمرات FID به طور کلی بالاتر است (یعنی مقادیر عددی پایین تر) زیرا تولید ارقام دست نویس واقع گرایانه در مقایسه با مجموعه داده های پیچیده تر مانند CIFAR-10 یا ImageNet کار پیچیده تری است.

تفسیر امتیاز 33: در زمینه MNIST، امتیاز 33 FID نشان می دهد که کیفیت و تنوع ارقام تولید شده متوسط است اما استثنایی نیست. از آنجایی که MNIST یک مجموعه داده پایه است، مدل های پیشرفته اغلب به امتیازات FID بسیار پایین تری دست می یابند که نشان دهنده تولید رقم دقیق تر و متنوع تر است.

محک زدن در برابر سایر مدل‌ها: مقایسه این امتیاز FID با امتیازهای بدست آمده توسط سایر مدل‌های GAN که در مجموعه داده‌های MNIST آموزش دیده‌اند، مهم است. در قلمرو MNIST، به دلیل سادگی مجموعه داده، امتیازات FID پایین نسبتاً رایج است. بنابراین، نمره 33 ممکن است نشان دهد که فضای قابل توجهی برای بهبود در مدل وجود دارد.

پتانسیل برای بهبود: با توجه به سادگی مجموعه داده MNIST، امتیاز 33 نشان می دهد که مدل ممکن است با گرفتن انواع سبک ها در ارقام دست نویس یا تولید تصاویر بدون مصنوعات قابل توجه مشکل داشته باشد. بهبود در معماری مدل، روش آموزشی، یا تنظیم هایپرپارامتر به طور بالقوه می تواند نتایج بهتری به همراه داشته باشد.

کیفیت ارقام تولید شده: ارزش بازرسی بصری ارقام تولید شده را نیز دارد. گاهی اوقات، نمرات FID ممکن است درک انسان از کیفیت تصویر را به طور کامل نشان ندهد. اگر ارقام تولید شده به راحتی قابل تشخیص و متنوع باشند، این مدل ممکن است علیرغم امتیاز عددی همچنان برای کاربردهای خاصی مفید باشد که دراین جا با توجه به تصاویر تولیدی تا حدی صادق است.

F) مقایسه و ارزیابی مدل های مختلف GAN

تکامل شبکه‌های متخاصم مولد (GAN) با معرفی مدل‌های پیشرفته مختلف مشخص شده است که هر کدام به محدودیت‌های خاص GAN اصلی می‌پردازند. در اینجا مروری بر این مدل‌ها، مشکلاتی که آنها به آن پرداخته‌اند، و نحوه کمک به بهبود GANها آورده شده است:

:Wasserstein GAN (WGAN)

- ✓ مشکل رفع شده: GAN اصلی از بی ثباتی آموزشی و فروپاشی state رنج می برد، که در آن ژنراتور فقط تعداد محدودی از خروجی ها را تولید می کرد.
- ✓ راه حل: WGAN فاصله Wasserstein را معرفی کرد، یک هدف آموزشی پایدارتر که خواص همگرایی فرآیند آموزش را بهبود می بخشد. این منجر به یک رفتار آموزشی پایدارتر می شود و به کاهش فروپاشی State کمک می کند.
- ✓ فرمول/مفهوم: ایده کلیدی استفاده از فاصله حرکت دهنده زمین (یا فاصله -Wasserstein 1) برای اندازه گیری تفاوت بین توزیع داده های تولید شده و واقعی است.

:(PG GAN)

- ✓ مشکل حل شده: مشکل در تولید تصاویر با وضوح بالا.
- ✓ راه حل: PG GAN با افزودن تدریجی لایه ها به ژنراتور و تفکیک کننده در طول فرآیند آموزش، وضوح تصاویر تولید شده را به تدریج افزایش می دهد. این رویکرد به شبکه اجازه می‌دهد ابتدا ساختار در مقیاس بزرگ را بیاموزد و سپس تمرکز را به جزئیات دقیق‌تر تغییر دهد، که منجر به تولید تصویر با کیفیت و وضوح بالا می‌شود.

- ✓ فرمول/مفهوم: این مدل با تصاویر با وضوح پایین شروع می‌شود و به تدریج لایه‌هایی را برای افزایش وضوح اضافه می‌کند و امکان یادگیری جزئیات بیشتری را در هر مرحله فراهم می‌کند.

:BigGAN

- ✓ مشکل حل شده: چالش‌هایی در افزایش مقیاس GAN برای تولید تصویر در مقیاس بزرگ با fidelity بالا.
- ✓ راه حل: BigGAN از معماری در مقیاس بزرگ و آموزش بر روی مجموعه داده های گسترده برای تولید تصاویر بسیار واقعی و متنوع با وضوح بالا استفاده می کند. چندین تکنیک مانند اندازه های بزرگ دسته ای و منظم سازی متعامد را برای پایداری آموزش معرفی می کند.
- ✓ فرمول/مفهوم: BigGAN بر استفاده از اندازه های بزرگ دسته ای و افزایش اندازه مدل و اندازه داده های آموزشی، همراه با تکنیک های منظم سازی خاص برای پایداری تأکید دارد.

:StyleGAN

- ✓ مشکل حل شده: کنترل محدود بر سبک و ویژگی های تصاویر تولید شده.
- ✓ راه حل: StyleGAN یک معماری ژنراتور مبتنی بر سبک را معرفی می کند که امکان کنترل دقیق بر سبک تصویر تولید شده در سطوح مختلف جزئیات را فراهم می کند. این مدل می تواند ویژگی ها را در مقیاس های مختلف دستکاری و ترکیب کند که منجر به تولید تصویر بسیار واقعی و قابل تنظیم می شود.
- ✓ فرمول/مفهوم: از لایه های Adaptive Instance Normalization (AdaIN) در هر سطح از ژنراتور برای کنترل سبک تصویر خروجی در مقیاس های مختلف استفاده می کند.
- ✓ هر یک از این مدل‌ها نشان‌دهنده یک گام به جلو در پرداختن به چالش‌های خاص معماری GAN است که منجر به قابلیت‌های تولید تصویر پایدارتر، باکیفیت‌تر و همه‌کاره‌تر می‌شود.

هر یک از این مدل‌ها نشان‌دهنده یک گام به جلو در پرداختن به چالش‌های خاص معماری GAN است که منجر به قابلیت‌های تولید تصویر پایدارتر، باکیفیت‌تر و ... می‌شود.

G) پیاده سازی WGAN

Wasserstein GAN (WGAN) در مقایسه با GAN اصلی از یک تابع لاس متفاوت استفاده می کند. این تابع از دست دادن بر اساس فاصله Wasserstein است که به عنوان فاصله حرکت دهنده زمین (EM)

نیز شناخته می شود. تابع ضرر WGAN به حداقل رساندن این فاصله بین توزیع داده های واقعی و توزیع داده های تولید شده است. فرمول ضرر WGAN در اینجا آمده است:

$$\text{WGAN Loss} = x\sim\sim P_g E[D(x\sim)] - x\sim P_r E[D(x)]$$

برای پیاده سازی کافیست در کد قبلی فقط تابع loss با عوض کنیم که کد آن به شرح زیر میباشد.

```
class GAN(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_wgan(self, x_real, *, device):
        '''
        Input Arguments:
        - x_real (torch.Tensor): training data samples (64, 1, 28, 28)
        - device (torch.device): 'cpu' by default

        Returns:
        - d_loss (torch.Tensor): WGAN discriminator (critic) loss
        - g_loss (torch.Tensor): WGAN generator loss
        '''

        # Generate fake data
        z = torch.randn((x_real.size(0), self.z_dim), device=device)
        x_fake = self.g(z)

        # Critic (discriminator) loss
        # In WGAN, the critic outputs a scalar (Wasserstein estimate)
        wasserstein_estimate_real = self.d(x_real)
        wasserstein_estimate_fake = self.d(x_fake.detach())

        # The critic aims to maximize the difference between its output on real and fake
data
        d_loss = -(torch.mean(wasserstein_estimate_real) -
torch.mean(wasserstein_estimate_fake))

        # Generator loss
        # The generator aims to minimize the critic's output on its generated data
        wasserstein_estimate_fake = self.d(x_fake)
        g_loss = -torch.mean(wasserstein_estimate_fake)

        return d_loss, g_loss
```

همانطور که مشاهده میشود لاس مطابق با فرمول تعریف شده است. در ادامه یکسری نکات برای پیاده سازی WGAN لازم است که در ادامه آورده شده است.

در پیاده سازی از weight clipping برای جلوگیری از بزرگ شدن وزن های discriminator استفاده شده است که در کد ترین بعد از dmodel optimization انجام میشود.

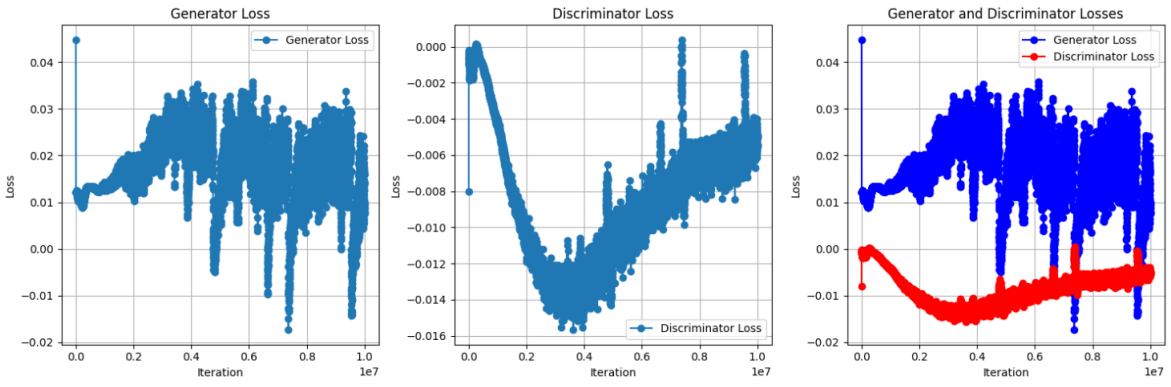
```
        # Update discriminator
        d_optimizer.zero_grad()
        discriminator_loss.backward()
        d_optimizer.step()

        # Weight clipping for the discriminator
        for p in discriminator.parameters():
            p.data.clamp_(-0.01,0.01)

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

برای optimizer نیز از RMSprop که برای WGAN عموما از این استفاده میشود استفاده شده است.

همچنین مقادیر lr نیز کاملا تجربی و به صورت آزمون و خطا به صورت زیر میباشد.

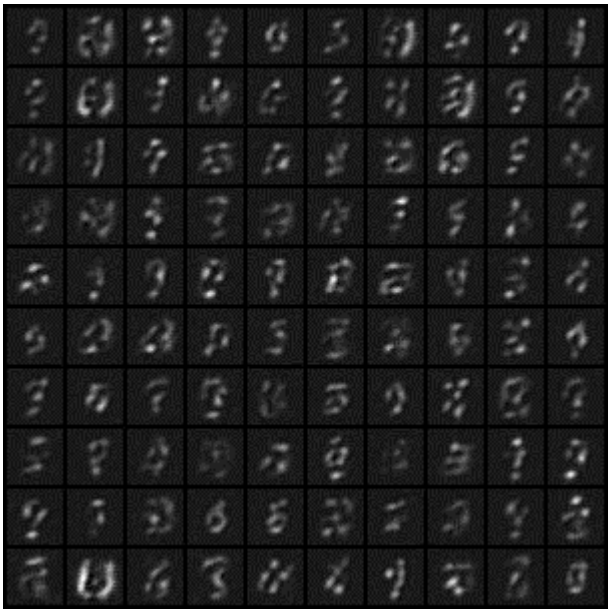
```
g_opt = torch.optim.RMSprop(model.g.parameters(), lr=1e-4)
d_opt = torch.optim.RMSprop(model.d.parameters(), lr=1e-5)
```

منحنی لاس پس از آموزش مدل به شرح زیر است:

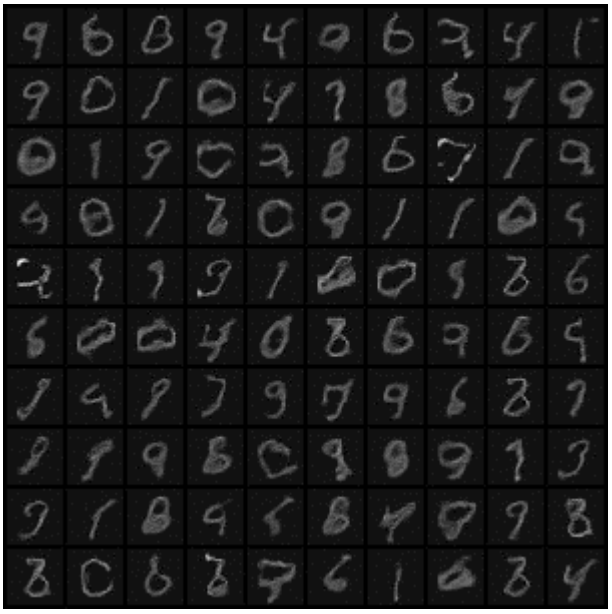


شکل 9 : منحنی لاس آموزش **WGAN**

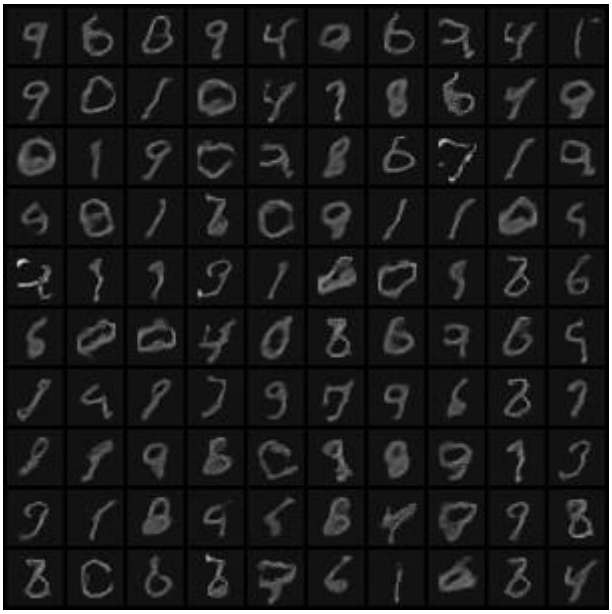
خروجی های مدل به ترتیب در ابتدا میانی و اخرین ایپاک به صورت زیر میباشد.



شکل 10 : خروجی مدل **WGAN** برای **1000** ایپاک



شکل 11 : خروجی مدل **WGAN** برای **5000** ایپاک



شکل 12 : خروجی مدل **WGAN** برای **10000** ایپاک

در نهایت نیز مقدار FID برابر 43 میشود که بهتر از قبل نشد اما با تغییر در هایپرپارامتر ها میتوان آن را بهتر کرد.

اما همچنان WGAN مشکلاتی را دارند.

در حالی که Wasserstein GAN (WGAN) پیشرفت قابل توجهی نسبت به GAN اصلی از نظر پایداری آموزشی داشته، هنوز با چندین چالش و زمینه برای بهبود مواجه است:

Weight Clipping: در WGAN اصلی، محدودیت Lipschitz با برش دادن وزن های discriminator به یک محدوده کوچک (به عنوان مثال، [-0.01، 0.01]) اعمال می شود. با این حال، این رویکرد می تواند منجر به مجموعه ای از مشکلات خاص خود شود، مانند:

- ✓ پتانسیل ناپدید شدن گرادیان، زیرا وزن ها بسیار کوچک نگه داشته می شوند.
- ✓ ظرفیت محدود discriminator ، زیرا clipping می تواند بسیار محدود کننده باشد و مانع از یادگیری عملکردهای پیچیده تر discriminator می شود.
- ✓ سرعت آموزش: WGAN ها می توانند در مقایسه با GAN های سنتی آهسته تر آموزش ببینند زیرا discriminator باید به طور کامل تر آموزش داده شود تا فاصله Wasserstein را به طور دقیق تقریب کند.

برای پرداختن به این مسائل، محققان پیشرفت‌های مختلفی را پیشنهاد کرده‌اند که قابل‌توجه‌ترین آنها Gradient Penalty (GP) است که در WGAN-GP معرفی شده است. این رویکرد برش وزن را با یک جریمه در هنجار گرادیان برای خروجی discriminator با توجه به ورودی آن جایگزین می‌کند.

$$\text{Gradient Penalty} = \lambda \cdot \mathbb{E}_{\hat{x} \sim \mathbb{P}_{\hat{x}}} \left[\left(\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1 \right)^2 \right]$$

Where:

- λ is a penalty coefficient (a hyperparameter).
- $\hat{x}$ are sampled uniformly along straight lines between pairs of points sampled from the real data distribution ($\mathbb{P}_r$) and the generated data distribution ($\mathbb{P}_g$).
- $\nabla_{\hat{x}} D(\hat{x})$ is the gradient of the critic's output with respect to its input $\hat{x}$.
- $\|\cdot\|_2$ denotes the L2 norm.

شکل 13: فرمول بندی **WGAN-GP**

ایده پشت WGAN_GP این است که محدودیت Lipschitz را به شیوه ای موثرتر و کمتر محدودتر از برش وزن اعمال کنیم. اگر هنجار گرادیان از 1 فاصله بگیرد، مدل را جریمه می‌کند و روند آموزشی نرم‌تر و پایدارتر را تضمین می‌کند. این افزایش تمایل به خواص همگرایی بهتر دارد و به یک رویکرد استاندارد در آموزش WGAN تبدیل شده است.

کد loss در این حالت به صورت زیر میشود.

```
def loss_wgan_gp(self, x_real, *, device, lambda_gp=10):
    '''
    Input Arguments:
    - x_real (torch.Tensor): training data samples (64, 1, 28, 28)
    - device (torch.device): 'cpu' by default
    - lambda_gp (float): Gradient penalty regularization factor

    Returns:
    - d_loss (torch.Tensor): WGAN-GP discriminator loss
    - g_loss (torch.Tensor): WGAN-GP generator loss
    '''

    # Generate fake data
    z = torch.randn((x_real.size(0), self.z_dim), device=device)
    x_fake = self.g(z)

    # Discriminator loss
    logits_real = self.d(x_real)
    logits_fake = self.d(x_fake.detach())
```

```
    d_loss = logits_fake.mean() - logits_real.mean()

    # Gradient penalty
    alpha = torch.rand(x_real.size(0), 1, 1, 1, device=device)
    x hat = alpha * x real.data + (1 - alpha) * x fake.data
    x_hat.requires_grad = True
    logits_hat = self.d(x_hat)
    gradients = torch.autograd.grad(outputs=logits_hat, inputs=x_hat,
                                    grad_outputs=torch.ones(logits_hat.size(),
device=device),
                                    create_graph=True, retain_graph=True,
only_inputs=True)[0]
    gradient_penalty = lambda_gp * ((gradients.norm(2, dim=1) - 1) ** 2).mean()

    d_loss += gradient_penalty

    # Generator loss
    logits_fake = self.d(x_fake)
    g_loss = -logits_fake.mean()

    return d_loss, g_loss
```

در این حالت خروجی به صورت زیر میشود:

WGAN-GP : خروجی مدل آموزش دیده با **14** شکل

در این حالت مقدار FID نیز برابر با 26 میشود که نشان از بهتر شدن مدل GAN دارد.

سوال ۲ – Diffusion Models

الف) سوالات تئوری

سوال ۱ : مقایسه شبکه های مولد

مقایسه مدل های مولد مانند NF ، GAN ، VAE و Diffusion Models بر اساس معیارهای کیفیت ، تنوع و سرعت نمونه برداری ، چشم انداز روشنی در مورد توانایی ها و محدودیت های آنها ارائه می دهد:

کیفیت

- ✓ NF : مولد با کیفیت بالا با جزییات و دقیق را ارائه می دهد. مستقیماً از توزیع داده ها استفاده می کند ، که اغلب منجر به fidelity و انسجام بالا در نمونه های تولید شده می شود.

- ✓ GAN : برای تولید تصاویر بسیار با کیفیت و واقع بینانه ، به ویژه در کارهایی مانند تولید تصویر شناخته شده است. فرآیند آن اغلب منجر به خروجی های چشمگیر و با وضوح بالا می شود.

- ✓ vae: به طور کلی خروجی های با کیفیت پایین را در مقایسه با GAN ها تولید می کند. ، زیرا Vae ها اغلب با گرفتن دقیق جزئیات با فرکانس بالا مشکل دارند.

- ✓ Diffusion Model : آنها توانایی قابل توجهی در تولید نمونه های با کیفیت بالا و متنوع ، به ویژه در کارهایی مانند تصویر و تولید صوتی نشان داده اند. این مدل ها اخیراً از نظر کیفیت به نتایج پیشرفته رسیده اند.

تنوع

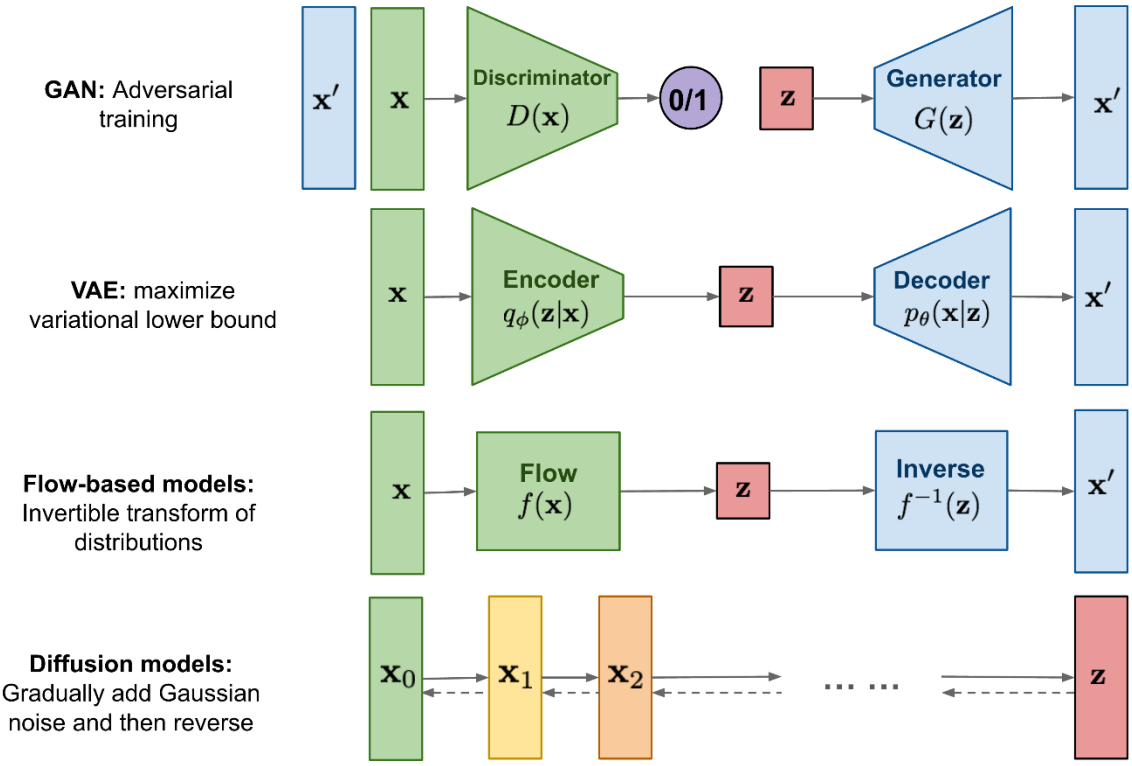
- ✓ NF : به دلیل آموزش دقیق آن ، قادر به تولید طیف گسترده ای از نمونه ها است. با این حال ، تنوع نمونه های تولید شده گاهی اوقات با بیان مدل محدود می شود.
- ✓ GAN: در حالی که GANS می تواند تصاویر با کیفیت بالا تولید کند ، اما بعضی اوقات از collapse (اینکه discriminator نتواند به خوبی fake و real را تشخیص دهد) رنج می برند ، جایی که آنها نتوانسته اند تنوع کامل داده های آموزش را نشان دهند.

- ✓ vae : تمایل به ارائه تنوع خوبی در نمونه های تولید شده دارد ، زیرا آنها صریحاً توزیع داده های اساسی را مدل می کنند. با این حال ، این تنوع اغلب به هزینه کیفیت تک نمونه می رسد.
- ✓ Diffusion Models : به دلیل تنوع عالی آنها در نسل شناخته شده است و طیف گسترده ای از تغییرات موجود در داده های آموزش را ضبط می کند.

سرعت نمونه برداری

- ✓ NF : نمونه برداری می تواند نسبتاً سریع و کارآمد باشد ، زیرا مستقیماً از توزیع آموخته شده بدون پالایش تکراری نمونه می گیرد.
- ✓ GAN : به طور معمول نمونه گیری سریع را ارائه می دهد ، زیرا شامل یک گذر مستقیم از طریق شبکه ژنراتور است. با این حال ، فرایند آموزش به خودی خود می تواند پررنگ و چالش برانگیز باشد.
- ✓ VAE: همچنین نمونه گیری سریع را ارائه می دهد زیرا شامل عبور از شبکه رمزگشایی ، مشابه GANS است.
- ✓ Diffusion Model : تمایل به سرعت نمونه برداری کندتر در مقایسه با سایر مدل ها دارند ، زیرا برای تولید یک نمونه به چندین مرحله تکراری نیاز دارند. این فرایند ، ضمن تولید نتایج با کیفیت بالا ، از نظر محاسباتی فشرده است.

به طور خلاصه ، هر مدل از نقاط قوت و ضعف خود برخوردار است: GAN ها و مدل های Diffusion در کیفیت ، VAE ها و NF تنوع خوبی را ارائه می دهند ، و GAN ها و VAE ها به طور کلی سرعت نمونه برداری سریعتر را ارائه می دهند. انتخاب مدل به شدت به الزامات خاص کار مورد نظر بستگی دارد.



شکل 15 : مقایسه معماری مدل های مولد

سوال ۲ : استفاده از $q(x_t|x_0)$ به صورت یک مرحله ای برای لایه های میانی

همانطور که در مقاله و در درس مطرح شده است در forward مقادیر به این صورت است که X1 نویزی که از X0 اصلی ساخته میشود به صورت:

$$X_1 = \sqrt{\alpha_1}\, X_0 + \sqrt{1-\alpha_1}\, \varepsilon_1$$

$$X_2 = \sqrt{\alpha_2}\, X_1 + \sqrt{1-\alpha_2}\, \varepsilon_2$$

و به همین ترتیب برای بقیه نیز میباشد...

که در آن ε یک گوسی با میانگین صفر و واریانس I بوده و مقادیر α بین صفر و یک میباشد.

حال اگر بیاییم هر یک از X ها فقط بر حسب X0 بنویسیم داریم :

$$X_1 = \sqrt{\alpha_1}\, X_0 + \sqrt{1-\alpha_1}\, \varepsilon_1$$

$$X_2 = \sqrt{\alpha_2\alpha_1}\, X_0 + \sqrt{\alpha_1(1-\alpha_2)}\, \varepsilon_1 + \sqrt{1-\alpha_2}\, \varepsilon_2$$

و به همین ترتیب برای بقیه نیز میباشد.

با توجه به مستقل بودن ε ها میتوان کل ترم دوم و سوم مثلا در X2 را به یک فرم نوشت (میانگین ها جمع میشود و میانگین گوسی جدید و جمع مجذور واریانس ها واریانس جدید میشود.)

$$\mathrm{X}_2 = \sqrt{\alpha_2\alpha_1}\, X_0 + \sqrt{1 - \alpha_1\alpha_2)}\, \varepsilon^*$$

$$\mathrm{X}_2 = \sqrt{\alpha_3\alpha_2\alpha_1}\, X_0 + \sqrt{1 - \alpha_3\alpha_1\alpha_2)}\, \varepsilon^*$$

اگر حاصل ضرب α_i ها را صورت $\bar{\alpha}$ نمایش دهیم داریم :

$$\mathrm{X}_t = \sqrt{\overline{\alpha_t}}\, X_0 + \sqrt{1 - \overline{\alpha_t})}\, \varepsilon^*$$

$$\rightarrow q(x_t|x_0) = N(x_t; \sqrt{\overline{\alpha_t}}\, X_0\, , (1 - \overline{\alpha_t})I)$$

سوال ۳ : در مسیر reverse چرا باید $q(x_{t-1}|x_t)$ گوسی باشد

چرا گوسی است ؟

مقاله DDPM در مورد چارچوبی برای مدل سازی مولد بحث می کند ، که شامل فرایندی برای اضافه کردن تدریجی نویز به داده ها و سپس یادگیری معکوس این فرآیند برای تولید نمونه های جدید است. در این زمینه ، روند معکوس بسیار مهم است.

فرض خاصی که به آن اشاره شده است که -که توزیع $q(x_{t-1}|x_t)$ گاوسی است-بخش مهمی از نحوه ساخت مدل است. در اینجا دلایل این فرض وجود دارد:

- ✓ سادگی و قابلیت تغییر ریاضی: توزیع های گاوسی به دلیل سادگی ریاضی و قابلیت تغییر ریاضی به طور گسترده در مدل سازی احتمالی مورد استفاده قرار می گیرد. آنها فقط با دو پارامتر (میانگین و واریانس) تعریف می شوند و کار آنها را در مقایسه با توزیع های پیچیده تر آسان تر می کنند.
- ✓ مدل سازی داده های continuous : توزیع گاوسی برای مدل سازی داده های continuous مؤثر است.
- ✓ فرآیند افزودن نویز : در چارچوب DDPM ، فرایند رو به جلو شامل اضافه کردن نویز گاوسی به داده ها در طی یک سری مراحل است. طبیعی است که فرآیند معکوس به طور مشابه مدل شود ، با فرض اینکه نویز در هر مرحله حذف شود نیز گاوسی است.

- ✓ موفقیت تجربی: فرض توزیع گاوسی در فرآیند معکوس موفقیت تجربی در تولید نمونه های با کیفیت بالا نشان داده است. این یک توجیه عملی قوی برای انتخاب است.
- ✓ مبانی نظری: فرض گاوسی با قضیه حد مرکزی تراز می شود ، که بیان می کند که در شرایط خاص ، متوسط تعداد زیادی از متغیرهای تصادفی ، صرف نظر از توزیع اساسی ، تقریباً به طور گوسی توزیع می شوند.
- ✓ محاسبات کارآمد: توزیع گاوسی از نظر محاسباتی برای کار با آنها کارآمد است ، به خصوص هنگام محاسبه احتمالات و نمونه گیری از توزیع ، که عملیات iterative در چارچوب DDPM هستند.

در زمینه $q(x_{t-1}|x_t)$، این فرض امکان تخمین ساده از پارامترهای توزیع گاوسی در هر مرحله از فرآیند معکوس را فراهم می کند. با فرض توزیع گاوسی ، مدل مشکل تخمین x_0(داده های اصلی و بدون نویز) را از x_t ساده می کند (داده های نویزی در مرحله t). این ساده سازی برای اثربخشی و کارآیی مدل در تولید نمونه های با کیفیت بالا مهم است.

چرا از x_0 هم استفاده میشود ؟

دلیل اصلی این کار این است که همانطور در کلاس نیز مطرح شد با اضافه کردن x_0 مساله دارای حل بسته میباشد.

اما برای دلایل دیگر میتوان به موارد زیر اشاره کرد:

- ✓ بازیابی داده های اصلی: در *DDPMS*، x_0 داده های اصلی و تمیز را قبل از افزودن هر نویز نشان می دهد. هدف کل فرآیند عقب *reverse* ، x_0 از داده های نویزی x_t است. بنابراین ، x_0 به عنوان یک هدف یا نقطه مرجع در این فرآیند *denoising* عمل می کند.
- ✓ هدایت فرآیند *denoising*: با درج x_0 در برآورد مسیر *reverse* ، مدل می تواند به طور مؤثر یاد بگیرد که چگونه روند افزودن نویز را معکوس کند. این یک مقایسه مستقیم بین داده های نویز در مرحله t و داده های تمیز اصلی را فراهم می کند ، و این مدل را در درک اینکه چه جنبه هایی از x_t نویز دارند و چه قسمت هایی با داده های اصلی مطابقت دارند ، هدایت می کند.
- ✓ ثبات و کیفیت آموزش: گنجاندن x_0 به تثبیت روند آموزش کمک می کند و کیفیت نمونه های تولید شده را بهبود می بخشد. با داشتن یک هدف واضح ، این مدل می تواند با دقت

بیشتری تفاوت های ظریف توزیع داده ها را بیاموزد و در تولید نمونه های وفاداری بالا بهتر شود.

- ✓ یادگیری توزیع مشروط : استفاده از x_0 در فرآیند *reverse* اساساً به این معنی است که مدل توزیع های مشروط را یاد می گیرد - نحوه توزیع x_{t-1} (داده ها در جدول زمانی قبلی) بستگی دارد در هر دو داده نویزی فعلی x_t و داده های اصلی x_0.این به ضبط وابستگی ها و ساختار موجود در داده های اصلی به طور مؤثر کمک می کند.
- ✓ تنوع نمونه و رئالیسم بهبود یافته : این مدل می تواند نمونه های متنوع تر و واقع بینانه تری ایجاد کند. این تضمین می کند که نمونه های تولید شده ویژگی های اصلی مجموعه داده اصلی را حفظ می کنند در حالی که هنوز امکان تغییرپذیری و خلاقیت در فرآیند تولید را دارند.

سوال ۴ : مفهوم هر یک از ترم های تابع هزینه

معادله (2.8) مربوط به هدف آموزش در یک مدل (DDPM) است ، که شبیه به (ELBO) در (VAES) است. بیایید اجزای معادله را تجزیه کنیم:

$$L_{VLB} = L_T + \sum_{t=1}^{T-1} L_t + L_0$$

جایی که

$$L_t = D_{KL}(q(X_t|X_{t-1}, X_0)||p_\theta(X_{t-1}|X_t))$$
$$L_0 = -logp_\theta(X_0|X_1)$$

در مورد هر یک از ترم ها داریم :

- ✓ L_T: این ترم با مرحله زمانی نهایی مطابقت دارد و احتمال ورود به سیستم داده ها را در زیر مدل prior شامل می کند. در بسیاری از موارد ، این ممکن است یک توزیع ساده مانند توزیع عادی استاندارد باشد.
- ✓ L_t: این ترم ها برای هر مرحله از زمان از t=1 تا t= t-1 است. هر L_T یک اصطلاح واگرایی Kullback-Leibler است که تفاوت بین دو توزیع را اندازه گیری می کند:
  - $q(X_t|X_{t-1}, X_0)$: توزیع واقعی
  - $p_\theta(X_{t-1}|X_t)$: توزیع تخمین زده شده مدل

- ✓ هدف در طول آموزش ، به حداقل رساندن این واگرایی KL است ، به این معنی که توزیع تخمین زده شده مدل تا حد امکان به توزیع واقعی نزدیک است.
- ✓ L_0 : این اصطلاح برای مرحله اولیه است و با توجه به اولین داده های نویزی x_1 احتمال ورود به سیستم داده های اصلی x_0 را شامل می شود. این ترم بسیار مهم است زیرا مستقیماً شامل توانایی مدل در بازسازی داده های اصلی از داده های نویزی است که هدف نهایی مدل تولیدی است.

به طور خلاصه ، این معادله چندین هدف را ترکیب می کند: کیفیت بازسازی در مرحله زمانی پایانی ، fidelity فرآیند معکوس در مراحل زمانی متوسط و توانایی کلی در بازسازی داده های اصلی از مشاهدات نویزی . با بهینه سازی این هدف ، مدل می آموزد که داده هایی را تولید کند که از نزدیک به داده های آموزش شبیه باشد ، به طور موثری توزیع داده های اساسی را ضبط می کند.

سوال ۵ : کدام ترم ها در ddpm در نظر گرفته نشده است

در مقاله ddpm ترم اول یا همان L_T ایگنور شده است که در ادامه دلیل آن امده است.

- ✓ L_T: این ترم به طور معمول شامل احتمال ورود به سیستم داده های تحت مدل *prior* است. در مورد *DDPMS* ، موارد قبلی اغلب در زمان نهایی T یک توزیع نرمال استاندارد فرض می شود ، و این ترم را می توان به راحتی محاسبه کرد یا حتی در صورت ثابت بودن آن نادیده گرفت زیرا این امر بر روی شیب ها تأثیر نمی گذارد.
- ✓ L_t: این ترم شامل واگرایی *KL* بین توزیع فرآیند معکوس واقعی و توزیع تقریبی فرآیند معکوس مدل است. در عمل ، این ترم اغلب ساده می شوند. مقاله نشان می دهد که در شرایط خاص ، فرآیند رو به جلو از نزدیک روند معکوس را تقریب می دهد به عنوان توزیع های گاوسی با میانگین و واریانس خاص. این ساده سازی امکان بیان بسته برای واگرایی *KL* را فراهم می کند ، که گاهی اوقات می توان با ثابت بودن واریانس فرآیند معکوس نادیده گرفت.
- ✓ L_0 : اصطلاح به طور معمول مهمترین برای یادگیری عملکرد *denoising* است و نادیده گرفته نمی شود. این اندازه گیری می کند که چگونه مدل می تواند داده های اصلی را از داده های نویزی بازسازی کند. این اصطلاح برای آموزش مدل برای تولید نمونه ها ضروری است و در عملکرد هدف نگهداری می شود.

به طور کلی دلیل اصلی نادیده گرفتن یا ساده شدن برخی ترم ها ، راندمان محاسباتی است. محاسبه کامل *VLB* می تواند برای مدل های پیچیده و مجموعه داده های بزرگ بسیار سنگین یا بسیار پرهزینه

باشد. با تمرکز بر مهمترین ترم هایی که در یادگیری عملکرد *denoising* نقش دارند ، آموزش در حالی که هنوز به عملکرد مورد نظر می رسد امکان پذیر تر می شود.

سوال ۶ : تاثیر جایگزینی توزیع پیچیده با توزیع گوسی

- ✓ افزایش پیچیدگی محاسباتی: توزیع گوسی راحت است زیرا در بسیاری از محاسبات از جمله واگرایی Kullback-Leibler (KL) امکان راه حل های بسته را فراهم می کند. توزیع پیچیده به احتمال زیاد به روشهای عددی برای محاسبه واگرایی KL ، که از نظر محاسباتی گران تر هستند ، نیاز دارد.
- ✓ دشواری در برآورد پارامترها: توزیع گاوسی با استفاده از میانگین و واریانس آنها تعریف می شود ، که برای تخمین نسبتاً ساده هستند. توزیع پیچیده تر ممکن است تعداد بیشتری از پارامترها یا پارامترها داشته باشد که تخمین آن دشوارتر است و باعث افزایش بار محاسباتی می شود.
- ✓ چالش های تخمین گرادیان: بهینه سازی در DDPM به روشهای مبتنی بر گرادیان متکی است. اگر مسیر عقب با توزیع پیچیده مدل شود ، گرادیان loss با توجه به پارامترهای مدل ممکن است فرم تحلیلی نداشته باشد. این می تواند به استفاده از تکنیک های تقریبی تخمین گرادیان ، مانند روشهای مونت کارلو ، که از نظر محاسباتی بیشتر مورد نیاز هستند ، ضروری باشد.

- ✓ افزایش نیازهای حافظه: توزیع های پیچیده تر می تواند نیاز به ذخیره اطلاعات اضافی در هر دو forward and backward passes باشد که باعث افزایش مصرف حافظه می شود. این می تواند یک اشکال مهم باشد ، به خصوص هنگام کار با مجموعه داده های بزرگ یا داده های با ابعاد بالا.
- ✓ همگرایی آهسته تر: اگر هزینه به دلیل غیر گاوسی بودن پیچیده تر باشد ، ممکن است آهسته تر همگرا شود. این می تواند بدان معنی باشد که epoch های آموزش بیشتری برای دستیابی به نتایج قابل مقایسه ، افزایش بیشتر هزینه های محاسباتی لازم است.
- ✓ overfitting: توزیع های پیچیده ظرفیت بیشتری برای متناسب بودن داده ها دارند ، که می تواند منجر به overfit شود ، به خصوص اگر مجموعه داده به اندازه کافی بزرگ نباشد تا پیچیدگی مدل را توجیه کند. این ممکن است نیاز به تکنیک های تنظیم مجدد اضافی داشته باشد ، که می تواند به هزینه محاسباتی بیفزاید.

سوال ۷ : روند رسیدن به تابع هزینه نهایی در ddpm

همانطور که در فرایند زیر مشاهده میشود کمینه کردن KL میتواند در حالتی که هر 2 گوسی باشند معادل کمینه کردن خطای مربعات میانگین باشد.

$$= \arg\min_{\boldsymbol{\theta}} D_{\mathrm{KL}}\left(\mathcal{N}\left(\boldsymbol{x}_{t-1}; \boldsymbol{\mu}_q . \boldsymbol{\Sigma}_q(t)\right) \| \mathcal{N}\left(\boldsymbol{x}_{t-1}; \boldsymbol{\mu}_{\boldsymbol{\theta}} . \boldsymbol{\Sigma}_q(t)\right)\right)$$

$$= \arg\min_{\boldsymbol{\theta}} \frac{1}{2}\left[\log \frac{\left|\boldsymbol{\Sigma}_q(t)\right|}{\left|\boldsymbol{\Sigma}_q(t)\right|} - d + \operatorname{tr}\left(\boldsymbol{\Sigma}_q(t)^{-1}\boldsymbol{\Sigma}_q(t)\right) + \left(\boldsymbol{\mu}_{\boldsymbol{\theta}} - \boldsymbol{\mu}_q\right)^T \boldsymbol{\Sigma}_q(t)^{-1}\left(\boldsymbol{\mu}_{\boldsymbol{\theta}} - \boldsymbol{\mu}_q\right)\right]$$

$$= \arg\min_{\boldsymbol{\theta}} \frac{1}{2}\left[\left(\boldsymbol{\mu}_{\boldsymbol{\theta}} - \boldsymbol{\mu}_q\right)^T \boldsymbol{\Sigma}_q(t)^{-1}\left(\boldsymbol{\mu}_{\boldsymbol{\theta}} - \boldsymbol{\mu}_q\right)\right]$$

$$= \arg\min_{\boldsymbol{\theta}} \frac{1}{2}\left[\left(\boldsymbol{\mu}_{\boldsymbol{\theta}} - \boldsymbol{\mu}_q\right)^T \left(\sigma_q^2(t)\mathbf{I}\right)^{-1}\left(\boldsymbol{\mu}_{\boldsymbol{\theta}} - \boldsymbol{\mu}_q\right)\right]$$

$$= \arg\min_{\boldsymbol{\theta}} \frac{1}{2\sigma_q^2(t)}\left[\left\|\boldsymbol{\mu}_{\boldsymbol{\theta}} - \boldsymbol{\mu}_q\right\|_2^2\right]$$

بنابراین طبق روابط فوق باید میانگین ها را به هم نزدیک کرد.

$$L_{t-1} = \mathbb{E}_q\left[\frac{1}{2\sigma_t^2}\left\|\tilde{\boldsymbol{\mu}}_t(\mathbf{x}_t, \mathbf{x}_0) - \boldsymbol{\mu}_\theta(\mathbf{x}_t, t)\right\|^2\right] + C$$

شکل 16 : رابطه شماره 8 مقاله **DDPM**

که در ادامه رابطه ا استفاده از رابطه زیر فرایند فوروارد نوشته میشود

$$\mathrm{X}_t = \sqrt{\overline{\alpha_t}}\, X_0 + \sqrt{1 - \overline{\alpha_t)}}\, \varepsilon^*$$

$$L_{t-1} - C = \mathbb{E}_{\mathbf{x}_0,\boldsymbol{\epsilon}}\left[\frac{1}{2\sigma_t^2}\left\|\tilde{\boldsymbol{\mu}}_t\left(\mathbf{x}_t(\mathbf{x}_0,\boldsymbol{\epsilon}), \frac{1}{\sqrt{\bar{\alpha}_t}}(\mathbf{x}_t(\mathbf{x}_0,\boldsymbol{\epsilon}) - \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon})\right) - \boldsymbol{\mu}_\theta(\mathbf{x}_t(\mathbf{x}_0,\boldsymbol{\epsilon}), t)\right\|^2\right]$$

$$= \mathbb{E}_{\mathbf{x}_0,\boldsymbol{\epsilon}}\left[\frac{1}{2\sigma_t^2}\left\|\frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t(\mathbf{x}_0,\boldsymbol{\epsilon}) - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}\right) - \boldsymbol{\mu}_\theta(\mathbf{x}_t(\mathbf{x}_0,\boldsymbol{\epsilon}), t)\right\|^2\right]$$

حال با توجه به رابطه فوق که مشخص میشود μ_θ باید چه چیزی را به شرط x_t پیشبینی کند میتوان از فرمول زیر در loss استفاده کرد.

$$\boldsymbol{\mu}_\theta(\mathbf{x}_t, t) = \tilde{\boldsymbol{\mu}}_t\left(\mathbf{x}_t, \frac{1}{\sqrt{\bar{\alpha}_t}}(\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t))\right) = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right)$$

بدین ترتیب لاس به صورت زیر درمیاید :

$$\mathbb{E}_{\mathbf{x}_0,\boldsymbol{\epsilon}}\left[\frac{\beta_t^2}{2\sigma_t^2\alpha_t(1-\bar{\alpha}_t)}\left\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t)\right\|^2\right]$$

توضیحات بیشتر در سایت زیر میباشد و منابع این بخش نیز بر اساس مقاله و سایت زیر میباشد.

https://lilianweng.github.io/posts/2021-07-11-diffusion-models

سوال ۸ : پارامتر زمان

در مقاله DDPM ، از معماری U-NET برای برآورد نویز استفاده می شود و برای همه زمان ها یکسان است. برای ترکیب پارامتر زمان گسسته در شبکه ، DDPM از یک تکنیک خاص استفاده می کند که شامل sin Positional Encoding است. این تعبیه به هر لایه u-net اضافه می شود در اینجا جزئیات وجود دارد:

- ✓ UNET با self Attention: DDPM از معماری U-NET با Self Attention استفاده می کند. این معماری با دیگران مانند NCSN متفاوت است.
- ✓ Conditioning on Time Parameter : در DDPM ، تمام لایه های U-NET با اضافه کردن Sin Positional Encoding در پارامتر زمان Condition می شوند. این برخلاف روشهای دیگر است که condition فقط در لایه های نرمال سازی یا در خروجی رخ می دهد. Positional Encoding تکنیکی از مدل های ترنسفورمری است است که به طور معمول برای رمزگذاری اطلاعات پی در پی در کارهای پردازش زبان طبیعی استفاده می شود. این راهی برای تزریق اطلاعات در مورد موقعیت (یا در این حالت ، مرحله زمانی) به شبکه است.

- ✓ اجرای در شبکه: positional Encoding به هر بلوک باقیمانده از شبکه U اضافه می شود. این روش تضمین می کند که اطلاعات زمان به طور یکنواخت در سراسر شبکه ادغام شده است و بر پردازش در هر مرحله از معماری U-Net تأثیر می گذارد

- ✓ هدف از time conditioning : دلیل conditioning در پارامتر زمان ، تطبیق رفتار شبکه بر اساس مرحله زمانی خاص در فرآیند diffusion است. از آنجا که U-NET برای برآورد نویز و معکوس کردن فرآیند diffusion استفاده می شود ، برای انجام تخمین های دقیق باید از مرحله خاص فرآیند آگاه باشد. پارامتر زمان نقش مهمی در این فرآیند بازی می کند و شبکه را در مورد چگونگی استفاده مناسب از تصویر در هر مرحله خاص هدایت می کند.

سوال ۹ : مقاله latent diffusion و cross attention

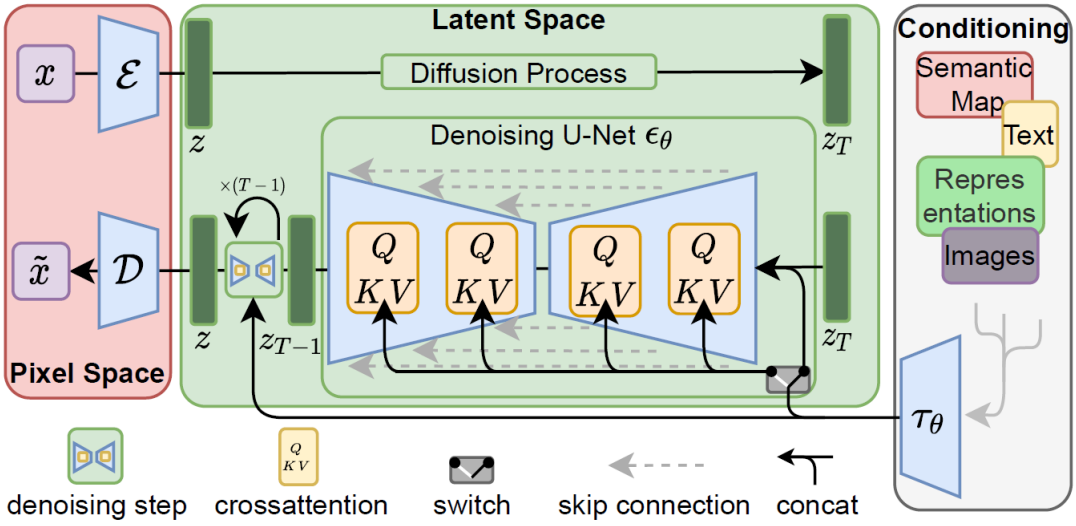
مدل Latent Diffusion (LDM) نشان دهنده یک تغییر دگرگون کننده در مدل سازی مولد، به ویژه در ایجاد تصاویر است. این مدل که توسط Rombach، Blattmann و همکاران مفهوم سازی شده است، برای اجرای فرآیندهای انتشار در یک فضای پنهان فشرده به جای فضای پیکسل سنتی طراحی شده است. این نوآوری به طور قابل ملاحظه ای منابع محاسباتی مورد نیاز برای آموزش را کاهش می دهد و فرآیند استنتاج را تسریع می کند. این کارایی است که به LDM اجازه می دهد تا تصاویر با کیفیت بالا با کسری از هزینه محاسباتی که معمولاً با چنین وظایفی مرتبط است تولید کند.

در چارچوب LDM، جزئیات ادراکی یک تصویر علیرغم فشرده‌سازی قابل توجه، تا حد زیادی حفظ می‌شوند. این امر با تقسیم فرآیند مدل‌سازی مولد به دو مرحله مجزا انجام می‌شود: فشرده‌سازی ادراکی و فشرده‌سازی معنایی. در ابتدا، یک رمزگذار خودکار، افزونگی در سطح پیکسل را کاهش می‌دهد و به طور خلاصه ماهیت یک تصویر را رمزگذاری می‌کند. سپس مدل از یک فرآیند انتشار در فضای پنهان کدگذاری شده برای دستکاری و فرموله کردن محتوای معنایی استفاده می کند و تصاویری با جزئیات غنی و انسجام مفهومی ایجاد می کند.

با پرداختن به شدت محاسباتی مدل‌های انتشار سنتی، LDM نمایش فشرده‌تری از داده‌ها را معرفی می‌کند. با استفاده از رمزگذار خودکار برای یادگیری فضای پنهان با ابعاد کاهش یافته در طول آموزش، این مدل پایه ای را برای فرآیند انتشار آماده می کند که محاسبات کمتری را ضروری می کند. در نتیجه، LDM نه تنها توان محاسباتی را حفظ می کند، بلکه یک مسیر ساده برای ایجاد تصاویر پیچیده را نیز فراهم می کند.

معماری LDM با ادغام مکانیسم‌های توجه متقابل (cross – Attention) ، که به مدل انعطاف‌پذیری قابل‌توجهی در تولید محتوا از ورودی‌های متنوع می‌دهد، بیشتر تقویت می‌شود. این توجه متقابل (cross – Attention) به مدل اجازه می‌دهد تا جنبه‌های مرتبط تصویر را تشخیص داده و ترکیب کند و کیفیت خروجی را به طور قابل توجهی بهبود بخشد. چنین پیشرفت‌های معماری همچنین راه را برای LDM برای

شرکت در آموزش چند وجهی هموار می‌کند، که در تبدیل توصیفات و طرح‌بندی‌های متنی به طور مستقیم به تصاویر واضح بسیار ارزشمند است. از طریق این قابلیت‌ها، LDM استاندارد جدیدی را در زمینه مدل‌سازی تولیدی تعیین می‌کند و کاربرد خود را فراتر از سنتز ساده تصویر به طیف گسترده‌ای از برنامه‌های کاربردی توسعه می‌دهد.



شکل 17 : معماری **LDM**

سوال ۱۰ : مدل DDIM

: DDPM

مفهوم: DDPM یک مدل تولیدی است که با اضافه کردن تدریجی نویز به یک تصویر یا نمونه داده تا زمانی که سیگنال کاملاً نویزی باشد کار می کند. سپس ، یاد می گیرد که این فرآیند را معکوس کند و داده های اصلی را از سیگنال نویزی بازسازی کند. این شبیه به شروع با یک تصویر واضح است و سپس به آرامی اضافه کردن مه تا زمانی که مبهم شود ، و سپس یادگیری حذف مه برای آشکار کردن تصویر.

این فرایند از نظر محاسباتی فشرده و وقت گیر است اما نمونه های با کیفیت بالا تولید می کند.

: DDIM

به روزرسانی: DDIM از نظر کارآیی فرآیند نمونه برداری پیشرفت نسبت به DDPM است. این نحوه برداشتن این مراحل نادیده گرفتن را اصلاح می کند.

کارآیی: DDIM تعداد مراحل مورد نیاز برای تولید نمونه را کاهش می دهد. از یک فرآیند غیر مارکووی استفاده می کند و باعث می شود تولید نمونه سریعتر و کارآمدتر در مقایسه با رویکرد سنتی DDPM باشد.

کیفیت در مقابل سرعت: در حالی که DDIM یک مزیت سرعت را ارائه می دهد ، ممکن است گاهی اوقات با هزینه کاهش جزئی در کیفیت نمونه در مقایسه با DDPM این کار را انجام دهد. با این حال ، این trade of اغلب قابل قبول است ، به ویژه در کارهایی که سرعت تولید بسیار مهم است.

با توجه به مقاله DDIM داریم :

عملیات reverse به صورت زیر انجام میشود :

$$x_{t-1} = \sqrt{\bar{\alpha}_{t-1}} \left(\frac{x_t - \sqrt{1-\bar{\alpha}_t}\epsilon_\theta^{(t)}(x_t)}{\sqrt{\bar{\alpha}_t}} \right) + \sqrt{1-\bar{\alpha}_{t-1}} \cdot \epsilon_\theta^{(t)}$$

DDIM Denoising Formula

شکل 18 : عملیات **revere** در **DDIM**

که با توجه به مقاله و مناسباتی که با DDPM دارد مقدار واریانس به صورت زیر میشود:

$$\sigma_t = \eta \sqrt{\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}} \sqrt{1 - \frac{\bar{\alpha}_t}{\bar{\alpha}_{t-1}}} = \eta \sqrt{\tilde{\beta}_t}$$

DDIM variance

شکل 19 : واریانس در **DDIM**

بدین تریب با توجه به فرمول بندی DDPM داریم :

Diffusion Model یک DDIM است که $\eta = 0$ و DDPM اصلی در هنگام $\eta = 1$ وجود ندارد. هر η بین 0 تا 1 یک درون یابی بین DDIM و DDPM است.

سوال ۱۱ : مدل diffusion-based در مساله Semantic Segmentation

ابتدا روش هایی که میتوان از آن برای بهتر کردن روش های segmentation استفاده کرد مطرح میشود:

- ✓ افزایش داده ها: مدل های diffusion می توانند تغییرات متنوع و واقع بینانه تصاویر موجود را ایجاد کنند. این تصاویر تولید شده به همراه نقشه های تقسیم بندی مربوطه می توانند برای تقویت مجموعه داده های آموزشی ، بهبود استحکام و تعمیم پذیری مدل تقسیم بندی استفاده شوند.
- ✓ استخراج و پالایش ویژگی: در تقسیم بندی ، از مدل های diffusion می توان برای تصحیح ویژگی های استخراج شده توسط یک شبکه عصبی Convolutional (CNN) استفاده کرد. آنها می توانند در مدل سازی الگوهای و بافت های پیچیده کمک کنند ، که ممکن است برای تقسیم بندی دقیق مفید باشد ، به خصوص در سناریوهای چالش برانگیز مانند تصویربرداری پزشکی یا صحنه های با جزئیات بالا.
- ✓ پس از پردازش : پس از تقسیم بندی اولیه با یک روش دیگر انجام می شود ، می توان از یک مدل انتشار برای پردازش تصاویر تقسیم شده استفاده کرد. این مرحله می تواند به صاف کردن لبه ها ، پر کردن شکاف ها یا تصحیح پیکسل های نادرست طبقه بندی شده کمک کند ، بنابراین باعث بهبود دقت و کیفیت بصری تقسیم بندی می شود. مانند مقاله هایی که یک uncertainty map به دست میاورند و به کمک diffusion model ها آن را تصحیح و بهتر میکنند.

- ✓ تقسیم بندی با روش مولد : در یک رویکرد پیشرفته تر ، یک مدل diffusion می تواند آموزش داده شود تا مستقیماً تصاویر تقسیم شده را از تصاویر ورودی تولید کند. این شامل آموزش مدل بر روی جفت تصاویر خام و تصاویر تقسیم شده مربوطه آنها ، آموزش آن برای درک و تکرار روند تقسیم بندی است.
- ✓ تقسیم بندی تعاملی: مدل های diffusion می توانند برای کارهای تقسیم بندی تعاملی سازگار شوند ، جایی که کاربر برخی از ورودی ها را بر روی تصویر فراهم می کند و مدل نقشه تقسیم بندی مربوطه را تولید می کند. این امر می تواند به ویژه در برنامه هایی که در آن تقسیم دقیق هدایت کاربر مورد نیاز است مفید باشد.

- ✓ انتقال سبک معنایی : از یک مدل انتشار برای ترکیب سبک معنایی یک تصویر با محتوای دیگر استفاده کنید. به عنوان مثال ، نقشه تقسیم بندی یک منظره شهری را بگیرید و آن را در یک صحنه طبیعی اعمال کنید و یک تصویر ترکیبی ایجاد کنید. سپس ، از تقسیم بندی معنایی برای تفسیر این تصویر جدید و ترکیبی استفاده کنید. این رویکرد می تواند به بینش جدیدی در مورد چگونگی درک و تقسیم محیط های مختلف توسط مدل ها منجر شود.

به عنوان مثال برای بخش به دست آوردن فیچر میتوان به مقاله زیر اشاره کرد.[1]

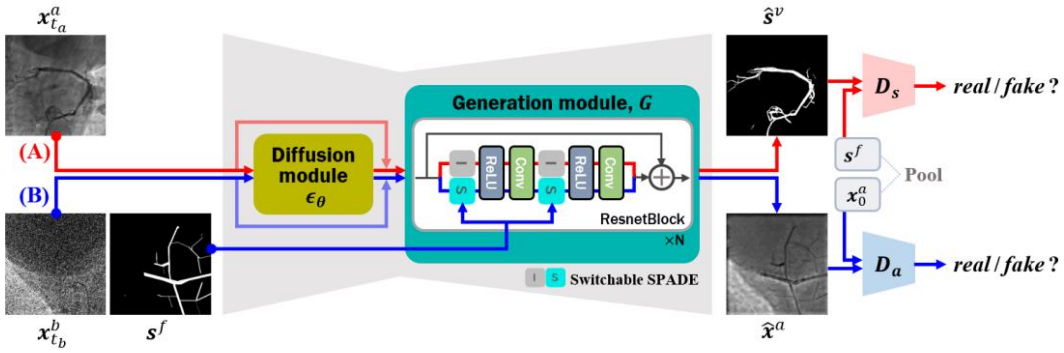


Figure 9: An overview of DARL [62]. Path (A) involves feeding a real noisy angiography image through the model to generate a segmentation map. Path (B) incorporates passing a noisy background image alongside a vessel-like fractal mask through the model to synthesize a synthetic angiography image.

شکل 20 : مدل **DARL**

ب) سوالات پیاده سازی

لود کردن دیتاست

با توجه به صورت مساله لازم است تا ابتدا دیتاست CIFAR 10 لود شود که این کار به کمک کد زیر انجام شده است.

```
#title cifar10 - 32px images in 10 classes

# Download and load the dataset
cifar10 = load_dataset('cifar10')

# View some examples:
```

[1] Kazerouni, Amirhossein, et al. "Diffusion models in medical imaging: A comprehensive survey." *Medical Image Analysis* (2023): 102846.

```
image = Image.new('RGB', size=(32*5, 32*2))
for i in range(10):
  im = cifar10['train'][i]['img']
  image.paste(im, ( (i%5)*32, (i//5)*32
  ))
image.resize((32*5*4, 32*2*4), Image.NEAREST)
```

شکل 21 : نمونه تصاویر دیتاست **CIFA10**

سوال ۱۲ : رابطه گام به گام مسیر رو به جلو

در اینجا هدف پیاده سازی زیر میباشد.

$$q(x_t|x_{t-1}) = \mathcal{N}\left(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t I\right)$$

$$q(x_{1:T}|x_0) = \prod_{t=1}^{T} q(x_t|x_{t-1})$$

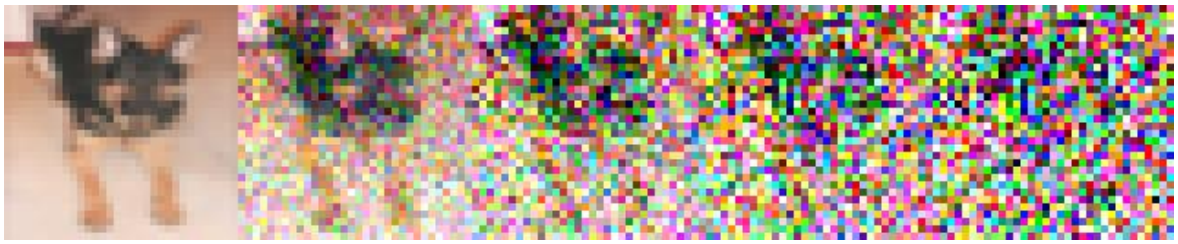
با توجه به فرمول بندی فوق میتوان تابع را به صورت زیر نوشت.

```
def q_xt_xtminus1(xtm1, t):
    """
    Generate noise using the given equation.

    This function calculates the conditional distribution q(x_t | x_{t-1}), which is a
normal distribution
    with mean √(1-βt)*xtm1 and variance βt*I.

    Args:
        xtm1: Tensor representing x_{t-1}.
        t: Time step or index for β.

    Returns:
        Tensor representing the noise added value of x_t.
```

```
    """
    # Calculate mean and variance
    mean = (1. - gather(beta, t)).sqrt() * xtm1
    variance = gather(beta, t)

    # Generate noise
    eps = torch.randn_like(xtm1)

    # Return the noise-added value
    return mean + (variance.sqrt() * eps)
```

بدین ترتیب به کمک زیر میتوان برای یک تصویر مسیر رفت که تصویر نویزی میشود را نمایش داد.

```
# Show im at different stages
ims = []
start_im = cifar10['train'][10]['img']
x = img_to_tensor(start_im).squeeze()
for t in range(n_steps):

  # Store images every 20 steps to show progression
  if t%20 == 0:
    ims.append(tensor_to_image(x))

  # Calculate Xt given Xt-1 (i.e. x from the previous iteration)
  t = torch.tensor(t, dtype=torch.long) # t as a tensor
  x = q_xt_xtminus1(x, t) # Modify x using our function above

# Display the images
image = Image.new('RGB', size=(32*5, 32))
for i, im in enumerate(ims):
  image.paste(im, ((i%5)*32, 0))
image.resize((32*4*5, 32*4), Image.NEAREST)
```



شکل 22 : سیر نویزی شدن تصویر در **diffusion model**

سوال ۱۳ : رابطه گام به گام مسیر رو به جلو در یک مرحله

هدف پیاده سازی فرمول زیر میباشد.

$$q(x_t|x_0) = \mathcal{N}\left(x_t; \sqrt{\overline{\alpha_t}}x_0, (1-\overline{\alpha_t})I\right)$$

$$\text{where } \overline{\alpha_t} = \prod_{i=1}^{T} \alpha_i$$

برای اینکار از کد زیر استفاده میشود.

```
n_steps = 100
beta = torch.linspace(0.0001, 0.04, n_steps)
alpha = 1. - beta
alpha_bar = torch.cumprod(alpha, dim=0)

def q_xt_x0(x0, t):
    """
    Calculate the conditional distribution q(x_t | x_0) as a normal distribution.

    This function computes the distribution with mean √(α¯_t) * x0 and variance (1 - α¯_t)
* I,
    where α¯_t = Π(α_i) from i=1 to T.

    Args:
        x0: Tensor representing x_0.
        t: Time step or index for α¯.

    Returns:
        Tensor representing the value of x_t with added noise.
    """
    # Calculate mean and variance
    mean = gather(alpha_bar, t).sqrt() * x0
    variance = 1 - gather(alpha_bar, t)

    # Generate noise
    eps = torch.randn_like(x0)

    # Return the noise-added value
    return mean + (variance.sqrt() * eps)

# Show im at different stages
ims = []
start_im = cifar10['train'][10]['img']
x0 = img_to_tensor(start_im).squeeze()
for t in [0, 20, 40, 60, 80]:
  x = q_xt_x0(x0, torch.tensor(t, dtype=torch.long)) # TODO move type to gather
```

```
  ims.append(tensor_to_image(x))

image = Image.new('RGB', size=(32*5, 32))
for i, im in enumerate(ims):
  image.paste(im, ((i%5)*32, 0))
image.resize((32*4*5, 32*4), Image.NEAREST)
```

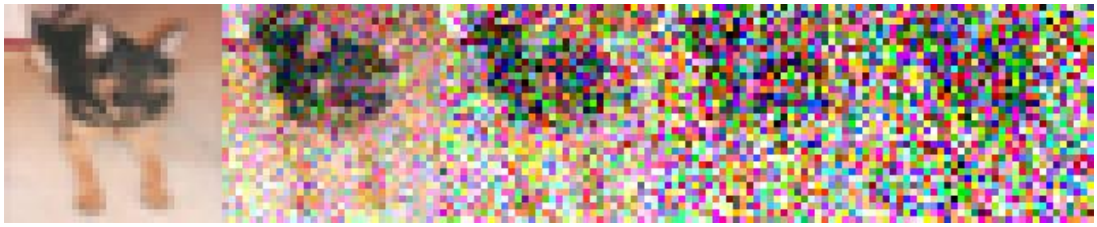
در کد فوق،

Initial Setup: کد برخی از پارامترها را برای فرآیندی تنظیم می کند که شامل اضافه کردن تدریجی نویز به تصاویر است. مقادیر خاصی (alpha و alpha_bar) را محاسبه می کند که برای کنترل میزان نویز اضافه شده در هر مرحله استفاده می شود.

عملکرد اضافه کردن نویز: یک تابع تعریف شده است (q_xt_x0) که یک تصویر و یک مرحله زمانی را می گیرد. این عملکرد بر اساس مرحله زمانی، مقدار خاصی نویز به تصویر اضافه می کند.

پردازش یک تصویر: کد یک تصویر از یک مجموعه داده (CIFAR-10) می گیرد و آن را در مراحل مختلف پردازش می کند (مانند مرحله 0، 20، 40، 60، 80). در هر مرحله، عملکرد اضافه کردن نویز را به تصویر اعمال می کند.

نمایش نتایج: پس از پردازش تصویر در مراحل مختلف، کد تمام این نسخه‌های تصویر را در یک تصویر واحد قرار می‌دهد. این تصویر نهایی نشان می دهد که چگونه تصویر اصلی با افزایش مقادیر نویز اضافه شده در مراحل مختلف به نظر می رسد.

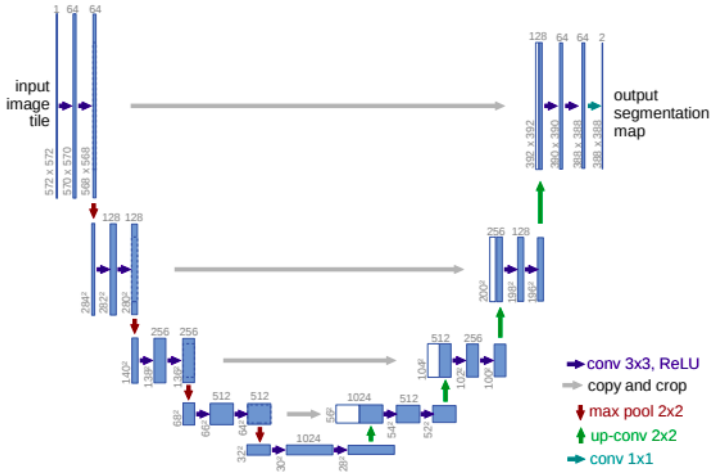


شکل 23 : فرایند نویزی کردن تصویر در **diffusion**

در دومی پرواضح است که فرایند نویزی زود و ازهمان ابتدا وارد میشود و عملکرد بدتر است.

سوال ۱۴ : آموزش و ارزیابی مدل diffusion

در بخش اول ابتدا UNET که مدل آن داده شده است تشریح میشود.



شکل 24 : ساختار **UNET**

در کد UNET داده شده،

- ✓ پردازش اولیه: مدل با تبدیل تصویر ورودی به مجموعه ای از ویژگی ها با استفاده از لایه های کانولوشن شروع می شود. این مانند آماده کردن تصویر برای پردازش بیشتر است.
- ✓ Going Deeper with Down Blocks: سپس مدل این ویژگی ها را از طریق یک سری "Down Blocks" عبور می دهد. در هر بلوک، ویژگی ها پردازش می شوند و وضوح کاهش می یابد (تصویر کوچکتر و فشرده تر می شود). در هر سطح، مدل جزئیات مختلفی از تصویر را ثبت می کند.
- ✓ بلوک میانی: در پایین‌ترین وضوح (فشرده‌ترین نسخه تصویر)، مدل از یک «بلاک میانی» استفاده می‌کند که ویژگی‌ها را بیشتر پردازش می‌کند. این بلوک بسیار مهم است زیرا عمیق‌ترین بخش شبکه را نشان می‌دهد، جایی که مدل سعی می‌کند انتزاعی‌ترین نمایش‌های تصویر را درک کند.
- ✓ Building Back Up with Up Blocks: بعد از بلوک میانی، مدل از "Up Blocks" برای افزایش وضوح استفاده می کند و به تدریج تصویر را به اندازه اصلی خود باز می گرداند. در طول این مرحله، مدل اطلاعات دقیقی را که در بلوک های پایین آموخته و اطلاعات انتزاعی از بلوک میانی را برای بازسازی تصویر ترکیب می کند.
- ✓ بلوک های Attention : هم در بلوک های پایین و هم در بلوک های بالا، این مدل گاهی اوقات از «بلوک های Attention» استفاده می کند. اینها به مدل کمک می کند تا روی قسمت های مهم تصویر تمرکز کند، مشابه اینکه چگونه انسان ها به بخش های خاصی از صحنه توجه بیشتری می کنند.

- ✓ مراحل نهایی: در نهایت، مدل مقداری نرمال سازی و یک لایه کانولوشنی دیگر را برای تولید تصویر نهایی اعمال می کند.

برای تست مدل از کد زیر استفاده شده است تا ابعاد خروجی آن را دربیاورد که نشان از عملکرد درست مدل دارد.

```
# Let's see it in action on dummy data:

# A dummy batch of 10 3-channel 32px images
x = torch.randn(10, 3, 32, 32)

# 't' - what timestep are we on
t = torch.tensor([50.], dtype=torch.long)

# Define the unet model
unet = UNet()

# The foreward pass (takes both x and t)
model_output = unet(x, t)

# The output shape matches the input.
model_output.shape
```

```
torch.Size([10, 3, 32, 32])
```

فرایند آموزش در ادامه بدین صورت است که ابتدا فقط مدل با ۱ ایپاک ران میشود که فرایند آن به صورت زیر است.(در این کد که در ادامه گفته شده چرا غلط میباشد مقادیر loss برای ترین به ازای هر batch ذخیره میشود و برای val پس از هر batch روی ترین یک دور کل Val مقدار loss محاسبه میشود)

```
# Initialize the UNet model
unet = UNet(n_channels=32).cuda()

# Set up training parameters
n_steps = 100
beta = torch.linspace(0.0001, 0.04, n_steps).cuda()
alpha = 1. - beta
alpha_bar = torch.cumprod(alpha, dim=0)
number_of_epochs = 1

def q_xt_x0(x0, t):
    """
```

```
    Apply the noise process to the input images.

    Args:
        x0: Tensor representing the original batch of images.
        t: Tensor of time steps for each image in the batch.

    Returns:
        Tuple of noised images and the applied noise.
    """
    mean = gather(alpha_bar, t).sqrt() * x0
    variance = 1 - gather(alpha_bar, t)
    noise = torch.randn_like(x0).to(x0.device)
    noised_images = mean + (variance.sqrt() * noise)

    return noised_images, noise

# Training parameters
batch_size = 128  # Adjust this based on your GPU memory availability
learning_rate = 2e-4  # Consider experimenting with this during full training

# Placeholder for storing loss values
losses = []
val_losses = []

# Dataset for training and val
dataset = cifar10['train']
val_dataset = cifar10['test']

# Define the optimizer
optimizer = torch.optim.AdamW(unet.parameters(), lr=learning_rate)

# Training loop

for epoch in range(number_of_epochs):
  for i in tqdm(range(0, len(dataset) - batch_size, batch_size), desc="Training"):
      # Fetch and process a batch of images
      images = [dataset[idx]['img'] for idx in range(i, i + batch_size)]
      tensor_images = [img_to_tensor(img).cuda() for img in images]
      x0_batch = torch.cat(tensor_images)

      # Select random time steps for each image in the batch
      time_steps = torch.randint(0, n_steps, (batch_size,), dtype=torch.long).cuda()

      # Apply the noising process
      noised_images, true_noise = q_xt_x0(x0_batch, time_steps)

      # Predict noise using the model
      predicted_noise = unet(noised_images.float(), time_steps)
```

```
    # Calculate loss and perform backpropagation
    loss = F.mse_loss(true_noise.float(), predicted_noise)
    losses.append(loss.item())
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    with torch.no_grad():  # Disable gradient computation
        val_batch = []
        for i in range(0, len(val_dataset) - batch_size, batch_size):
          # Process a batch of validation images
          images = [val_dataset[idx]['img'] for idx in range(i, i + batch_size)]
          tensor_images = [img_to_tensor(img).cuda() for img in images]
          x0_batch = torch.cat(tensor_images)

          # Select random time steps for each image in the batch
          time_steps = torch.randint(0, n_steps, (batch_size,), dtype=torch.long).cuda()

          # Apply the noising process
          noised_images, true_noise = q_xt_x0(x0_batch, time_steps)

          # Predict noise using the model
          predicted_noise = unet(noised_images.float(), time_steps)

          # Calculate validation loss
          val_loss = F.mse_loss(true_noise.float(), predicted_noise)
          val_batch.append(val_loss.item())
    val_losses.append(sum(val_batch) / len(val_batch))
```

در کد فوق،

راه‌اندازی مدل:

- ✓ یک مدل UNet با تنظیمات کانال مشخص ایجاد می‌شود و برای انجام محاسبات سریع‌تر روی دستگاه CUDA (GPU) آماده می‌شود.

تنظیم پارامترهای آموزشی:

- ✓ n_steps تعداد مراحل زمانی در فرآیند نویز را مشخص می کند.
- ✓ بتا، آلفا و alpha_bar برای فرآیند تولید نویز تنظیم شده اند. اینها پارامترهایی هستند که نحوه اضافه شدن نویز به تصاویر را در طول زمان کنترل می کنند.
- ✓ تعداد ایپاک روی 1 تنظیم شده است، که نشان می دهد کل مجموعه داده آموزشی یک بار از مدل عبور داده شده است.

تابع q_xt_x0

- ✓ این تابع برای افزودن نویز به تصاویر تعریف شده است. یک تصویر اصلی (x0) و یک گام زمانی (t) می گیرد، سپس بر اساس این پارامترها نویز به تصویر اضافه می کند.

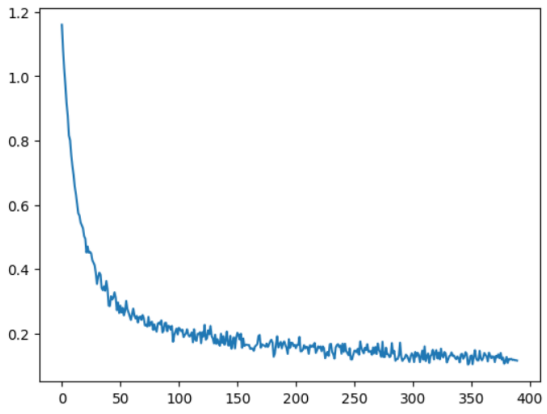
حلقه آموزش:

- ✓ این مدل بر روی مجموعه داده آموزشی CIFAR-10، با استفاده از اندازه دسته ای مشخص و نرخ یادگیری آموزش داده شده است.
- ✓ برای هر دسته از تصاویر، کد: تصاویر را به تانسور تبدیل می کند و آنها را برای پردازش توسط مدل آماده می کند. به طور تصادفی مراحل زمانی را انتخاب می کند و فرآیند نویز را برای هر تصویر در دسته اعمال می کند. تصاویر نویز شده و مراحل زمانی مربوط به آنها را به مدل UNet برای پیش بینی نویز تغذیه می کند. میانگین مربعات خطا بین نویز پیش بینی شده و نویز واقعی اعمال شده بر روی تصاویر را محاسبه می کند. در نهایت به روز رسانی وزن ها را انجام میدهد.

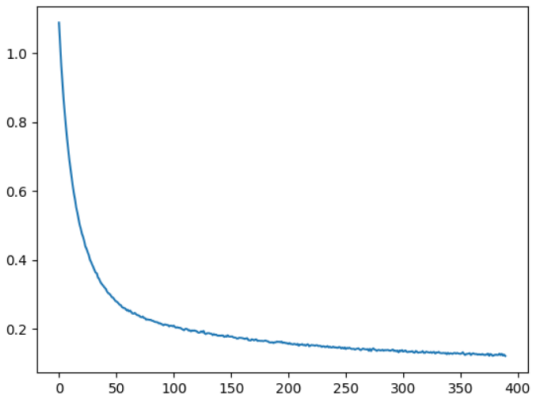
حلقه اعتبارسنجی:

- ✓ پس از هر تکرار آموزشی، عملکرد مدل بر روی یک مجموعه اعتبارسنجی (مجموعه آزمون CIFAR-10) ارزیابی می شود.
- ✓ مراحل مشابهی برای حلقه تمرین دنبال می شود، اما هیچ به روز رسانی وزن مدل انجام نمی شود. در عوض، لاس اعتبارسنجی برای نظارت بر عملکرد مدل بر روی داده‌های دیده نشده محاسبه و ذخیره می‌شود.

منحنی loss به ازای هر batch در این 1 ایپاک به صورت زیر میباشد.



شکل 25: منحنی **loss** آموزش به ازای هر **batch**برای 1 ایپاک



شکل 26 : محنی**loss** برای **val** به ازای هر **batch** برای 1 ایپاک

به کمک کد زیر فرایند reverse انجام میشود.

```
def p_xt(current_noise_image, predicted_noise, time_step):
    alpha_t = gather(alpha, time_step)
    alpha_bar_t = gather(alpha_bar, time_step)
    epsilon_coefficient = (1 - alpha_t) / (1 - alpha_bar_t) ** 0.5
    mean = 1 / (alpha_t ** 0.5) * (current_noise_image - epsilon_coefficient *
predicted_noise)
    variance = gather(beta, time_step)
    random_noise = torch.randn(current_noise_image.shape,
device=current_noise_image.device)
    return mean + (variance ** 0.5) * random_noise

# Initialize a random noise image
initial_noise_image = torch.randn(1, 3, 32, 32).cuda()
generated_images = []



# Iterate over the diffusion steps
for step in range(n_steps):
    time_step = torch.tensor(n_steps - step - 1, dtype=torch.long).cuda()
    with torch.no_grad():
        # Predict the noise using the U-Net model
        predicted_noise = unet(initial_noise_image.float(), time_step.unsqueeze(0))
        # Perform the reverse diffusion step
        initial_noise_image = p_xt(initial_noise_image, predicted_noise,
time_step.unsqueeze(0))

        # Save the generated images at specified intervals
        if step % 24 == 0:
            generated_images.append(tensor_to_image(initial_noise_image.cpu()))

# Create a composite image to display the generated images
```

```
composite_image = Image.new('RGB', size=(32 * 5, 32))
for i, img in enumerate(generated_images[:5]):
    composite_image.paste(img, ((i % 5) * 32, 0))
composite_image.resize((32 * 4 * 5, 32 * 4), Image.NEAREST)
```

در کد فوق ،

تابع مرحله معکوس (p_xt):

- ✓ این تابع روند افزایش نویز را معکوس می کند. یک تصویر نویزدار، نویز پیش‌بینی‌شده از مدل UNet و مرحله زمانی فعلی را می‌گیرد.
- ✓ میانگین توزیع معکوس را با کم کردن نویز پیش بینی شده از تصویر نویز شده و تنظیم آن بر اساس مرحله زمانی محاسبه می کند.
- ✓ نویز تصادفی با واریانس بسته به مرحله زمانی به میانگین اضافه می شود تا تصویر بعدی در فرآیند معکوس ایجاد شود.

مقداردهی اولیه:

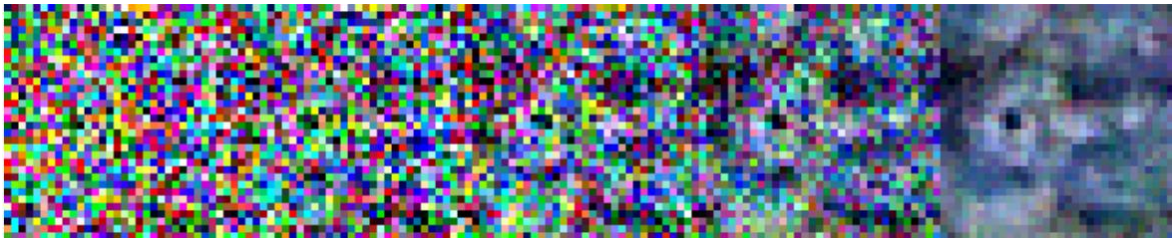
- ✓ کد با یک تصویر نویز تصادفی اولیه شروع می شود. این تصویر کاملاً نویز است و هنوز محتوای معنی‌داری ندارد.

Iterative Reverse Diffusion:

- ✓ فرآیند در مراحل انتشار به ترتیب معکوس تکرار می شود و از آخرین مرحله شروع می شود و به مرحله اول برمی گردد.
- ✓ در هر تکرار، کد: مرحله زمانی فعلی (time_step) را محاسبه می کند. از مدل UNet برای پیش بینی نویز اضافه شده در این مرحله استفاده می کند. تابع p_xt را برای معکوس کردن نویز اضافه شده در این مرحله اعمال می کند و تصویر را یک قدم به یک تصویر معنادار نزدیکتر می کند. به صورت اختیاری برخی از این تصاویر میانی را برای مشاهده بعدی ذخیره میشود.

تولید و نمایش تصاویر:

- ✓ این کد تصاویر را در فواصل زمانی معین (در این مورد هر 24 مرحله) ذخیره می کند تا فرآیند انتشار معکوس را تجسم کند یک تصویر ترکیبی برای نمایش این تصاویر ذخیره شده در کنار هم ایجاد می شود و پیشرفت از نویز به یک تصویر ساختارمندتر را نشان می دهد.



شکل 27 : فرایند **reverse** مدل آموزش دیده پس از 1 ایپاک

مشاهده میشود که تلاش شده تا تصویر معناداری ایجاد کند ولی همچنان دقت خوبی ندارد.

به همین دلیل فرایند را با ۵ ایپاک تکرار میکنیم ولی در کد تغییراتی هم میدهیم دیگر در هر ایپاک مقادیر loss در هر batch را گرفته و میانگین آن را محاسبه میکنیم و در نهایت یک عدد به عنوان loss آن ایپاک گزارش میشود بدین ترتیب کد آموزش به صورت زیر میشود.

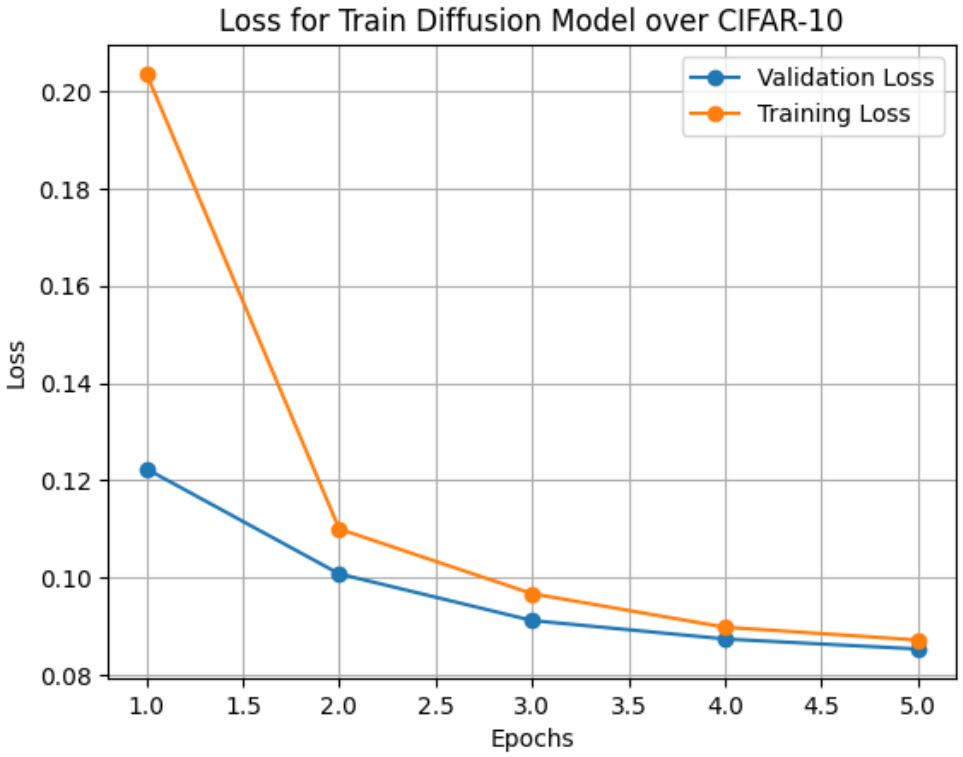
```
# Initialize the UNet model
unet = UNet(n_channels=32).cuda()

# Set up training parameters
n_steps = 100
beta = torch.linspace(0.0001, 0.04, n_steps).cuda()
alpha = 1. - beta
alpha_bar = torch.cumprod(alpha, dim=0)
number_of_epochs = 5

def q_xt_x0(x0, t):
    """
    Apply the noise process to the input images.

    Args:
        x0: Tensor representing the original batch of images.
        t: Tensor of time steps for each image in the batch.

    Returns:
        Tuple of noised images and the applied noise.
    """
    mean = gather(alpha_bar, t).sqrt() * x0
    variance = 1 - gather(alpha_bar, t)
    noise = torch.randn_like(x0).to(x0.device)
    noised_images = mean + (variance.sqrt() * noise)

    return noised_images, noise

# Training parameters
batch_size = 128  # Adjust this based on your GPU memory availability
learning_rate = 2e-4  # Consider experimenting with this during full training
```

```
# Placeholder for storing loss values
losses = []
val_losses = []

# Dataset for training and val
dataset = cifar10['train']
val_dataset = cifar10['test']

# Define the optimizer
optimizer = torch.optim.AdamW(unet.parameters(), lr=learning_rate)

# Training loop

for epoch in range(number_of_epochs):
  print(f"Epoch: {epoch+1}")
  unet.train()
  train_loss_batch = []
  for i in tqdm(range(0, len(dataset) - batch_size, batch_size), desc="Training"):
      # Fetch and process a batch of images
      images = [dataset[idx]['img'] for idx in range(i, i + batch_size)]
      tensor_images = [img_to_tensor(img).cuda() for img in images]
      x0_batch = torch.cat(tensor_images)

      # Select random time steps for each image in the batch
      time_steps = torch.randint(0, n_steps, (batch_size,), dtype=torch.long).cuda()

      # Apply the noising process
      noised_images, true_noise = q_xt_x0(x0_batch, time_steps)

      # Predict noise using the model
      predicted_noise = unet(noised_images.float(), time_steps)

      # Calculate loss and perform backpropagation
      loss = F.mse_loss(true_noise.float(), predicted_noise)
      train_loss_batch.append(loss.item())
      optimizer.zero_grad()
      loss.backward()
      optimizer.step()
  unet.eval()
  val_batch = []
  with torch.no_grad():  # Disable gradient computation
      for i in range(0, len(val_dataset) - batch_size, batch_size):
        # Process a batch of validation images
        images = [val_dataset[idx]['img'] for idx in range(i, i + batch_size)]
        tensor_images = [img_to_tensor(img).cuda() for img in images]
        x0_batch = torch.cat(tensor_images)
```

```
        # Select random time steps for each image in the batch
        time_steps = torch.randint(0, n_steps, (batch_size,), dtype=torch.long).cuda()

        # Apply the noising process
        noised_images, true_noise = q_xt_x0(x0_batch, time_steps)

        # Predict noise using the model
        predicted_noise = unet(noised_images.float(), time_steps)

        # Calculate validation loss
        val_loss = F.mse_loss(true_noise.float(), predicted_noise)
        val_batch.append(val_loss.item())
    losses.append(sum(train_loss_batch)/len(train_loss_batch))
    val_losses.append(sum(val_batch) / len(val_batch))
    print(f"Train loss : {sum(train_loss_batch)/len(train_loss_batch)}" )
    print(f"Val loss : {sum(val_batch)/len(val_batch)}" )
```

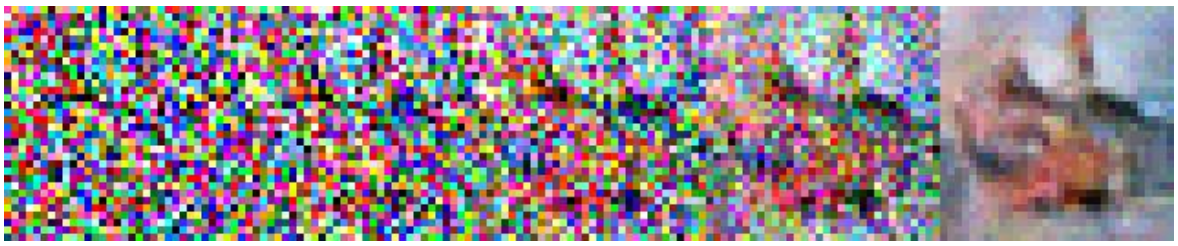
کد فوق فرایندآموزش را تا ۵ ایپاک پیگیری میکند که این عدد هم تجربی و بر اساس رفتار loss val میباشد.

منحنی loss در این حالت به صورت زیر میشود.



شکل 28 : محنی **loss** پس از آموزش مدل **diffusion** به ازای ۵ ایپاک

خروجی در نهایت به صورت زیر میشود. که نشان از بسیار بهتر شدن مدل دارد.



شکل 29 : خروجی نویزی به تصویر معنادار مدل با ۵ ایپاک

سوال ۱۵ : تولید ۱۰۰ نمونه تصویر

جهت تولید ۱۰۰ تصویر میتوان از کد زیر استفاده کرد.

```
#@title Make and show 10 examples:
x = torch.randn(100, 3, 32, 32).cuda() # Start with random noise
ims = []
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))

for i in range(100):
  ims.append(tensor_to_image(x[i].unsqueeze(0).cpu()))

image = Image.new('RGB', size=(32*10, 32*10))
for i, im in enumerate(ims):
  image.paste(im, ((i%10)*32, 32*(i//10)))
image.resize((32*4*10, 32*4*10), Image.NEAREST)
```

در کد فوق

- ✓ با نویز تصادفی شروع میشود: کد ابتدا مجموعه ای از 100 الگوی نویز تصادفی را ایجاد می کند. اینها به عنوان نقطه شروع برای تولید تصویر عمل می کنند.
- ✓ ایجاد تصاویر: از مدل برای تبدیل این نویز تصادفی به تصاویر در چندین مرحله استفاده می کند.
- ✓ تبدیل به تصاویر: هر یک از الگوهای به دست آمده به فرمت تصویر تبدیل می شود.

- ✓ ایجاد یک شبکه از تصاویر: این 100 تصویر در یک شبکه 10*10 مرتب شده اند و یک تصویر ترکیبی بزرگتر ایجاد می کنند.
- ✓ تغییر اندازه برای نمایش: در نهایت، اندازه تصویر ترکیبی برای دید بهتر تغییر می‌کند.

شکل 30 : تولید ۱۰۰ عکس با مدل **diffusion**

همانطور که مشاهدات خروجی ها به نسبت خوب و مناسب میباشند.

سوال ۱۶ : ارزیابی مدل با FID

در بخش قبل به طور کامل فرایند پیاده سازی FID (با استفاده از کتاب خانه های آماده و یا از اسکرچ آمده است)

در اینجا مجددا توضیح داده نمیشود.

همانطور که قبل تر توضیح داده شد برای این بخش نیاز به ۲ فولدر حاوی تصاویر اصلی و تولیدی میباشد که در اینجا 3000 سمپل تولید شده است.

به کمک کد زیر تصاویر generate شده تولید میشود.

```
from torchvision.utils import save_image
import os
from tqdm import tqdm


# Create a directory to save generated images
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'generated_images/sample_{k:04d}.png')
```

و به کمک کد زیر نیز تصاویر CIFAR10 تست تولید میشود.

```
import torchvision
import torchvision.transforms as transforms

# Create a directory for CIFAR-10 images
os.makedirs('cifar10_images', exist_ok=True)

# Load and normalize CIFAR-10 dataset
transform = transforms.Compose([
    transforms.ToTensor()
    #transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
cifar10_dataset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True,
transform=transform)

# Save a subset of CIFAR-10 images
for i, (image, _) in enumerate(cifar10_dataset):
    torchvision.utils.save_image(image, f'cifar10_images/cifar10_{i:04d}.png')
    if i >= 2999:  # Reduce to 3000 images
        break
```

در نهایت نیز به کمک FID TORCH مقدار FID به کمک دستور زیر محاسبه میشود.

```
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_images'

# Calculate the FID score
fid_value = calculate_fid_given_paths([path_to_generated_images, path_to_real_images],
                                      batch_size=50,  # You can change the batch size
depending on your machine's capability
                                      device='cuda',  # Or 'cpu' if you're not using a GPU
                                      dims=2048)  # The number of dimensions for feature
extraction; 2048 is standard for InceptionV3
print(f'FID score: {fid_value}')
```

```
100%|■■■■■■■■■■| 60/60 [00:13<00:00,  4.41it/s]
100%|■■■■■■■■■■| 60/60 [00:13<00:00,  4.44it/s]
FID score: 121.43642801637861
```

مشاهده میشود که مقدار FID برابر با 121 میشود که همانطور که در بخش های قبلی گفته شده این عدد مناسب نمیباشد یکی از علت ها میتواند تعداد کم ۳۰۰۰ داده یا عدم اموزش کافی و مناسب (نیازمند تعداد ایپاک بیشتر یا تغییراتی در هایپرپارامتر ها و ...) میباشد.