به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

مدل‌های مولد عمیق

تمرین شماره ۲پارت دوم

نام و نام خانوادگی	مهسا ندافی قهنویه
شماره دانشجویی	۸۱۰۱۰۰۴۹۰
تاریخ ارسال گزارش	

فهرست گزارش سوالات

سوال ۱ – Generative Adversarial Networks (GANs) ۳

سوال ۲ – Diffusion Model ۱۳

سوال ۱ – Generative Adversarial Networks (GANs)

A

کلاس `PixelShuffle` در PyTorch، ابزاری برای انجام کانولوشن با استفاده از sub-pixel convolution است. این کلاس، ابعاد یک تانسور با شکل (*, C × r^2, H, W) را به شکل (*, C, H × r, W × r) تغییر می‌دهد، که در آن r یک عامل افزایش مقیاس است. این کلاس برای پیاده‌سازی کانولوشن با sub-pixel convolution با گام 1/r مناسب است.

منشاء و هدف PixelShuffle که در اصل برای کارها ی با وضوح فوق العاده توسعه یافته بود، که وضوح تصاویر را در شبکه های عصبی کانولوشنال به طور موثر بهتر میکند. هدف اصلی PixelShuffle تبدیل نقشه های ویژگی با ابعاد کانال بالاتر به نقشه های ویژگی با وضوح فضایی بالاتر است .این اغلب در ارتباط با لایه های کانولوشنی sub-pixel استفاده می شود.

Pixel shuffleپیکسل‌ها را در نقشه ویژگی‌ها مجدداً مرتب می‌کند تا به ارتقاء مورد نظر برسد .این شامل گرفتن گروه هایی از پیکسل های همسایه و مرتب کردن مجدد آنها در یک شبکه بزرگتر است .ترتیب خاص به اجرا بستگی دارد، اما اغلب شامل در هم آمیختن مقادیر است.

در این مسئله استفلده از PixelShuffle نیز باعث افزایش وضوح و کیفیت تصاویر تولیدی در generator میشود.

B

بخش generator:

```
class Generator(torch.nn.Module):
    def __init__(self, z_dim, num_channels=1):
        super().__init__()
        self.z_dim = z_dim
         ############################## TODO ##############################
        # Complete this part according to the introduced generator table on
the provided manuscript!
        # YOUR CODE STARTS HERE

        self.generator = nn.Sequential(
            nn.Linear(self.z_dim, 512),
            nn.BatchNorm1d(512),
            nn.ReLU(),

            nn.Linear(512,  64 * 7 * 7),  # Adjusted output size for
reshaping before PixelShuffle
```

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

            nn.Unflatten(1, (64, 7, 7)),  # Reshape to image-like structure

            nn.PixelShuffle(2),  # Pixel shuffle for upsampling

            nn.Conv2d(16, 32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.PixelShuffle(2),  # Pixel shuffle for upsampling
            nn.Conv2d(8, 1, kernel_size=3, padding=1)  # Output with a
single channel
            #nn.Tanh()  # Assuming that you want the output in the range [-
1, 1]
        )

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

معماری نهایی بصورت خلاصه:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                  [-1, 512]          33,280
       BatchNorm1d-2                  [-1, 512]           1,024
              ReLU-3                  [-1, 512]               0
            Linear-4                 [-1, 3136]       1,608,768
       BatchNorm1d-5                 [-1, 3136]           6,272
              ReLU-6                 [-1, 3136]               0
         Unflatten-7            [-1, 64, 7, 7]               0
      PixelShuffle-8           [-1, 16, 14, 14]               0
            Conv2d-9           [-1, 32, 14, 14]           4,640
      BatchNorm2d-10           [-1, 32, 14, 14]              64
             ReLU-11           [-1, 32, 14, 14]               0
      PixelShuffle-12            [-1, 8, 28, 28]               0
           Conv2d-13            [-1, 1, 28, 28]              73
================================================================
Total params: 1,654,121
Trainable params: 1,654,121
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.33
Params size (MB): 6.31
Estimated Total Size (MB): 6.64
----------------------------------------------------------------
```

شکل ۱-معماری شبکه **GENERATOR**

ورودی های PixelShuffle باید بیشتر از سه بعد باشد پس قبل از آن تغییر سایز را استفاده کرده ایم.

:Discriminator

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

        ############################## TODO ##############################
        # Complete this part according to the in
        # troduced discriminator table on the provided manuscript!

        # YOUR CODE STARTS HERE
        self.discriminator = nn.Sequential(
            nn.Conv2d(num_channels, 32, kernel_size=4, stride=2,
padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2, padding=1),
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(64 * 7 * 7, 512),
            nn.ReLU(),
            nn.Linear(512, 1),
            nn.Sigmoid()  # Sigmoid activation at the end
        )

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

معماری بصورت خلاصه:

```
----------------------------------------------------------------
       Layer (type)               Output Shape         Param #
================================================================
           Conv2d-1          [-1, 32, 14, 14]             544
             ReLU-2          [-1, 32, 14, 14]               0
           Conv2d-3            [-1, 64, 7, 7]          32,832
             ReLU-4            [-1, 64, 7, 7]               0
          Flatten-5                [-1, 3136]               0
           Linear-6                 [-1, 512]       1,606,144
             ReLU-7                 [-1, 512]               0
           Linear-8                   [-1, 1]             513
          Sigmoid-9                   [-1, 1]               0
================================================================
Total params: 1,640,033
Trainable params: 1,640,033
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.18
Params size (MB): 6.26
Estimated Total Size (MB): 6.43
----------------------------------------------------------------
```

شکل 2-معماری شبکه**DISCRIMINATOR**

به منظور خارج شدن خروجی از حالت logit از سیگموید در لایه آخر استفاده شده است.

C

تابع خطا در قسمت gan_step بصورت زیر پیاده سازی شده است:

```
z_fake = torch.randn(x_real.size(0), self.num_latents, device=self.device)
        x_fake = self.model.g(z_fake)
        y_fake = self.model.d(x_fake)

        # Generator loss (minimize log(1 - D(G(z))))
        g_loss = nn.BCELoss()(y_fake,
torch.ones_like(y_fake).to(self.device))

        # Backpropagate and update generator weights
        g_loss.backward()
        self.optimizers[0].step()

        # Discriminator step
        y_real_pred = self.model.d(x_real)
        y_fake_pred = self.model.d(x_fake.detach())

        # Discriminator loss (minimize log(D(x)) + log(1 - D(G(z))))
        d_real_loss = nn.BCELoss()(y_real_pred,
torch.ones_like(y_real_pred).to(self.device))
        d_fake_loss = nn.BCELoss()(y_fake_pred,
torch.zeros_like(y_fake_pred).to(self.device))
        d_loss = d_real_loss + d_fake_loss
```

D

روند آموزش مدل بصورت زیر پیاده شده است:

```
# train models for multiple epochs
        with tqdm(total=int(self.iter_max)) as pbar:
            for epoch in range(20):
                for x, y in train_loader:

                    x_real, y_real = self.build_input(x, y)

                    # Perform a GAN step
                    d_loss, g_loss = self.gan_step(x_real, y_real)

                    # Append losses to lists
                    G_losses.append(g_loss)
                    D_losses.append(d_loss)

                    # Update progress bar and log losses
```

```
                    pbar.update(1)
                    pbar.set_description(f'Epoch {epoch +
1}/{self.iter_max}, G_loss: {g_loss:.4f}, D_loss: {d_loss:.4f}')

                    # Checkpoint and log
                    self.checkpoint_and_log(global_step, (g_loss, d_loss),
None)
                    global_step += 1

                    # Break loop if reaching the maximum number of
iterations
                    if global_step >= self.iter_max:
                        break
```

روند آموزش مدل برای lr=0.3e-4

```
Epoch 10000/10000, G_loss: 0.5780, D_loss: 1.0035: : 19978it [10:22,
32.11it/s]
```

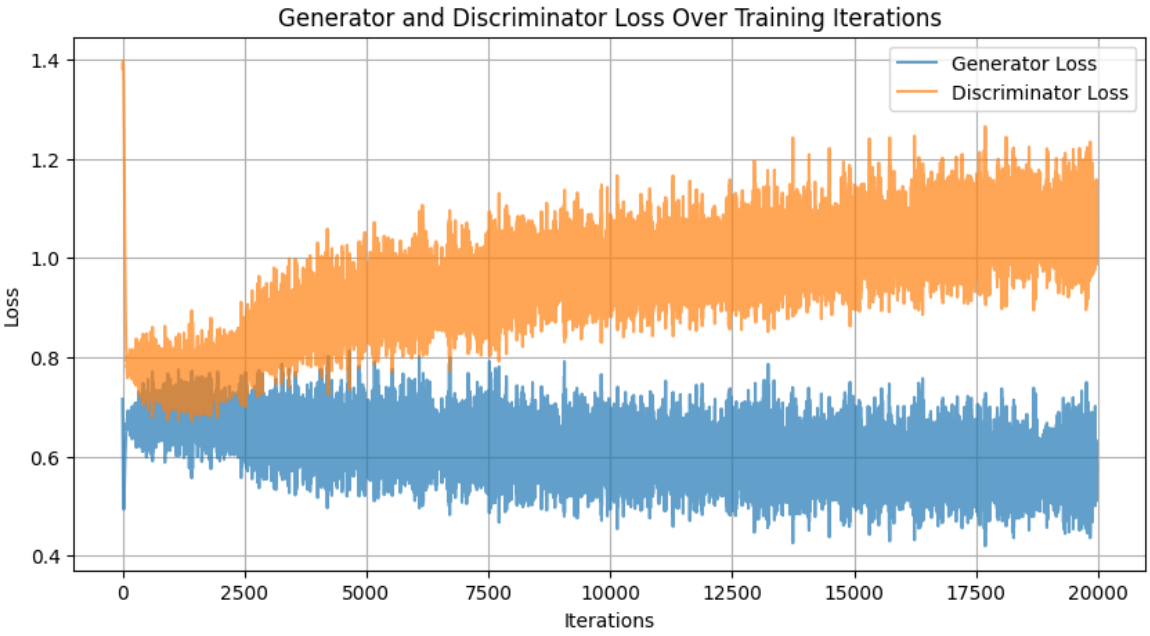
E

a •

کد مربوط به رسم توابع خطا:

```
plt.figure(figsize=(10, 5))
plt.title("Generator and Discriminator Loss Over Training Iterations")
plt.plot(G_losses, label="Generator Loss", alpha=0.7)
plt.plot(D_losses, label="Discriminator Loss", alpha=0.7)
plt.xlabel("Iterations")
plt.ylabel("Loss")
plt.legend()
plt.grid("True")
plt.show()
```

نمودار حاصل شده:



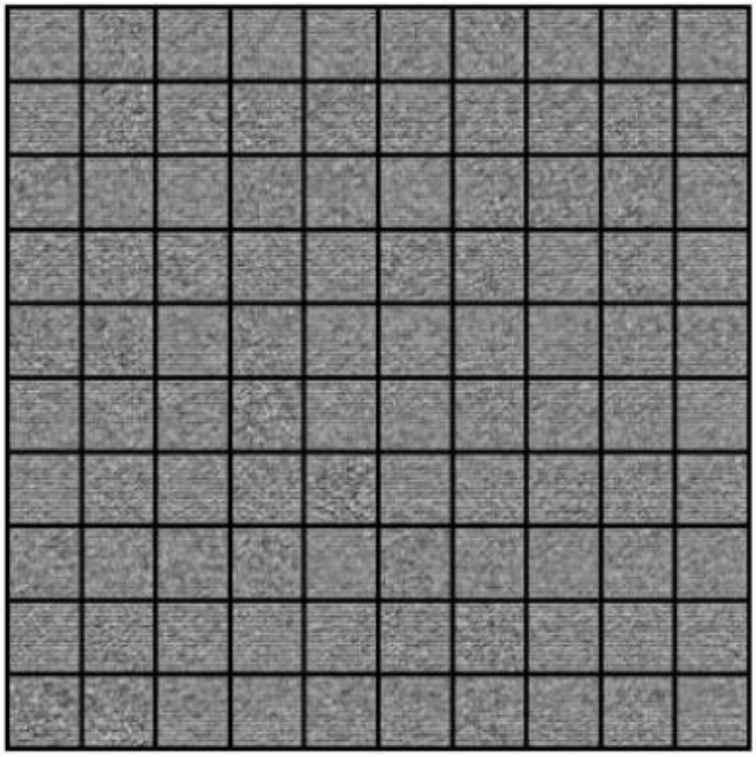
شکل ۳-نمودار های لاس به ازای تکرار های انجام شده بر روی هر **BATCH**

نمودار لاس مشام میدهد که discriminator بین نمونه های جعلی در ابتدای آموزش به خوبی تمایز قائل میشود اما با گذشت زمان generator میتواند آن را گول بزند و تصویری تولید کند که لاس discriminator آن افزایش یابد این در حالی است که generator در حال تولید تصاویری شبیه تر به تصاویر اصلی است.

- b

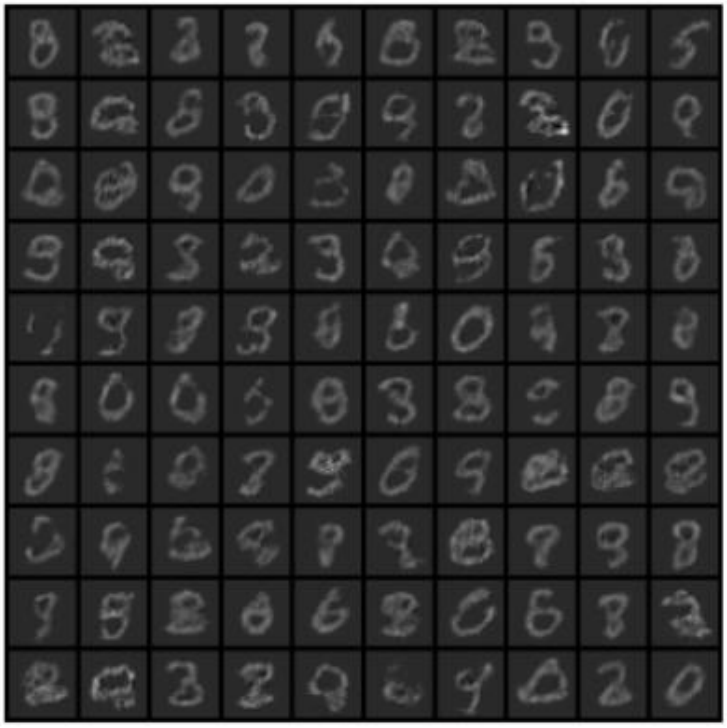
در این قسمت به تولید نمونه توسط مدل در فرایند آموزش با استفاده از کد داده شده میپردازیم تقریبا ۲۰ ایپاک آموزش را انجام داده ایم.

ایپاک ابتدایی:



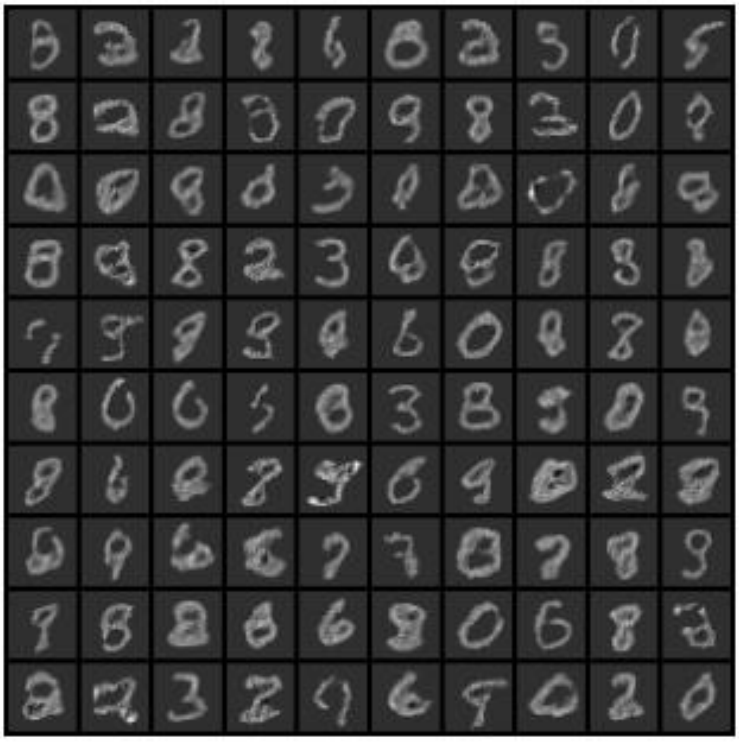
شکل ۴-۱۰۰ نمونه تولید شده در ایپاک ابتدایی

ایپاک میانی:



شکل ۵-۱۰۰ نمونه تولید شده در ایپاک میانی

ایپاک نهایی:



شکل ۶-۱۰۰ نمونه تولید شده در ایپاک نهایی

بعد از گذشت زمان gan دچار mode collaps شده و در واقع به سمت تولید داده های شبیه به هم مثلا صفر در شکل بالا میرود. این قضع را در افزایش لاس discriminator در تکرار های پایانی نیز میتوان مشاهده کرد.

- C

FID یک معیار برای ارزیابی کیفیت تصاویر تولید شده توسط مدلهای generative است. در این معیار از دو مجموعه تصاویر، یکی به عنوان مجموعه تصاویر مرجع و دیگری به عنوان مجموعه تصاویر تولید شده توسط مدل تولیدی استفاده می‌شود. سپس، دو توزیع گاوسی برای هر دو مجموعه تصاویر به صورت جداگانه تخمین زده می‌شود. در نهایت، امتیاز FID برابر با فاصله‌ی Fréchet بین دو توزیع گاوسی تخمین زده شده به عنوان عددی نمایش داده میشود.

امتیاز پایین تر FID نشان می دهد که تصاویر تو لید شده بیشتر شبیه به تصاویر واقعی هستند. امتیاز بالای FID نشان دهنده اختلاف بیشتر بین تصاویر تولید شده و واقعی است که نشان دهنده مشکلات کیفیت یا تنوع تصاویر تولید شده است.

بدین منظور در ابتدا تصاویر واقعی و تولید شده را در پوشه های مجزا ذخیره میکنیم:

```
preprocess = transforms.ToTensor()
checkpoint = torch.load('model_19000.pt')
model = checkpoint[0]
def save_real_images(dataset, num_images, folder='real_images'):
    if not os.path.exists(folder):
        os.makedirs(folder)

    for idx in range(num_images):
        image, _ = dataset[idx]
        save_image(image, os.path.join(folder, f'real_{idx}.png'))
mnist_dataset = datasets.MNIST(root='./data', train=False, download=True,
transform=preprocess)
save_real_images(mnist_dataset, 500)
def generate_images(model, num_images, folder='fake_images'):
    if not os.path.exists(folder):
        os.makedirs(folder)
    model.eval()
    for i in range(num_images):
        with torch.no_grad():
            noise = torch.randn(1, 64).to(device)
            generated_image = model(noise)
            save_image(generated_image, os.path.join(folder,
f'fake_{i}.png'))
generate_images(model, 500)
```

در نهایت این معیار را بر روی ۵۰۰ عکس توسط شکل زیر تست میکنیم:

```
from torchmetrics.image.fid import FrechetInceptionDistance
fake_images = torch.zeros(500,3,28,28)
real_images = torch.zeros(500,3,28,28)

for i in range(500):
  fake_images[i]  =
torchvision.io.read_image(f'/content/fake_images/fake_{i}.png')
  real_images[i]  =
torchvision.io.read_image(f'/content/real_images/real_{i}.png')

fid = FrechetInceptionDistance(feature=192, normalize=True)
# generate two slightly overlapping image intensity distributions

fid.update(real_images.type(torch.uint8), real=True)
fid.update(fake_images.type(torch.uint8), real=False)
fid.compute()
```

خروجی برابر 38.8508 است که میتواند به مسئله تولید تصاویر تکراری در مدل اشاره کند.

F

WGAN

WGANبه مشکلات بی ثباتی آموزشی که GAN با آن مواجه است، می پردازد .از فاصله Wasserstein (Earth Mover's distance) به جای از لاس GAN اصلی استفاده می کند که منجر به شیب نرم تر در طول تمرین می شود .این ثبات منجر به همگرایی قابل اعتمادتر و پویایی تمرین بهتر می شود. همچنین خروجی های discriminator این مدل معنا دار ترمیشود و این موضوع خود را در گرادیان نشان میدهد و این به مولد اجازه می دهد تا به طور موثرتری یاد بگیرد و از vanishing gradients یا Mode Collapse جلوگیری کند که منجر به تولید نمونه متنوع تر و واقعی تر می شود.

PG-GAN

یک استراتژی رشد progressive را معرفی می کند که با تصاویر با وضوح پایین شروع می شود و به تدریج وضوح را در طول آموزش افزایش می دهد، که به کاهش چالش های آموزشی کمک می کند، از Mode Collapse جلوگیری می کند و منجر به ترکیب تصویر با کیفیت بالاتر می شود همچنین امکان آموزش پایدار و همگرایی بهتر را فراهم می کند .رویکرد progressive به PG-GAN اجازه می دهد تا تصاویری با وضوح بالا در ابعاد بزرگ و تنوع بهتر در مقایسه با GAN های استاندارد که ناپایدار هستند تولید کند.

BIG GAN

GAN ها برای سنتز تصویر موثر هستند اما معمولاً تصاویر کوچکی تولید می کنند (مثلاً x64۶۴ یا x128۱۲۸ پیکسل) همچنین به تنظیم هایپر پارامتر های دقیق تر و انتخاب معماری مناسب نیاز دارند. BigGAN این مشکل را با بزرگ‌کردن class-conditional image synthesis بر طرف میکند, که منجر به تصاویری با وضوح و کیفیت بالا می‌شود. همچنین اندازه مدل و اندازه batch را افزایش می دهد .مدل های بزرگتر می توانند تصاویر بسیار بزرگتر و با کیفیت بالاتری تولید کنند.

STYLE GAN

StyleGANبا جدا کردن سبک (فضای latent) و ساختار (معماری شبکه) امکان کنترل دقیق بر روی تصاویر تولید شده را می دهد StyleGAN .تصاویری با وضوح بالا با کیفیت بصری چشمگیر تولید می کند که باعث انعطاف بیشتر این مدل میشود .در مقایسه با GAN ها ، مصنوعات را کاهش می دهد و واقع گرایی را بهبود می بخشد.همچنین انتقال smooth بین بردارهای latent را نشان می دهد، که امکان درونیابی seamless بین سبک های مختلف تصویر را فراهم می کند.

سوال ۲ – Diffusion Model

سوالات تئوری:

سوال ۱

همانطور که در جدول زیر آمده است مدلهای VAE,Normalizing flow کیفیت تولید بالایی نداشته اما تنوع خوبی در تولید داده ها دارند و سرعت آنها نیز بالا است.

مدلهای GAN سرعت و کیفیت بالایی دارند ولی تنوع خوبی ندارند و دچار مشکل mode collapse میشوند.

در نهایت مدلهای Diffusion کیفیت و تنوع خوبی در تولید داده دارند اما به علت طی کردن مراحل رو به عقب زیاد سرعت کمی دارند.

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

سوال ۲

برای نمونه اول و دوم داریم:

$$x_1 = \sqrt{\alpha_1}x_0 + \sqrt{1-\alpha_1}\varepsilon_1, \varepsilon_1 = N(0,I), 0 \leqslant \alpha_{\leqslant 1}$$

$$x_2 = \sqrt{\alpha_2}x_1 + \sqrt{1-\alpha_2}\varepsilon_2$$

$$\Rightarrow x_2 = \sqrt{\alpha_2}\left(\sqrt{\alpha_1}x_0 + \sqrt{1-\alpha_1}\varepsilon_1\right) + \sqrt{1-\alpha_2}\varepsilon_2$$

$$= \sqrt{\alpha_1\alpha_2}x_0 + \sqrt{\alpha_2-\alpha_1\alpha_2}\varepsilon_1 + \sqrt{1-\alpha_2}\varepsilon_2$$

$$= \sqrt{\alpha_1\alpha_2}x_0 + \sqrt{1-\alpha_1\alpha_2}\tilde{\varepsilon},\ \tilde{\varepsilon} \sim N(0, 1-\alpha_1\alpha_2)$$

در معادلات بالا در انتها با توجه به اینکه $\varepsilon_1, \varepsilon_2 = N(0,I)$ میتوان به جای آنها از نویز دیگری استفاده کرده که میانگین آن برابر با جمع میانگین در ضرایب آنها و واریانس آن برابر با ضرب واریانس در ضرایب آنها به توان دو می باشد.

پس برای یک نمونه t ام داریم:

$$x_{t-1} = \sqrt{\alpha_{t-1}}x_{t-2} + \sqrt{1-\alpha_{t-1}}\varepsilon_{t-1}$$

$$\Rightarrow x_t = \sqrt{\alpha_t}x_{t-1} + \sqrt{1-\alpha_t}\varepsilon_t = \sqrt{\alpha_t}\left(\sqrt{\alpha_{t-1}}x_{t-2} + \sqrt{1-\alpha_{t-1}}\varepsilon_{t-1}\right) + \sqrt{1-\alpha_t}\varepsilon_t$$

$$= \sqrt{\alpha_t\alpha_{t-1}}\cdot x_{t-2} + \sqrt{1-\alpha_t\alpha_{t-1}}\tilde{\varepsilon}$$

$$\vdots$$

$$= \sqrt{\prod_{i=1}^{t}\alpha_i}\, x_0 + \sqrt{1-\prod_{i=1}^{t}\alpha_i}\,.\varepsilon = \sqrt{\overline{\alpha}_t}x_0 + \sqrt{1-\overline{\alpha}_t}\varepsilon\,, \varepsilon \sim N(0, 1-\overline{\alpha}_t)$$

سوال ۳

همانطور که نویز استفاده شده در مسیر forward گوسی فرض میشود در مسیر backward نیز به شرط کوچک بودن استپ ها نویز تخمین زده شده در هر مرحله که با استفاده از $q(x_{t-1}|x_t)$

تخمین زده میشود را میتوان گوسی فرض کرد.برای درک بهت بهتر است به مسیر forward توجه کرد که نویز به صورت تدریجی به داده ها اضافه می شود تا زمانی که به نویز گوسی تبدیل شود .مسیر backward اساساً معکوس است، جایی که ما به طور مکرر نویزها را حذف می کنیم.

برای محاسبه ی $q(x_{t-1}|x_t)$ باید از تابع $x_t = a\, x_{t-1} + b\, \varepsilon$ استفاده کرد اما چون x_t و نویز از یکدیگر مستقل نیستند و با توجه به قانون بیز این تایع فرم بسته ندارد پس از تابع $q(x_{t-1} \mid x_0, x_t) = N(\mu, \sum)$ استفاده میکنیم که دارای فرم بسته است.

سوال ۴

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

L_T همان **Prior Matching** است.

در تلاش است که توزیع نمونه T ام که یک نویز گوسی توسط P در مسیر Backward است را به توزیع نمونه نهایی به شرط مشاهده عکس واقعی که در مسیر forward تولید میشود نزدیک کند. این ترم نشان میدهد که چقدر توزیع نویزی شده نهایی که از ورودی به دست آمده است، به توزیع گوسی پیشین (Standard Guassian Prior) نزدیک است و کمک میکند تا متوجه شویم تا چه میزان همخوانی دارد.

L_t همان **Denoising Matching** است.

ما در حالت ایده آل میخواهیم رابطه ی $p(x_{t-1} \mid x_t) = q(x_{t-1} \mid x_t)$ برقرار باشد اما با توجه به اینکه $q(x_{t-1} \mid x_t)$ فرم بسته نداشته و با توجه به قانون بیز نرمال نیست پس $q(x_{t-1} \mid x_{0,}x_t)$ که دارای فرم بسته است و در مقاله نیز مطرح شده است را به جای آن قرار میدهیم در هر مرحله از زمان سعی میشود که نویز تخمین زده شده توسط p_θ که در واقع تتا پارامترهای شبکه در حال آموزش است با نویزی که در همان مرحله توسط q به تصویر اعمال شده است دارای توزیع های مشابهی باشند.

L_0 همان **Reconstruction Term** است.

این ترم بر اساس داده ی مشاهده شده ی x_1 به پیش بینی نویز در مرحله اول و در نهایت حدس تصویر اصلی در مسیر backward میپردازد این تعریف با منفی لگاریتم likelihood در تابع خطا ظاهر میشود.همچنین این ترم بر بازسازی initial state بر اساس داده مشاهده شده $\boldsymbol{x_1}$ تمرکز دارد و ارزیابی میکند که مدل تا چه حد میتواند تصویر اصلی را با استفاده از داده های موجود در $\boldsymbol{x_1}$ تخمین بزند.

$$\underbrace{\mathbb{E}_{q(\boldsymbol{x}_1|\boldsymbol{x}_0)}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}_0|\boldsymbol{x}_1)\right]}_{\text{reconstruction term}} - \underbrace{D_{\mathrm{KL}}(q(\boldsymbol{x}_T|\boldsymbol{x}_0) \parallel p(\boldsymbol{x}_T))}_{\text{prior matching term}} - \sum_{t=2}^{T} \underbrace{\mathbb{E}_{q(\boldsymbol{x}_t|\boldsymbol{x}_0)}\left[D_{\mathrm{KL}}(q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t))\right]}_{\text{denoising matching term}}$$

سوال ۵

دو ترم prior matching و reconstruction در نظر گرفته نشده است دلیل اول این است که ترم denoising جمعی بر تمام زمان ها برای آموزش تتا در مسیر backward است که در مقابل دو ترم دیگر تاثیر بسزایی دارد.

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

همچنین q پارامتری برای آموزش ندارد پس رسما ترم دوم نیز تاثیری در تابع لاس ندارد علت این انتخاب به دلیل ثابت شدن واریانسهاست. بنابراین، فرآیند بهینه سازی شامل تنظیم یا یادگیری پارامترها ی مربوط به L_T نمیشود. این تصمیم باعث ساده تر شدن فرآیند بهینه سازی بر روی سایر اجزای قابل تنظیم در مدل میشود و تمرکز را از روی پارامترهای غیرقابل آموزش برمیدارد. همچنین نویسنده ذکر کرده است که بدون reconstruction term نتیجه ی بهتری گرفته است.

سوال ۶

علت در نظر گرفتن توزیع گوسی سادگی محاسباتی است که به همراه دارد چرا که در نهایت برای محاسبه kl دو تابع گوزی باید میانگین آنها را به یکدیگر نزدیک کنیم حال اگر در فرایند رو به عقب

$p_\theta(x_{t-1} \mid x_t)$ را گوسی فرض نکنیم kl فرم بسته نخواهد داشت و دچار پیچیدگی محاسباتی میشود این در حالی است که ممکن است توزیع پیچیده تر بهتر بتواند روند کاهش نویز را تقریب بزند و مدلسازی دقیق تری انجام دهد.

یک توزیع پیچیده ممکن است نیازمند عملیات ریاضی پیچیده تر و احتمالاً روابط غیرخطی برای دقیقتر مدل کردن فرآیند معکوس باشد. این امر ممکن است منجر به افزایش نیازهای محاسباتی در هر دو مرحله ی آموزش و استنتاج شود و به طور ویژه بر کارآیی و قابلیت مقیاس پذیری مدل تأثیر بگذارد.

سوال ۷

لاس فانکشن که بصورت زیر معرفی میشود:

$$L_{t-1} := D_{KL}(q(x_{t-1}|x_t, x_0)||p_\theta(x_{t-1}|x_t))$$

همچنین طبق تعاریف مقاله داریم:

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

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

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

$$p_\theta(x_{t-1}|x_t) = \mathcal{N}(x_{t-1}; \mu_\theta(x_t, t), \sigma^2 I)$$

در نهایت در محاسبه ی kl دو توزیع گوسی سعی میشود که میانگین آنها به یکدیگر نزدیک شوند پس داریم:

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

حال از repareterization trick برای جایگذاری x_0 استفاده میکنیم:

$$x_t := \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon$$

یعنی x_0 حاصل از معادله بالا را استفاده میکنیم.

$$\mu_\theta(x_t, x_0) = \frac{1}{\sqrt{\bar{x}_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(x_t, t)\right)$$

$$\bar{\mu}(x_t, x_0) = \frac{1}{\sqrt{\bar{x}_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_t\right)$$

در نهایت بعد از سازی به معادله زیر میرسیم:

$$L_{\text{simple}}(\theta) := \mathbb{E}_{t,\mathbf{x}_0,\boldsymbol{\epsilon}}\left[\left\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t)\right\|^2\right]$$

سوال ۸

پارامتر t *به روش* Time Embedding *در شبکه* U-Net *حضور یافته است:*

در ابتدا این لایه در کلاس TimeEmbedding *بصورت* sinusoidal position embeddings *توسط فرمول های زیر ساخته شده*

$$\overrightarrow{p_t}^{(i)} = f(t)^{(i)} := \begin{cases} \sin(\omega_k . t), & \text{if } i = 2k \\ \cos(\omega_k . t), & \text{if } i = 2k+1 \end{cases} \quad \omega_k = \frac{1}{10000^{2k/d}}$$

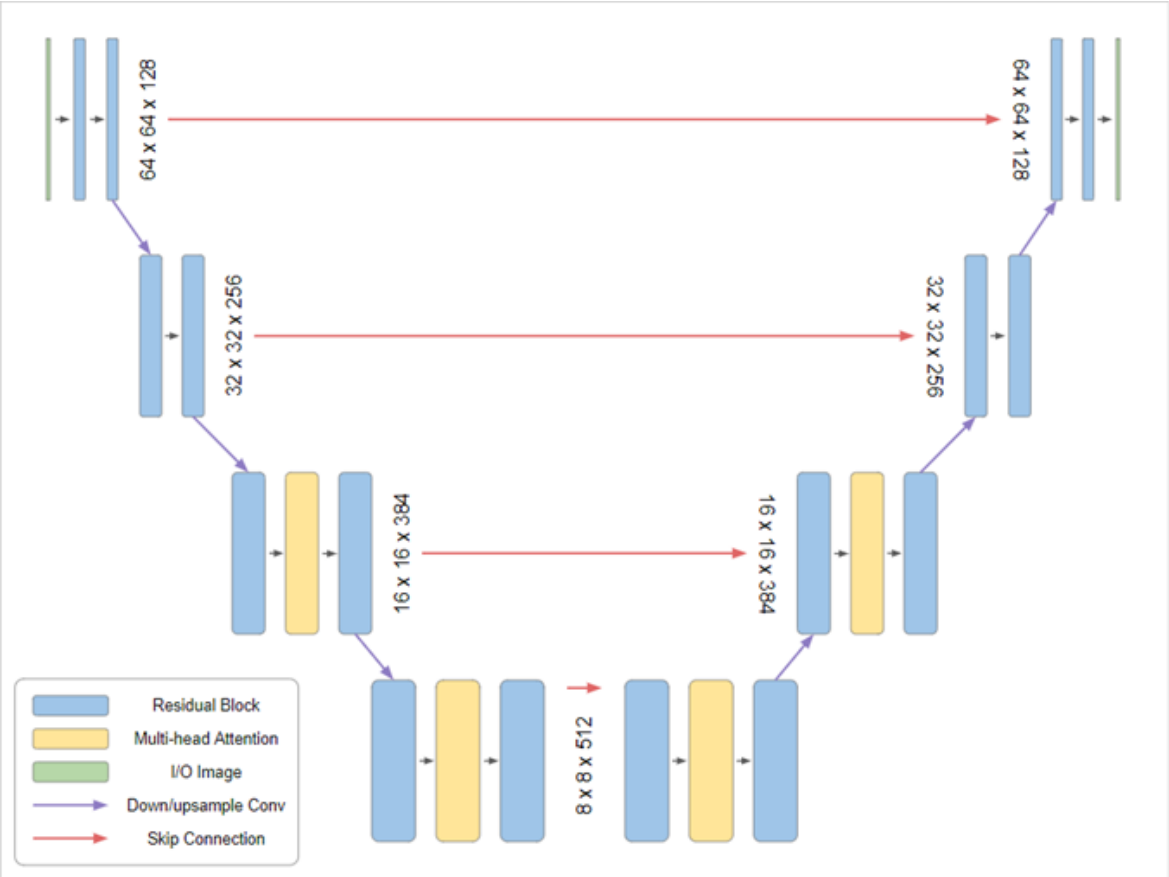
و سپس از یک لایه linear *یک لایه فعالسازی* swish *و سپس یک لایه ی دیگر* linear *عبور میکند این ساختار در کلاس* U-Net *نهایی ساخته شده و به شکل زیر در ساختار آن استفاده میشود:*

```
* `t` has shape `[batch_size]`
        """
# Get time-step embeddings
t = self.time_emb(t)
```

در نهایت t *در بطن شبکه . در دل لایه های* residual *خود را نشان میدهد لایه ی* residual */از دو لایه کانولوشنی تشکیل شده است که* t *در لایه زیر حضور می یابد:*

```
self.time_emb = nn.Linear(time_channels, out_channels)
```

این لایه در بین دو لایه کانولوشنی residual *حضور دارد و در حقیقت باعث درک مدل از زمان میشود که در مرحله از زمان فرایند حذف نویز باید چگونه صورت پذیرد.* این تغییرات به شبکه امکان میدهد که تغییرات زمانی را در مراحل مختلف تخمین نویز در نظر بگیرد.



شکل ۷-ساختار **U-Net**

سوال ۹

در مقاله "High-Resolution Image Synthesis with Latent Diffusion Models"، نویسندگان با استفاده از autoencoderهای denoising و مدل‌های Diffusion ، به حل مشکل مقیاس بالا در مدل Diffusion پرداخته‌اند. برای کاهش پیچیدگی و حفظ جزئیات، از مدل‌های Diffusion روی فضای autoencoder, latentهای پیش‌آموزش‌شده قدرتمند استفاده شده است. با اضافه کردن لایه‌های cross-attention به معماری مدل، مدل‌های Diffusion به generator های قدرتمند و قابل تنظیمی برای انواع ورودی‌ها تبدیل شده‌اند. این مدل‌ها با کاهش نیازهای محاسباتی نسبت به مدل‌های پیکسلی، عملکرد بسیار خوبی در تولید تصاویر با کیفیت بالا دارند.

برای توضیح دقیق تر بیشر ، از آنجایی که این مدل‌ها معمولاً مستقیماً در فضای پیکسل کار می‌کنند، بهینه‌سازی DMهای قدرتمند اغلب صدها روز GPU را مصرف می‌کند و استنتاج به دلیل ارزیابی‌های متوالی سخت و هزینه بر است. برای فعال کردن آموزش DM بر روی منابع محاسباتی محدود در حالی که کیفیت و انعطاف‌پذیری آنها حفظ می‌شود، نویسندگان آنها را در فضای latent، autoencoder از

پیش آموزش دیده اعمال می‌کنند. آموزش مدل‌های Diffusion در چنین حالتی برای اولین بار اجازه می‌دهد تا به یک نقطه تقریباً بهینه بین کاهش پیچیدگی و حفظ جزئیات برسیم و وفاداری بصری را تا حد زیادی افزایش می‌دهد. با وارد کردن cross-attention layers به معماری مدل، مدل‌های انتشار را به generatorهای قدرتمند و انعطاف‌پذیر برای ورودی‌های شرطی عمومی مانند متن و سنتز با وضوح بالا به شیوه‌ای کانولوشنی امکان‌پذیر می‌شود. latent diffusion models (LDM) به وضعیت جدیدی از مدلها برای نقاشی درون تصویر و عملکرد بسیار رقابتی در کارهای مختلف، از جمله تولید تصویر بدون قید و شرط، سنتز صحنه معنایی و وضوح فوق‌العاده دست می‌یابند، در حالی که به طور قابل توجهی نیازهای محاسباتی را در مقایسه با DMهای مبتنی بر پیکسل کاهش می‌دهند.

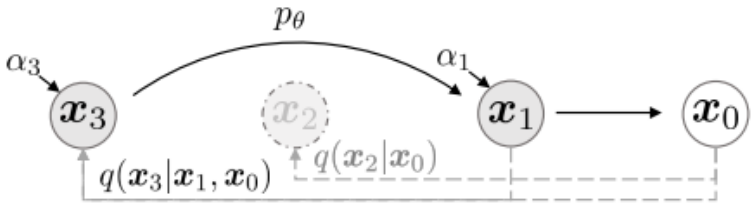
معما ری U-Net برای DMs تغییرات قابل توجهی را نسبت به ساختار پایه دارد. با استفاده از لایه های کانولوشن ۲بعدی، این امکان فراهم میشود تا مدل بر جزئیات معنایی مهم داده Pertinent semantic details تمرکز کند. علاوه بر این، اضافه شدن Reweighted variational bound، بر عناصر معنایی مرتبط را در فضای latent هنگام فرایند مدلسازی generate تاکید میکند.

به طور خلاصه . ا ین تغییرات ساختاری در معمار ی U-Net، به ویژه با تأکید بر کانولوشنهای ۲ بعدی و Pertinent semantic details، توانایی مدل را در تمرکز بر جزئیات معنایی ارتقا میدهد. علاوه بر این، ادغام تکنیک Cross-Attention، امکانات بهتری برای شرایط flexible و کنترل بیشتر بر فرایند سنتز تصویر با استفاده از ورودیهای متنوع را فراهم میکند.

سوال ۱۰

مقاله *DDIM* روشی را برای سرعت بخشیدن به تولید تصویر با کاهش کیفیت تصویر معرفی می کند. این کار را با تعریف مجدد فرآیند انتشار به عنوان یک فرآیند غیرمارکوویی انجام می دهد.

DDIM روشی را برای غیرمارکوین کردن فرآیند پیشنهاد می‌کند که به شما این امکان را می‌دهد تا مراحل فرآیند حذف نویز را *skip* کنید، بدون اینکه نیازی به بازدید از تمام حالت‌های گذشته قبل از وضعیت فعلی نباشد. به عنوان مثال در مسیر *backward* میتوان ۱۰ مرحله را رد کرد و از x_t به x_{t-10} رسید.



شکل ۸-روند بازگشت رو به عقب **ddpm**

در ابتدا فرمول فرایند رو به عقب به شکل زیر تغییر پیدا میکند:

$$x_{t-1} = \sqrt{\alpha_{t-1}} \underbrace{\left(\frac{x_t - \sqrt{1-\alpha_t}\epsilon_\theta^{(t)}(x_t)}{\sqrt{\alpha_t}} \right)}_{\text{“ predicted } x_0\text{”}} + \underbrace{\sqrt{1-\alpha_{t-1}-\sigma_t^2} \cdot \epsilon_\theta^{(t)}(x_t)}_{\text{“direction pointing to } x_t\text{”}} + \underbrace{\sigma_t \epsilon_t}_{\text{random noise}}$$

در نهایت واریانس مدل *diffusion* را به عنوان یک درون یابی بین *DDIM* و *DDPM* با استفاده از فرمول زیر در نظر میگیریم:

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

مدل *diffusion* یک *DDIM* است وقتی $\eta=0$ چون نویز وجود ندارد و یک *DDPM* اصلی وقتی $\eta=1$ است. هر η بین ۰ و ۱ درون یابی بین *DDIM* و *DDPM* است.

سوال ۱۱

مدلهای مبتنی بر *Diffusion* قادرند روابط و وابستگیهای پیچیده در داده را با *Sampling* یک فرایند *Diffusion* به دست آورند. این فرایند شامل گسترش اطلاعات به شکل *Iterative* از پیکسلهای اطراف و در همسایگی آن پیکسل در یک تصویر است.

پس میتوان با تغییر روش آموزش و *Sampling* در مدل‌های *diffusion* تشخیص مناطق مختلف تصویر را انجام داد. برای تولید یک *Semantic Segmentation* برای هر تصویر، میتوان مدل را روی تشخیص دقیق اولیه آموزش داده و تصویر را به عنوان پیش‌زمینه در هر مرحله از فرایند *Sampling* استفاده کرد. با فرایند *Sampling* تصادفی،میتوان توزیعی از ماسک‌های *segmentation* را تولید کرد. این ویژگی اجازه می‌دهد تا *mask* های عدم اطمینان پیکسلی تشخیص را محاسبه کرده و اجازه می‌دهد به یک مجموعه ضمنی از تشخیص‌ها که باعث افزایش عملکرد *segmentation* می‌شود رسید.

روشهای دیگری که بدین منظور میتوان انجام داد تغییر در قسمت *forward* به شکلی که برای *segmentation* مناسب باشد همچنین میتوان از یک مدل *diffusion* از پیش آموزش دیده استفاده کرد و با آموزش دوباره آن به *segmentation* مد نظر در مسیر رو به عقب رسید. در واقع میتوان *diffusion* را به این شکل استفاده کرد که هر تصویر به همراه *label* را دریافت کند بصورتی که این لیبل نشان دهنده یک منطقه خاص مثلا سلول های سرطانی به شکل ماسک می باشد و در فرایند رو به عقب سعی کند به جای تصویر اصلی تصویر سگمنت شده را بازسازی کند در واقع به جای حذف نویز به دنبال ناحیه های غیر مشابه با ماسک باشد.

سوال ۱۲

در این قسمت به اضافه کردن نویز به تصاویر مان می پردازیم. برای این کار از دو روش استفاده شده است که روش دوم به مراتب به صرفه تر از روش اول می باشد. در روش اول مطابق با رابطه زیر نویز را به صورت مرحله به مرحله به شبکه مان اضافه می کنیم.

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

همان طور که در این رابطه مشاهده می شود کار اضافه کردن نویز به صورت مرحله به مرحله می باشد .لازم به ذکر است که این روش برای تعداد step های زیاد خیلی عملیات زمانبری خواهد شد. کد مربوط به این روش در زیر آورده شده است.

```
n_steps = 1000
beta = torch.linspace(0.0001, 0.04, n_steps)

def q_xt_xtminus1(xtm1, t):
    mean =  torch.sqrt(gather(1. - beta, t)) * xtm1 # √(1-βt)*xtm1
    var = torch.sqrt(gather(beta, t)) # βt I
    eps = torch.randn_like(xtm1)
    q_xt_xtminus1 = mean + var * eps
    return q_xt_xtminus1
```

نمونه های تولیدی با فاصله ی ۲۰۰ از یکدیگر:

شکل ۹-تصاویر نویزی شده در ۵ زمان مختلف

سوال ۱۳

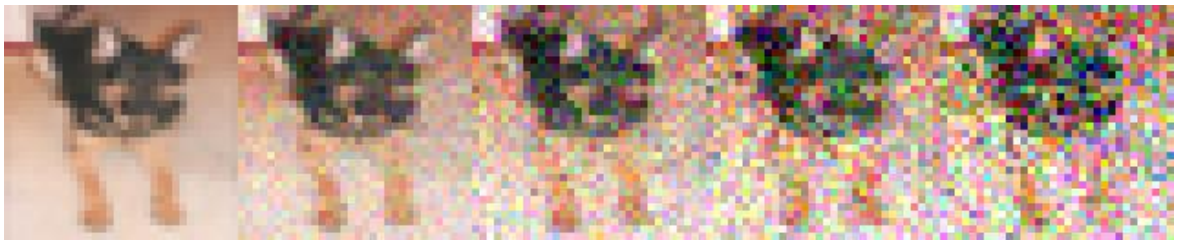
بدی روش اول این می باشد که برای نویزی کردن یک تصویر در لحظه t باید تصویر لحظه t-1 را داشته باشیم که این کار کمی محاسبات را بیشتر می کند. بنابراین برای حل این مشکل سراغ روش دوم می رویم که براساس reparametrization metric کار می کند و با استفاده از آن تصویر نویزی لحظه t در هر لحظه به صورت یک فرم بسته از تصویر اولیه و بدون نویز مان بدست می آید. این روش به صورت رابطه زیر می باشد:

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

حال این رابطه را در قالب کد در آورده ایم که کد مربوطه در زیر آورده شده است.

```
n_steps = 1000
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):
    mean =  torch.sqrt(gather(alpha_bar, t)) * x0
    var = torch.sqrt(1-gather(alpha_bar, t)) # (1-alpha_bar)
    eps = torch.randn_like(x0)
    q_xt_x0 = mean + var * eps
    return q_xt_x0
```

تصاویر حاصل به صورت زیر است:



شکل ۱۰-تصویر بدست آمده در لحظه **t** از تصویر اولیه

سوال ۱۴

در ابتدا با استفاده از روش دوم اضافه کردن نویز یعنی همان روش $q(x_t|x_0)$ کار اضافه کردن نویز را انجام می دهیم ولی با این تفاوت که این بار دو تا خروجی می گیریم یکی همان تصویر نویزی شده در لحظه t را می گیریم و خروجی دیگر مان همان نویزمان می باشد که برای محاسبه تابع لاس به کار می رود بنابراین این تابع به صورت کد زیر تعریف می شود.

```
# Set up some parameters
n_steps = 1000
beta = torch.linspace(0.0001, 0.04, n_steps).cuda()
alpha = 1. - beta
alpha_bar = torch.cumprod(alpha, dim=0)

# Modified to return the noise itself as well
def q_xt_x0(x0, t):
    mean =  torch.sqrt(gather(alpha_bar, t)) * x0
    var = torch.sqrt(1-gather(alpha_bar, t))
    eps = torch.randn_like(x0).to(x0.device)
    q_xt_x0 = mean + var * eps
    return q_xt_x0, eps
```

هایپر پارامتر ها و اپتیمایز انتخاب شده در زیر آورده شده است:

```
# Training params
batch_size = 700# Lower this if hitting memory issues
lr = 2e-4 # Explore this - might want it lower when training on the
full dataset


losses = [] # Store losses for later plotting
optim = torch.optim.AdamW(unet.parameters(), lr=lr)
```

خطا را به شکل زیر مینویسیم:

```
loss = F.mse_loss(n.float(), pred_n)
```

کد نهایی برای آموزش شبکه در ۶۰ ایپاک:

```
val_dataset = cifar10['test']
dataset = cifar10['train']

# Set the model to training mode
unet.train()

# Number of epochs
num_epochs = 60

# Initialize a list to store training losses

epoch_train_loss = []
epoch_val_loss = []
# Training loop
for epoch in range(num_epochs):
    train_losses = []
    for i in tqdm(range(0, len(dataset) - batch_size, batch_size)):
        images = [dataset[j]['img'] for j in range(i, i +
batch_size)]
```

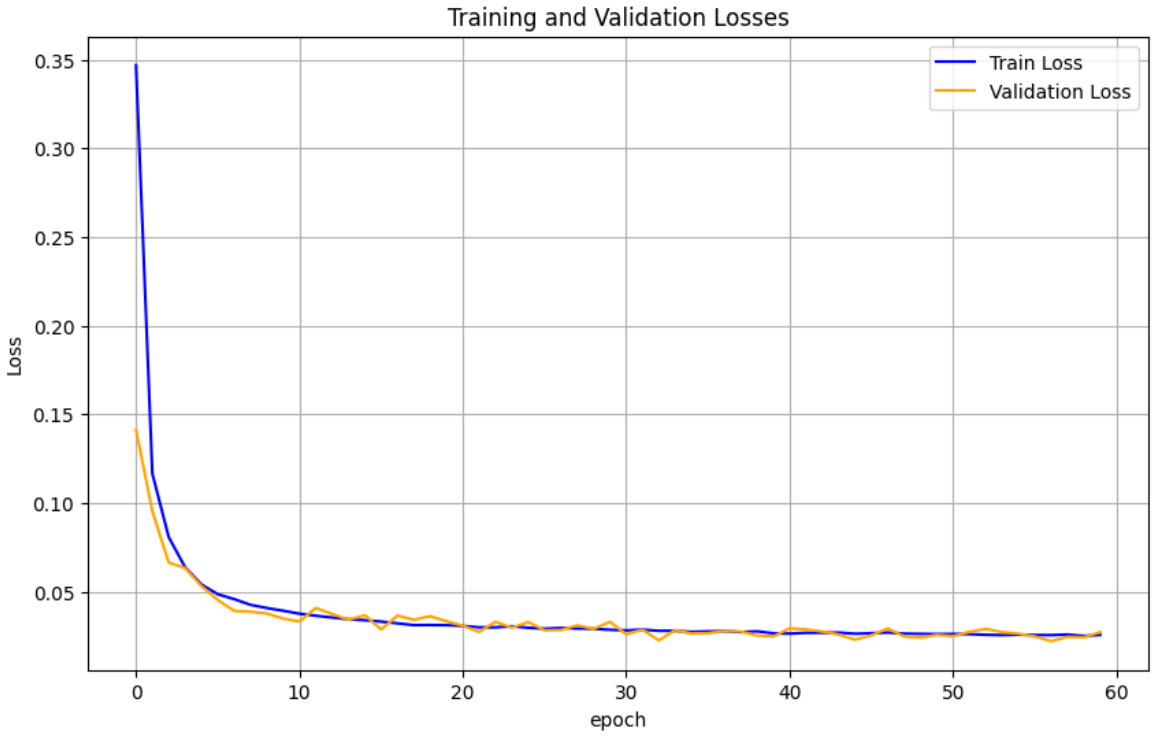
```
        t_ims = [img_to_tensor(each_image).cuda() for each_image in
images]
        x0 = torch.cat(t_ims)
        t = torch.randint(0, n_steps, (batch_size,),
dtype=torch.long).cuda()
        xt, n = q_xt_x0(x0, t)
        pred_n = unet(xt.float(), t)
        loss = F.mse_loss(n.float(), pred_n)
        train_losses.append(loss.item())
        optim.zero_grad()
        loss.backward()
        optim.step()
    # Validation loop
    unet.eval()
    val_losses = []
    with torch.no_grad():
        for k in tqdm(range(0,1)):
            images = [val_dataset[w]['img'] for w in range(k, k +
batch_size)]
            t_ims = [img_to_tensor(each_image).cuda() for each_image
in images]
            x0 = torch.cat(t_ims)
            t = torch.randint(0, n_steps, (batch_size,),
dtype=torch.long).cuda()
            xt, n = q_xt_x0(x0, t)
            pred_n = unet(xt.float(), t)
            loss = F.mse_loss(n.float(), pred_n)
            val_losses.append(loss.item())

    # Calculate and print mean training and validation losses
    mean_train_loss = sum(train_losses) / len(train_losses)
    epoch_train_loss.append(mean_train_loss)
    mean_val_loss = sum(val_losses) / len(val_losses)
    epoch_val_loss.append(mean_val_loss)
    print(f"Epoch {epoch + 1}/{num_epochs}, Train Loss:
{mean_train_loss}, Val Loss: {mean_val_loss}")
    # Set the model back to training mode
    unet.train()
```

روند آموزش شبکه در ۵ ایپاک نهایی:

```
Epoch 55/60, Train Loss: 0.02568612670079923, Val Loss:
0.026225611567497253
100%
71/71 [02:09<00:00, 1.83s/it]
100%
1/1 [00:01<00:00, 1.49s/it]
Epoch 56/60, Train Loss: 0.02550133776811647, Val Loss:
0.024696597829461098
```

```
100%
71/71 [02:09<00:00, 1.84s/it]
100%
1/1 [00:01<00:00, 1.48s/it]
Epoch 57/60, Train Loss: 0.025436740893293435, Val Loss:
0.02203209325671196
100%
71/71 [02:09<00:00, 1.83s/it]
100%
1/1 [00:01<00:00, 1.53s/it]
Epoch 58/60, Train Loss: 0.02578811977111118, Val Loss:
0.02444308064877987
100%
71/71 [02:10<00:00, 1.84s/it]
100%
1/1 [00:01<00:00, 1.49s/it]
Epoch 59/60, Train Loss: 0.02496385870789978, Val Loss:
0.024212423712015152
100%
71/71 [02:10<00:00, 1.85s/it]
100%
1/1 [00:01<00:00, 1.47s/it]
Epoch 60/60, Train Loss: 0.02553077362162966, Val Loss:
0.02711590565741062
```

در نهایت نمودار لاس برای هر ایپاک برای آموزش و ارزیابی به شکل زیر رسم میشود تعداد ۷۱ استپ برای آموزش و یک استپ برای ارزیابی داریم:



شکل ۱۱-نمودار لاس ارزیابی و آموزش به ازای ۶۰ ایپاک

سوال ۱۵

در این بخش نیز به تولید داده ها می پردازیم برای این کار ابتدا تابع $P_\theta(x_{t-1}| x_t)$ را پیاده سازی می کنیم کد مربوط به این تابع در شکل زیر آورده شده است. در واقع این تابع سعی می کند در هر مرحله تصویر نویزی را بگیرد و مقدار نویزش را یک step کم تر کند.

```
def p_xt(xt, noise, t):
    alpha_t = gather(alpha, t)
    alpha_bar_t = gather(alpha_bar, t)

    c = noise*(1-alpha_t)/torch.sqrt(1-alpha_bar_t)
    mean = 1 / torch.sqrt(alpha_t) * (xt - c)
    var = torch.sqrt(gather(beta, t))
    eps = torch.randn(xt.shape, device=xt.device)
    x_tm1 = mean + var * eps

    return x_tm1
```

حال برای تولید هر داده ابتدا یک نمونه برداری از فضای نویزی نرمال خواهیم کرد و این داده نویزی را به تابع مان می دهیم و سعی می کنیم به اندازه ۱۰۰۰ تا step این کار را تکرار کنیم تا در نهایت داده نویزی اولیه مان در ۱۰۰۰ مرحله متوالی denoise شود. شکل زیر، صد تصویر تولید شده توسط این الگوریتم را نشان می دهد. همان طور که مشاهده می شود تصاویر تولیدی تقریبا نمودی از تصاویر اولیه را دارند برای گرفتن تصاویر بهتر، بهتر است که تعداد ایپاک ها و تعداد step ها را بیشتر کنیم. اما برای این کار محدودیت رم داریم.

شکل ۱۲-۱۰۰ تصویر بدست آمده از شبکه

سوال ۱۶

برای محاسبه معیار fid در ابتدا ۵۰۰ تصویر واقعی و مصنوعی تولید میکنیم و آنها را در فایل های خاصی ذخیره میکنیم در زیر کدهای مربوز به این دو بخش آورده شده است:

تصاویر مصنوعی:

```
for k in range(500):
    x = torch.randn(1, 3, 32, 32).to(device) # Start with random
noise
    for i in range(n_steps):
      t = torch.tensor(n_steps-i-1, dtype=torch.long).to(device)
      with torch.no_grad():
        pred_noise = unet(x.float(), t.unsqueeze(0)).to(device)
        x = p_xt(x, pred_noise, t.unsqueeze(0))
    # TODO: save the generated sample (x) in a directory
```

```
    # See this link:
https://pytorch.org/vision/stable/generated/torchvision.utils.save_im
age.html
    # Be sure to assign a different name to each image!
    if not os.path.exists(f"/content/fake_images"):
        os.makedirs(f"/content/fake_images")
    save_image( x , f"/content/fake_images/fake{k}.png")
```

تصاویر واقعی:

```
#save real images from testset to a directory and finally move it to
google drive manually
from torchvision.datasets import CIFAR10
from torchvision.transforms import ToTensor
testset = CIFAR10(root='testset/', download=True, train=False,
transform=ToTensor())
testloader=torch.utils.data.DataLoader(testset, batch_size=500,
shuffle=True)
for data in testloader:
  real_images = data[0]
  break
if not os.path.exists(f"/content/real_images"):
        os.makedirs(f"/content/real_images")
for k in range(500):
  save_image( real_images[k] , f"/content/real_images/real{k}.png")
```

در ادامه داده های موجود در این دو فولدر را فراخوانی کرده و به تابع fid مان می دهیم. لازم به ذکر است که تابع fid مان را از کتابخانه torchmetrics.image.fid و با نام FrechetInceptionDistance فراخوانی کرده ایم. در ضمن تعداد فیچر های لازم برای تصاویر را برابر ۶۴ در نظر گرفته ایم. کد مربوط به این بخش و همچنین مقدار خروجی fid برای این تصاویر در زیر آورده شده است.

```
# read real and fake images from google derive
fake_images = torch.zeros(500,3,32,32)
real_images = torch.zeros(500,3,32,32)
for i in range(500):
  fake_images[i]  =
torchvision.io.read_image(f'/content/fake_images/fake{i}.png')
  real_images[i]  =
torchvision.io.read_image(f'/content/real_images/real{i}.png')


fid = FrechetInceptionDistance(feature=192)
# generate two slightly overlapping image intensity distributions

fid.update(real_images.type(torch.uint8), real=True)
fid.update(fake_images.type(torch.uint8), real=False)
```

```
fid.compute()
```

همان طور که مشاهده می شود مقدار fid برابر ۴۴.۲۰۸۵ شده است. این مقدار احتمالا با افزایش نمونه ها از ۵۰۰ به ۱۰۰۰ به حدود ۸۰ اقزایش می یابد که با توجه به خروجی های تولید شده توسط مدل قبل قبول است.