به نام خدا

تمرین سری سوم مبانی بینایی

معصومه پاسبانی 99243022

سوال اول)

الف) اتو انکودر استاندارد: فضای نهان را به صورت مستقیم و قطعی یاد می‌گیرد. این فضای نهان معمولاً یک بردار مشخص است که نگاشتی از داده‌های ورودی استVAEفضای نهان را به صورت احتمالی مدل می‌کند. به جای یک بردار مشخص، یک توزیع احتمال (مانند توزیع نرمال) برای فضای نهان یاد می‌گیرد.

اتو انکودر استاندارد هدفش فشرده‌سازی داده‌ها به فضای نهان و بازسازی دقیق داده‌های ورودی است.علاوه بر بازسازی داده‌ها، هدف آن یادگیری یک توزیع احتمال قابل نمونه‌برداری در فضای نهان است که امکان تولید داده‌های جدید مشابه داده‌های ورودی را فراهم کند.درVAE ، نگاشت به فضای نهان به جای یک بردار قطعی، با دو پارامتر (میانگین و انحراف معیار) مدل می‌شود که توزیع احتمال گوسی را تعریف می‌کند.

با یادگیری یک توزیع احتمال در فضای نهان، می‌توان نمونه‌های جدید از این توزیع گرفت و داده‌های جدیدی مشابه داده‌های آموزشی تولید کرد. استفاده از توزیع احتمال (مانند توزیع نرمال) باعث می‌شود فضای نهان منظم‌تر شود و نقاط نزدیک در این فضا به داده‌های مشابهی نگاشت شوند.

توزیع احتمالی در فضای نهان امکان انتقال دانش به مسائل یا داده‌های دیگر را فراهم می‌کند.درVAE ، استفاده از توزیع احتمال باعث می‌شود هم بازسازی داده‌ها و هم قابلیت تولید داده‌های جدید با کیفیت بالا ممکن شود.

در نتیجه، تفاوت اصلی در این است که VAE به‌جای بازسازی صرف، به تولید داده‌های جدید نیز تمرکز دارد، که این هدف با یادگیری توزیع احتمال در فضای نهان محقق می‌شود.

ب) تابع خطا در VAEشامل دو جزء اصلی است که با هم ترکیب می‌شوند تا عملکرد مدل بهینه شود:

1. **خطای بازسازی:**(Reconstruction Loss)

   این بخش از تابع خطا، تفاوت بین داده‌های اصلی و داده‌های بازسازی‌شده توسط دیکدر را محاسبه می‌کند. این خطا تعیین می‌کند که مدل چقدر توانسته اطلاعات ورودی را حفظ و به‌درستی بازسازی کند.

   - معیارهای رایج:
     - اگر داده‌ها گسسته باشند (مانند تصاویر باینری): از **آنتروپی متقاطع** استفاده می‌شود.
     - اگر داده‌ها پیوسته باشند: از **میانگین مربع خطا** (MSE) استفاده می‌شود.

   هدف: حفظ جزئیات ورودی با دقت بالا.

2. **واگرایی کولبک-لیبلر:**(KL Divergence)

   این بخش، فاصله بین توزیع فضای نهان (که از داده‌ها یاد گرفته شده) و توزیع نرمال استاندارد را اندازه‌گیری می‌کند.

هدف این است که فضای نهان به‌صورت منظم و شبیه به یک توزیع گوسی استاندارد باشد. این امر باعث می‌شود نمونه‌گیری از فضای نهان ساده‌تر شود و مدل بتواند داده‌های جدید تولید کند.

- این بخش با فرمول KL-Divergence به مدل فشار می‌آورد که توزیع فضای نهان را به سمت یک توزیع مشخص (معمولاً گوسی) سوق دهد.

اهمیت این دو بخش:

- **بازسازی دقیق داده‌ها :**خطای بازسازی تضمین می‌کند که داده‌های ورودی در هنگام عبور از فضای نهان دچار تغییرات شدید و غیرقابل‌بازسازی نشوند.
- **تضمین ساختار فضای نهان :**واگرایی KL به مدل کمک می‌کند تا فضای نهانی ایجاد کند که مرتب و قابل نمونه‌برداری باشد، که برای تولید داده‌های جدید ضروری است.
- **تعادل بین بازسازی و تولید:**
  - اگر فقط روی بازسازی تمرکز شود، مدل به سمت حفظ جزئیات داده‌های ورودی متمایل می‌شود و از قابلیت تولید نمونه‌های جدید باز می‌ماند.
  - اگر فقط به واگرایی KL توجه شود، بازسازی دقیق داده‌ها امکان‌پذیر نخواهد بود.

---

encoderداده‌های ورودی را دریافت کرده و آن‌ها را به یک نمایش فشرده در فضایی با ابعاد کمتر به نام فضای نهان نگاشت می‌کند.

لایه اول:(fc1)

یک لایه کاملاً متصل که داده‌های ورودی را از ابعاد 784 به 256 کاهش می‌دهد.از تابع فعال‌سازی ReLU استفاده می‌شود تا ویژگی‌های غیرخطی داده‌ها را استخراج کند.

میانگین (fc_mean)

این لایه داده‌های خروجی از لایه اول را به برداری با ابعاد فضای نهان (پیش‌فرض 2) تبدیل می‌کند.این بردار نمایش فشرده‌شده اصلی است.

واریانس:(fc_log_variance)

این لایه خروجی دیگری ایجاد می‌کند که میزان پراکندگی یا عدم قطعیت داده‌ها را در فضای نهان مشخص می‌کند.

ابتدا داده‌های ورودی از لایه اول عبور کرده و پردازش می‌شوند.سپس دو خروجی تولید می‌شود:

بردار میانگین: اطلاعات اصلی داده‌ها.

بردار لگاریتم واریانس: میزان پراکندگی داده‌ها.

decoderوظیفه دارد داده‌های فشرده‌شده از فضای نهان را دریافت کرده و آن‌ها را به شکل اولیه (مانند تصاویر ورودی) بازسازی کند.

لایه اول:(fc1)

این لایه بردار نهان (با ابعاد پیش‌فرض 2) را به یک بردار بزرگ‌تر (256 بعدی) نگاشت می‌کند.از تابع فعال‌سازی ReLU برای پردازش استفاده می‌شود.

لایه دوم:(fc2)

این لایه بردار را به ابعاد اصلی داده‌ها (784 بعد) بازمی‌گرداند.از تابع سیگموید استفاده می‌شود تا مقادیر خروجی در بازه [0, 1] قرار گیرند) مطابق مقادیر پیکسلی تصاویر.(MNIST

بردار نهان از طریق لایه‌های رمزگشا عبور داده می‌شود.خروجی نهایی، بازسازی‌ای از تصویر اصلی است که با نسخه ورودی تطابق دارد.

```
class Encoder(nn.Module):
    def __init__(self, input_dim=784, hidden_dim=256, latent_dim=2):
        super(Encoder, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc_mean = nn.Linear(hidden_dim, latent_dim)
        self.fc_log_variance = nn.Linear(hidden_dim, latent_dim)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        mean_vector = self.fc_mean(x)
        log_variance = self.fc_log_variance(x)
        return mean_vector, log_variance
```

```
class Decoder(nn.Module):
    def __init__(self, latent_dim=2, hidden_dim=256, output_dim=784):
        super(Decoder, self).__init__()
        self.fc1 = nn.Linear(latent_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, latent_vector):
        latent_vector = torch.relu(self.fc1(latent_vector))
        latent_vector = torch.sigmoid(self.fc2(latent_vector))
        return latent_vector
```

این کلاس مدل اصلی شبکه **VAE**را تعریف می‌کند و از دو بخش **رمزگذار (Encoder)** و **رمزگشا (Decoder)** استفاده می‌کند. همچنین، شامل یک مرحله کلیدی به نام **بازنمونه‌گیری (Reparameterization)** است که بخشی از فرآیند تولید داده‌های جدید یا بازسازی داده‌های ورودی می‌باشد.

Init : در این قسمت، مدل VAE ایجاد می‌شود و شامل موارد زیر است:

رمزگذار :(Encoder) این بخش ورودی را به یک فضای فشرده (فضای نهان) نگاشت می‌کند و بردارهای میانگین و واریانس را تولید می‌کند.

رمزگشا :(Decoder) این بخش داده‌های فشرده‌شده را گرفته و بازسازی تصویر اصلی را انجام می‌دهد.

Reparameterization

فرآیند بازنمونه‌گیری یکی از ویژگی‌های کلیدی VAE است. این کار برای اطمینان از قابلیت مشتق‌پذیری مدل و امکان آموزش آن با پس‌انتشار خطا (Backpropagation) انجام می‌شود.

بردار میانگین :(mean_vector) ویژگی‌های اصلی داده.

بردار واریانس (:(log_varianceعدم قطعیت داده.

forwardتابع اصلی مدل که داده‌های ورودی را پردازش می‌کند و سه خروجی تولید می‌کند:

تصویر بازسازی‌شده :(reconstructed_image) داده‌ای که رمزگشا تولید کرده است.

بردار میانگین :(mean_vector) ویژگی‌های فشرده داده‌ها در فضای نهان.

بردار واریانس :(log_variance) میزان پراکندگی داده‌ها در فضای نهان.

داده‌های ورودی به رمزگذار داده می‌شود تا بردارهای میانگین و واریانس تولید شوند. بردار میانگین و واریانس به یک بردار فشرده (بردار نهان) تبدیل می‌شوند و بردار نهان به رمزگشا داده می‌شود تا داده اصلی بازسازی شود.

```
class VAE(nn.Module):
    def __init__(self, input_dim=784, hidden_dim=256, latent_dim=2):
        super(VAE, self).__init__()
        self.encoder = Encoder(input_dim, hidden_dim, latent_dim)
        self.decoder = Decoder(latent_dim, hidden_dim, input_dim)

    def reparameterize(self, mean_vector, log_variance):
        std = torch.exp(0.5 * log_variance)
        eps = torch.randn_like(std)
        return mean_vector + eps * std

    def forward(self, x):
        mean_vector, log_variance = self.encoder(x)
        latent_vector = self.reparameterize(mean_vector, log_variance)
        reconstructed_image = self.decoder(latent_vector)
        return reconstructed_image, mean_vector, log_variance

def loss_function(reconstructed_image, original_image, mean_vector, log
```

خطای بازسازی :(Reconstruction Loss) این بخش میزان اشتباه مدل در بازسازی ورودی‌ها را محاسبه می‌کند. به این معنی که تصویر ورودی را به تصویر جدیدی تبدیل می‌کنیم و میزان تفاوت این دو تصویر را اندازه‌گیری می‌کنیم. برای این کار از آنتروپی متقاطع باینری استفاده می‌شود، که به‌طور خاص برای مقایسه تصاویر باینری مناسب است. سپس این مقدار با تعداد نمونه‌ها تقسیم می‌شود تا میانگین خطا به دست آید.

واگرایی :(KL Divergence) KLاین بخش از مدل برای مجازات کردن مدل به دلیل فاصله زیاد میان توزیع احتمالاتی که مدل می‌آموزد و توزیع نرمال استاندارد (که معمولاً توزیع دلخواه است) استفاده می‌شود. این بخش از زیان به مدل می‌گوید که توزیع‌های نهانش باید شبیه به توزیع نرمال باشند.

سپس داده‌های MNIST آماده می‌شوند:

تصاویر MNIST به تانسور تبدیل می‌شوند تا مدل بتواند آن‌ها را پردازش کند.داده‌ها از مجموعه MNIST دانلود و آماده می‌شوند.داده‌ها به گروه‌هایی از ۱۲۸ تصویر تقسیم می‌شوند که مدل به‌طور همزمان این گروه‌ها را پردازش کند.

این مراحل به مدل کمک می‌کند تا داده‌ها را برای آموزش به‌طور مؤثر استفاده کند.

```
def loss_function(reconstructed_image, original_image, mean_vector, log_variance, beta=0.1):
    reconstruction_loss = nn.functional.binary_cross_entropy(reconstructed_image, original_image, reduction='s
    kl_divergence = -0.5 * torch.sum(1 + log_variance - mean_vector.pow(2) - log_variance.exp()) / original_im
    return reconstruction_loss + beta * kl_divergence

transform = transforms.Compose([transforms.ToTensor()])
dataset = datasets.MNIST(root='./data', train=True, transform=transform, download=True)
data_loader = DataLoader(dataset, batch_size=128, shuffle=True)
```

در این بخش کد، مدل VAE برای آموزش آماده می‌شود و از Adam Optimizer برای بهینه‌سازی پارامترهای آن استفاده می‌شود. مراحل به‌طور خلاصه به این صورت است:

مدل VAE ایجاد و به دستگاه یا منتقل می‌شود.یک optimizer (Adam) برای به‌روزرسانی پارامترهای مدل تعریف می‌شود.

مدل در حالت train قرار می‌گیرد تا برای آموزش آماده شود.در هرepoch ، داده‌ها به صورت مینی‌بچ پردازش می‌شوند.تصاویر ورودی به تانسور مسطح تبدیل می‌شوند و به مدل وارد می‌شوند.مدل خروجی (تصویر بازسازی‌شده، میانگین و واریانس لاگ) تولید می‌کند.زیان (loss) بین تصویر بازسازی‌شده و تصویر اصلی محاسبه می‌شود.

backpropagationانجام می‌شود و سپس optimizer پارامترهای مدل را به‌روزرسانی می‌کند epoch_loss .در هر دوره به‌روزرسانی شده و چاپ می‌شود.

در انتهای هر epoch، زیان متوسط برای ارزیابی عملکرد مدل گزارش می‌شود.

```
variational_autoencoder = VAE().to(device)
adam_optimizer = optim.Adam(variational_autoencoder.parameters(), lr=1e-3)
```

+ Code + Text

```
num_epochs = 20
variational_autoencoder.train()
for epoch in range(num_epochs):
    epoch_loss = 0
    for batch_images, _ in data_loader:
        batch_images = batch_images.view(batch_images.size(0), -1).to(device)

        adam_optimizer.zero_grad()
        reconstructed_image, mean_vector, log_variance = variational_autoencoder(batch_images)
        loss = loss_function(reconstructed_image, batch_images, mean_vector, log_variance)
        loss.backward()
        adam_optimizer.step()

        epoch_loss += loss.item()

    print(f"Epoch [{epoch + 1}/{num_epochs}], Loss: {epoch_loss / len(dataset):.4f}")
```

```
Epoch [1/20], Loss: 1.4783
```

تابع plot_reconstructed_images برای نمایش تصاویری که مدل بازسازی کرده، تعریف می‌شود.

ابتدا تصاویر اصلی و تصاویر بازسازی‌شده به صورت تانسور به ابعاد 28 28x تغییر اندازه داده می‌شوند.سپس تصاویر به فرمت numpy تبدیل می‌شوند تا برای رسم آماده شوند.یک subplot با ۲ ردیف و ۱۰ ستون برای نمایش تصاویر اصلی و بازسازی‌شده ایجاد می‌شود.در هر ستون، تصویر اصلی در ردیف اول و تصویر بازسازی‌شده در ردیف دوم قرار می‌گیرد.تصاویر با استفاده از cmap='gray' به صورت سیاه‌وسفید نمایش داده می‌شوند.پس از پایان حلقه، تابع plt.show() برای نمایش تصاویر فراخوانی می‌شود.مدل به حالت ارزیابی تغییر وضعیت می‌دهد تا از dropout و batch normalization در حین آزمایش جلوگیری شود.تعدادی تصویر آزمایشی از داده‌ها گرفته می‌شود و به مدل وارد می‌شود.تصاویر وارد شده به مدل، به فرمت مسطح تبدیل می‌شوند و به دستگاه منتقل می‌شوند.با استفاده از torch.no_grad() محاسباتی که نیاز به گرادیان ندارند، انجام می‌شود تا مصرف حافظه کاهش یابد.در نهایت، تصاویر بازسازی‌شده توسط مدل به تابع plot_reconstructed_images ارسال می‌شوند و نمایش داده می‌شوند.

```
def plot_reconstructed_images(original_images, generated_images, num_imag
    original_images = original_images[:num_images].view(-1, 28, 28).cpu().detach().numpy()
    generated_images = generated_images[:num_images].view(-1, 28, 28).cpu().detach().numpy()

    fig, axes = plt.subplots(2, num_images, figsize=(15, 4))
    for i in range(num_images):
        axes[0, i].imshow(original_images[i], cmap='gray')
        axes[0, i].axis('off')
        axes[1, i].imshow(generated_images[i], cmap='gray')
        axes[1, i].axis('off')
    plt.show()

variational_autoencoder.eval()
test_images, _ = next(iter(data_loader))
test_images = test_images.view(test_images.size(0), -1).to(device)
with torch.no_grad():
    reconstructed_images, _, _ = variational_autoencoder(test_images)

plot_reconstructed_images(test_images, reconstructed_images)
```

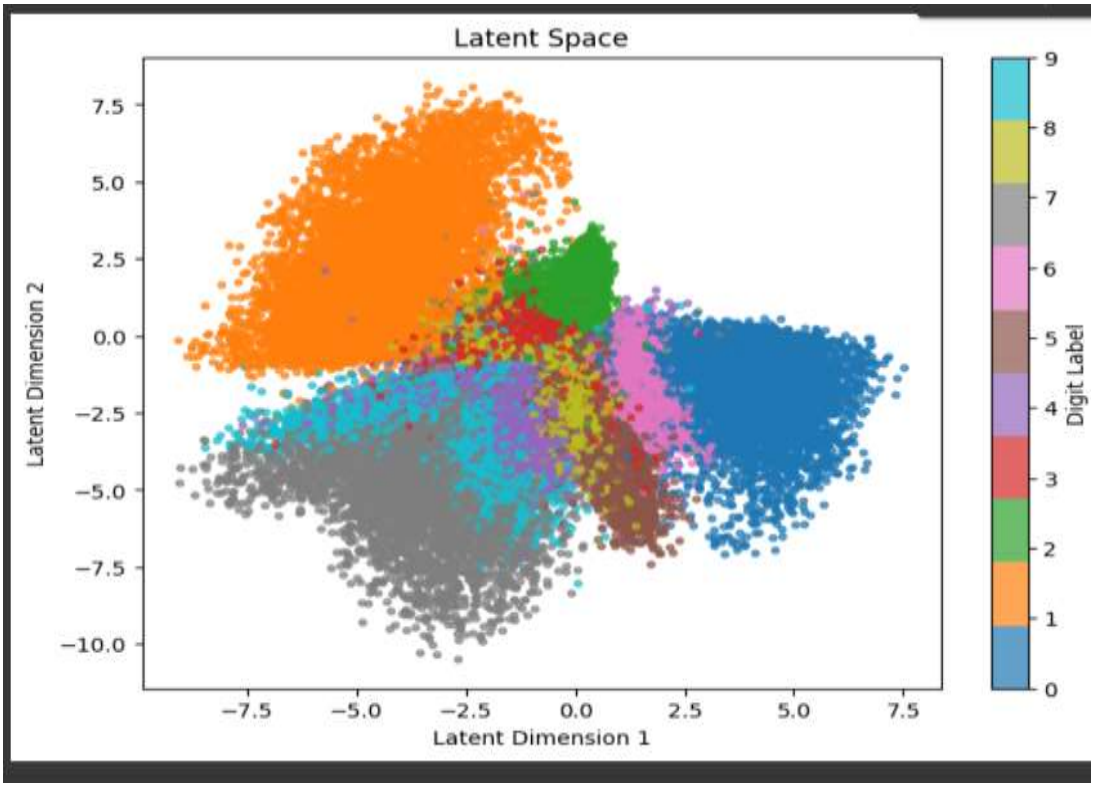
تابع plot_latent_space برای تجسم فضای نهان مدل تعریف می‌شود.مدل در حالت eval قرار می‌گیرد تا از dropout و batch normalizationجلوگیری شود.دو لیست خالی برای ذخیره بردارهای نهان (latent_vectors) و برچسب‌های مربوط به تصاویر (labels_list) ایجاد می‌شود.

درون یک حلقه برای هر دسته از داده‌ها درdata_loader ، تصاویر به صورت مسطح به مدل وارد می‌شوند.مدل از طریق encoderمیانگین و واریانس لاگ را محاسبه می‌کند.سپس با استفاده ازreparameterize ، بردار نهان تولید می‌شود.

بردارهای نهان و برچسب‌ها به ترتیب به لیست‌های latent_vectors و labels_list افزوده می‌شوند.پس از پایان حلقه، بردارهای نهان و برچسب‌ها به صورت یک تانسور واحد ترکیب می‌شوند.سپس با استفاده از matplotlib یک scatter plot از بردارهای نهان در دو بعد اول ایجاد می‌شود.

در این نمودار، رنگ هر نقطه بر اساس برچسب دیجیتال (اعداد 0 تا 9) تعیین می‌شود.نمودار به گونه‌ای طراحی می‌شود که ابعاد نهان را به‌وضوح نشان دهد و از cmap='tab10' برای رنگ‌آمیزی استفاده می‌کند.در نهایت، این نمودار به کمک plt.show() نمایش داده می‌شود.

```python
def plot_latent_space(variational_autoencoder, data_loader):
    variational_autoencoder.eval()
    latent_vectors, labels_list = [], []

    with torch.no_grad():
        for batch_images, labels in data_loader:
            batch_images = batch_images.view(batch_images.size(0), -1).to(device)
            mean_vector, log_variance = variational_autoencoder.encoder(batch_images)
            latent_vector = variational_autoencoder.reparameterize(mean_vector, log_variance)
            latent_vectors.append(latent_vector.cpu())
            labels_list.append(labels)

    latent_vectors = torch.cat(latent_vectors)
    labels = torch.cat(labels_list)

    plt.figure(figsize=(8, 6))
    scatter = plt.scatter(latent_vectors[:, 0], latent_vectors[:, 1], c=labels, cmap='tab10', alpha=0.7, s=10)
    plt.colorbar(scatter, label='Digit Label')
    plt.xlabel("Latent Dimension 1")
    plt.ylabel("Latent Dimension 2")
    plt.title("Latent Space")
    plt.show()

plot_latent_space(variational_autoencoder, data_loader)
```



تابع train_vae برای آموزش مدل VAE با latent_dim مشخص ایجاد می‌شود.مدل VAE با ابعاد نهان داده‌شده و به دستگاه منتقل می‌شود.

Adam Optimizerبرای به‌روزرسانی پارامترهای مدل با نرخ یادگیری 0.001 تعریف می‌شود.مدل در حالت train قرار می‌گیرد تا به‌روزرسانی وزن‌ها در حین آموزش انجام شود.در هرepoch ، داده‌ها به صورت مینی‌بچ پردازش می‌شوند و برای هر دسته، زیان محاسبه می‌شود.پس از محاسبه گرادیان‌ها با()loss.backward ، ()optimizer.stepبرای به‌روزرسانی وزن‌ها انجام می‌شود.در پایان هرepoch ، زیان متوسط محاسبه شده و نمایش داده می‌شود.پس از اتمام آموزش، مدل آموزش‌دیده باز می‌گردد.

```python
def train_vae(latent_dim, num_epochs=20):
    variational_autoencoder = VAE(latent_dim=latent_dim).to(device)
    adam_optimizer = optim.Adam(variational_autoencoder.parameters(), lr=1e-3)
    variational_autoencoder.train()

    for epoch in range(num_epochs):
        epoch_loss = 0
        for batch_images, _ in data_loader:
            batch_images = batch_images.view(batch_images.size(0), -1).to(device)

            adam_optimizer.zero_grad()
            reconstructed_image, mean_vector, log_variance = variational_autoencoder(batch_images)
            loss = loss_function(reconstructed_image, batch_images, mean_vector, log_variance)
            loss.backward()
            adam_optimizer.step()

            epoch_loss += loss.item()

        print(f"Latent Dim: {latent_dim}, Epoch [{epoch + 1}/{num_epochs}], Loss: {epoch_loss / len(dataset):.4f}")

    return variational_autoencoder
```

این بخش کد برای آموزش مدل‌های VAE با ابعاد نهان مختلف (2، 4، و 16) استفاده می‌شود. در هر بار حلقه، مدل با ابعاد نهان مشخص به تابع train_vae داده می‌شود و مدل آموزش‌دیده در دیکشنری models ذخیره می‌شود.

```
latent_dims = [2, 4, 16]
models = {}
for dim in latent_dims:
    print(f"\nTraining VAE with latent dimension: {dim}")
    models[dim] = train_vae(latent_dim=dim)
```

```
Training VAE with latent dimension: 2
Latent Dim: 2, Epoch [1/20], Loss: 1.4748
Latent Dim: 2, Epoch [2/20], Loss: 1.2958
Latent Dim: 2, Epoch [3/20], Loss: 1.2575
Latent Dim: 2, Epoch [4/20], Loss: 1.2345
Latent Dim: 2, Epoch [5/20], Loss: 1.2188
Latent Dim: 2, Epoch [6/20], Loss: 1.2067
Latent Dim: 2, Epoch [7/20], Loss: 1.1969
Latent Dim: 2, Epoch [8/20], Loss: 1.1884
Latent Dim: 2, Epoch [9/20], Loss: 1.1808
Latent Dim: 2, Epoch [10/20], Loss: 1.1742
Latent Dim: 2, Epoch [11/20], Loss: 1.1686
```

تابع compare_reconstructions برای مقایسه بازسازی تصاویر از مدل‌های مختلف تعریف شده است.ابتدا تصویر تست به دستگاه منتقل شده و به صورت یک بردار مسطح تبدیل می‌شود.برای هر مدل در دیکشنریmodels ، مدل به حالت eval تغییر وضعیت می‌دهد تا از آموزش جلوگیری شود.

با استفاده از تصاویر تست، مدل بازسازی تصاویر را بدون محاسبه گرادیان‌ها انجام می‌دهد.سپس تصاویر بازسازی‌شده به ابعاد اصلی (28x28) تغییر اندازه داده و به فرمت NumPy تبدیل می‌شوند.تصاویر بازسازی‌شده برای هر مدل در یک subplot نمایش داده می‌شوند، به طوری که در هر ردیف تصاویر مربوط به یک مدل قرار می‌گیرند.برچسب ابعاد نهان (latent dimension)در کنار اولین تصویر هر مدل نمایش داده می‌شود.در نهایت، عنوان کلی Comparison of Reconstructionsدر بالای تمام تصاویر نمایش داده می‌شود و تمام تصاویر به کمک plt.show()به نمایش درمی‌آیند.

```
def compare_reconstructions(models, test_images):
    plt.figure(figsize=(18, 6))
    num_images = 10
    test_images = test_images[:num_images].to(device)
    test_images_flat = test_images.view(-1, 784)

    for i, (latent_dim, model) in enumerate(models.items()):
        model.eval()
        with torch.no_grad():
            reconstructed_images, _, _ = model(test_images_flat)
            reconstructed_images = reconstructed_images.view(-1, 28, 28).cpu().detach().numpy()

        for j in range(num_images):
            plt.subplot(len(models), num_images, i * num_images + j + 1)
            plt.imshow(reconstructed_images[j], cmap='gray')
            plt.axis('off')
            if j == 0:
                plt.ylabel(f"Latent Dim: {latent_dim}", fontsize=12)

    plt.suptitle("Comparison of Reconstructions", fontsize=16)
    plt.show()
```

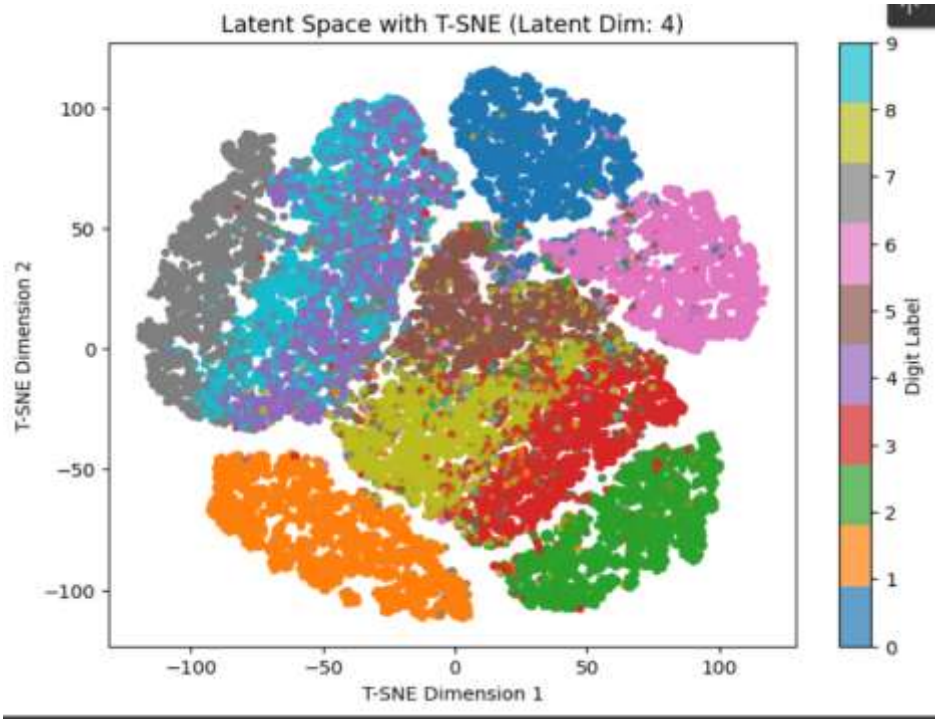
تابع plot_latent_space_with_tsne فضای نهان مدل را با استفاده از T-SNE به دو بعد کاهش می‌دهد.

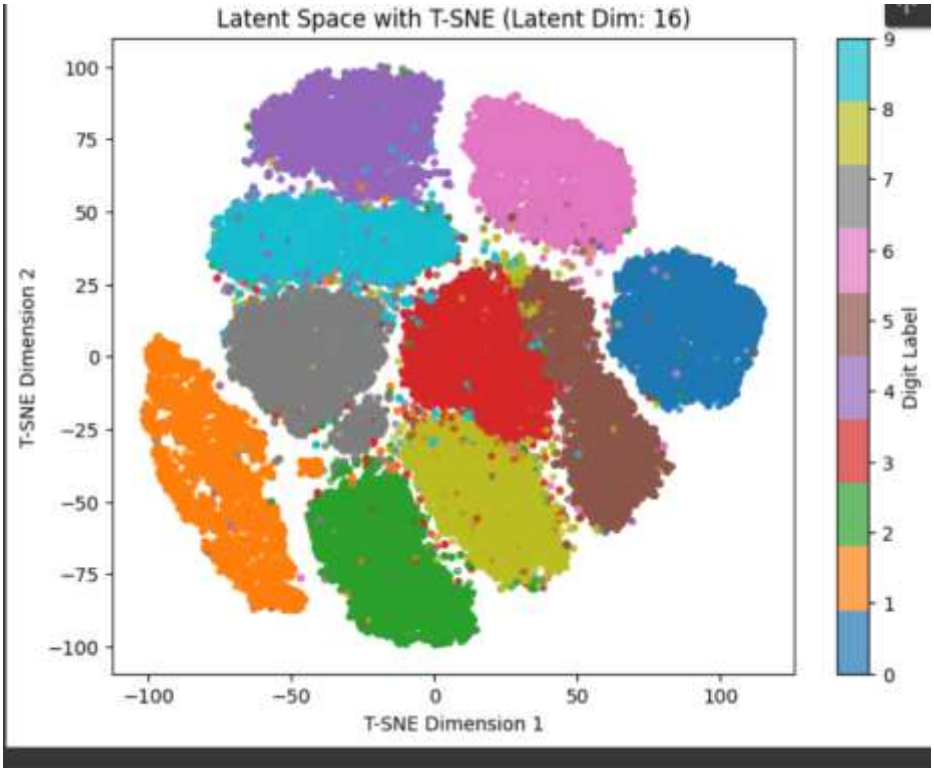
ابتدا تصاویر ورودی از data_loader گرفته شده و به صورت بردار مسطح تبدیل می‌شوند.سپس میانگین و لگاریتم واریانس از بخش Encoder مدل استخراج می‌شود و فضای نهان به کمک reparameterization به دست می‌آید.فضای نهان به همراه برچسب‌های مربوطه در لیست‌ها ذخیره می‌شود.

T-SNEبرای کاهش ابعاد فضای نهان به 2 بعد استفاده می‌شود.سپس فضای نهان کاهش‌یافته با استفاده از plt.scatter نمایش داده می‌شود، که رنگ‌ها نمایانگر برچسب‌های ارقام هستند.عنوان، برچسب‌ها و نمودار رنگی برای شناسایی هر عدد اضافه می‌شود.

در نهایت این عملیات برای ابعاد نهان مختلف (4 و 16) انجام می‌شود و فضای نهان هر یک نمایش داده می‌شود.

```python
def plot_latent_space_with_tsne(model, data_loader, latent_dim):
    model.eval()
    latent_vectors, labels_list = [], []

    with torch.no_grad():
        for batch_images, labels in data_loader:
            batch_images = batch_images.view(batch_images.size(0), -1).to(device)
            mean_vector, log_variance = model.encoder(batch_images)
            latent_vector = model.reparameterize(mean_vector, log_variance)
            latent_vectors.append(latent_vector.cpu())
            labels_list.append(labels)

    latent_vectors = torch.cat(latent_vectors)
    labels = torch.cat(labels_list)

    tsne = TSNE(n_components=2, random_state=42)
    tsne_latent = tsne.fit_transform(latent_vectors)

    plt.figure(figsize=(8, 6))
    scatter = plt.scatter(tsne_latent[:, 0], tsne_latent[:, 1], c=labels, cmap='tab10', alpha=0.7, s=10)
    plt.colorbar(scatter, label='Digit Label')
    plt.xlabel("T-SNE Dimension 1")
    plt.ylabel("T-SNE Dimension 2")
    plt.title(f"Latent Space with T-SNE (Latent Dim: {latent_dim})")
    plt.show()

for dim in [4, 16]:
    print(f"\nVisualizing latent space with T-SNE for latent dimension: {dim}")
    plot_latent_space_with_tsne(models[dim], data_loader, latent_dim=dim)
```


Latent Space with T-SNE (Latent Dim: 4)
T-SNE Dimension 1
T-SNE Dimension 2
Digit Label
100
50
0
−50
−100
−100
−50
0
50
100
9
8
7
6
5
4
3
2
1
0


Latent Space with T-SNE (Latent Dim: 16)
T-SNE Dimension 1
T-SNE Dimension 2
Digit Label
100
75
50
25
0
−25
−50
−75
−100
−100
−50
0
50
100
9
8
7
6
5
4
3
2
1
0

سوال دوم)

الف) GANیک مدل یادگیری عمیق است که از دو بخش اصلی **مولد** (Generator) و **تمایزدهنده** (Discriminator) تشکیل شده است. این دو بخش به‌صورت رقابتی آموزش می‌بینند تا مولد بتواند داده‌هایی مشابه داده‌های واقعی تولید کند. این فرآیند شبیه به رقابت بین یک هنرمند و یک منتقد است؛ هنرمند تلاش می‌کند آثار خود را به گونه‌ای خلق کند که واقعی به نظر برسند، درحالی‌که منتقد سعی دارد تفاوت آثار هنرمند با آثار واقعی را تشخیص دهد.

Generator

وظیفه :تولید داده‌های جدید که به داده‌های واقعی شباهت داشته باشد.
مولد یک نویز تصادفی (به‌عنوان ورودی) دریافت می‌کند و آن را از طریق چندین لایه عصبی پردازش کرده و به یک داده خروجی (مثل یک تصویر) تبدیل می‌کند. این داده خروجی باید به حدی واقعی به نظر برسد که بتواند تمایزدهنده را فریب دهد.یادگیری نحوه تولید داده‌هایی که تمایزدهنده نتواند تفاوت آن‌ها را با داده‌های واقعی تشخیص دهد.

Discriminator

وظیفه :تشخیص اینکه داده ورودی واقعی است (از داده‌های آموزشی) یا جعلی است (تولیدشده توسط مولد).
تمایزدهنده داده‌های واقعی و جعلی را دریافت می‌کند، آن‌ها را از طریق چندین لایه عصبی پردازش کرده و یک احتمال خروجی می‌دهد. این احتمال نشان می‌دهد که داده چقدر واقعی به نظر می‌رسد.بهبود توانایی خود در تشخیص داده‌های واقعی از داده‌های جعلی.

مولد یک داده جعلی (مثل یک تصویر) تولید می‌کند.تمایزدهنده هم داده‌های واقعی و هم داده‌های جعلی را دریافت کرده و سعی می‌کند آن‌ها را از هم تشخیص دهد.مولد از بازخورد تمایزدهنده استفاده می‌کند تا داده‌های بهتری تولید کند که بتواند تمایزدهنده را فریب دهد.این فرآیند به‌صورت تکراری ادامه پیدا می‌کند.

مولد تلاش می‌کند داده‌های واقعی‌تر بسازد.

تمایزدهنده تلاش می‌کند قوی‌تر شود و داده‌های جعلی را بهتر تشخیص دهد.

ب) تابع زیان در GAN رقابتی طراحی شده است تا فرآیند یادگیری میان دو شبکه، مولد (Generator) و تمایزدهنده (Discriminator)، به‌طور متقابل بهبود یابد. در این رقابت، هدف مولد تولید داده‌های واقعی‌تر و هدف تمایزدهنده تشخیص داده‌های واقعی از جعلی است.

تعریف تابع زیان

$$LD=-Ex\sim Pdata[logD(x)]-Ez\sim Pz[log(1-D(G(z)))]$$

زیان تمایزدهنده:(Discriminator Loss)

تمایزدهنده تلاش می‌کند داده‌های واقعی را از داده‌های جعلی تشخیص دهد. زیان آن به‌صورت مجموع احتمال درست بودن داده‌های واقعی و اشتباه بودن داده‌های تولیدی است.

زیان مولد:(Generator Loss)

مولد تلاش می‌کند داده‌هایی تولید کند که تمایزدهنده نتواند آن‌ها را از داده‌های واقعی تشخیص دهد. به این ترتیب، مولد سعی دارد داده‌های خود را به‌طوری تولید کند که تمایزدهنده آن‌ها را واقعی بپندارد.

این رقابت باعث می‌شود که مولد به‌طور مداوم داده‌های خود را بهبود دهد تا واقعی‌تر شوند، و تمایزدهنده نیز یاد می‌گیرد که تفاوت بین داده‌های واقعی و جعلی را بهتر تشخیص دهد. هدف نهایی این است که مولد قادر به تولید داده‌هایی باشد که تمایزدهنده نتواند آن‌ها را از داده‌های واقعی تمایز دهد.

ج)

هدف CycleGAN :برای تبدیل تصاویر از یک دامنه به دامنه دیگر بدون نیاز به جفت داده‌های متنی (paired data) طراحی شده است.

از دو شبکه Generator و Discriminator مشابه GAN استاندارد استفاده می‌کند.به‌جای جفت داده‌ها، به‌طور خودکار تصاویر از یک دامنه را به دامنه دیگر تبدیل می‌کند.از دور زدن چرخه‌ای (Cycle Consistency Loss) برای اطمینان از حفظ ویژگی‌های اصلی تصویر در فرایند تبدیل استفاده می‌کند.

تفاوت با GAN استاندارد CycleGAN :نیاز به جفت‌های واقعی تصاویر ندارد و فرآیند تبدیل دوطرفه را انجام می‌دهد.

هدف StyleGAN :برای تولید تصاویر با کیفیت بالا (مانند تصاویر انسان‌ها، اشیاء و مناظر) از یک فضای نهان (latent space) استفاده می‌کند.شبکه‌های مولد چندلایه: به جای استفاده از فضای نهان استاندارد، StyleGANاز لایه‌های مختلف استفاده می‌کند تا سبک‌های مختلف تصویر (مانند سبک، فرم، رنگ و بافت) را کنترل کند.استفاده از تکنیک‌های نرمال‌سازی و ویژگی‌های سبک (style-based architecture) برای کنترل دقیق‌تر ویژگی‌های تصویر.

تفاوت با GAN استاندارد StyleGAN :به جای یک فضای نهان یکپارچه، از یک فضای چندلایه استفاده می‌کند که به‌طور دقیق‌تری می‌تواند ویژگی‌های مختلف تصویر را مدل‌سازی کند.

هدف DCGAN :برای بهبود عملکرد GAN در زمینه تولید تصاویر با استفاده از معماری‌های کانولوشنی عمیق طراحی شده است.از لایه‌های کانولوشنی و لایه‌های نرمال‌سازی برای آموزش و تولید تصاویر با کیفیت بالا استفاده می‌کند.از بهینه‌سازی‌های خاص مثل استفاده از Batch Normalization و ReLU activations برای بهبود پایداری فرآیند آموزش استفاده می‌کند.

تفاوت با GAN استاندارد DCGAN :از معماری‌های عمیق‌تر و پیچیده‌تری استفاده می‌کند تا تصاویر با کیفیت بهتری تولید کند و همچنین آموزش پایدارتری داشته باشد.

تفاوت‌های کلیدی بین این مدل‌ها و GAN استاندارد:

Data Pairing: CycleGANنیازی به جفت داده‌ها ندارد، در حالی که GAN استاندارد به جفت داده‌های واقعی و تولیدی نیاز دارد.

کیفیت تصویر StyleGAN :با استفاده از ساختارهای پیچیده‌تر و کنترل‌های سبک، تصاویر بسیار با کیفیت‌تری نسبت به GAN استاندارد تولید می‌کند.

معماری DCGAN :از معماری کانولوشنی عمیق استفاده می‌کند که به آن اجازه می‌دهد تصاویر با کیفیت بالاتری تولید کند.

در مجموع، این مدل‌ها با استفاده از تکنیک‌های مختلف سعی در بهبود کیفیت و پایداری تولید تصاویر دارند و تفاوت‌های مهمی با GANاستاندارد از نظر کاربرد و معماری دارند.

د) شباهت‌ها:

هر دو مدل، تولید داده‌های جدید و شبیه‌سازی داده‌های ورودی (مثلاً تصاویر) را هدف دارند. این هدف اصلی به تولید نمونه‌های جدید با ویژگی‌های مشابه داده‌های واقعی است.هم GAN و هم VAE از شبکه‌های مولد (Generative Networks) برای تولید داده‌ها استفاده می‌کنند. هر دو مدل به‌گونه‌ای طراحی شده‌اند که قادر به تولید داده‌های جدید از یک فضای نهان هستند.

تفاوت‌ها:

درVAE ، فضای نهان به‌طور پیوسته مدل‌سازی می‌شود و از توزیع‌های احتمالی استفاده می‌کند. به‌طور خاص، VAEیک مدل احتمالی است که از شبکه‌های اتوانکودر و مولد برای یادگیری توزیع داده‌ها استفاده می‌کند. فرآیند آموزش VAE شامل حداقل‌سازی تابع خطای بازسازی و مقدار KL divergence برای همسان‌سازی توزیع فضای نهان با توزیع نرمال است.

درGAN ، یک Generatorو یک Discriminatorرقابت می‌کنند .Generator داده‌های جعلی تولید می‌کند و Discriminatorتلاش می‌کند تفاوت بین داده‌های واقعی و جعلی را تشخیص دهد. هدف این است که Generator به‌گونه‌ای بهینه شود که Discriminator نتواند تفاوت بین داده‌های واقعی و جعلی را تشخیص دهد.

VAEبه‌طور کلی از حداقل‌سازی تابع خطای بازسازی (reconstruction error)و KL divergenceبرایهماهنگ‌سازی توزیع احتمالی استفاده می‌کند. فرآیند آموزش به‌صورت بهینه‌سازی همزمان دو هدف است.

GANاز یک فرآیند رقابتی بین Generator و Discriminator استفاده می‌کند. هدف Generator ایجاد داده‌های واقعی به‌قدری شبیه داده‌های واقعی است که Discriminator نتواند آن‌ها را از داده‌های واقعی تشخیص دهد.

VAEاز فضای نهان پیوسته برای تولید داده‌ها استفاده می‌کند. این بدان معناست که خروجی‌های تولیدی VAE معمولاً نرم‌تر و کمی تکراری هستند.

GANها قادر به تولید داده‌هایی با ویژگی‌های دقیق‌تر و کیفیت بالاتر هستند زیرا در طول فرآیند آموزش، Generatorتلاش می‌کند تا داده‌هایی شبیه به داده‌های واقعی تولید کند.آموزش VAE معمولاً پایدارتر است زیرا به‌طور همزمان از دو هدف استفاده می‌کند که باعث همگرایی بهتر می‌شود.آموزش GAN پیچیده‌تر است و به‌ویژه ممکن است در صورتی که Generator و Discriminatorبه‌طور مناسب تراز نشوند، منجر به نوسانات یا عدم همگرایی شود.

تابع str2img یک رشته متنی ورودی را به یک آرایه عددی تبدیل می‌کند، سپس طول آن را می‌سازد تا به ابعاد مربع تبدیل شود (مربوط به ابعاد تصویر).

برای هر مجموعه داده(y_train, y_valid, y_test) ، ابتدا مقادیر پیکسل‌ها فیلتر شده و سپس با استفاده از str2img به آرایه‌های 2 بعدی تبدیل می‌شود.در نهایت، ابعاد آرایه‌های تبدیل‌شده برای مجموعه‌های آموزشی، اعتبارسنجی و تست چاپ می‌شود.

```python
import numpy as np
import math


def str2img(x):
  data = np.array([int(val) for val in x.split()])
  dimention = int(math.sqrt(len(data)))
  data = data.reshape(dimention, dimention)
  return data

y_train = df.loc[df['Usage'] == 'Training', 'pixels']
y_train = np.stack(y_train.apply(str2img), axis=0)

y_valid = df.loc[df['Usage'] == 'PrivateTest', 'pixels']
y_valid = np.stack(y_valid.apply(str2img), axis=0)

y_test = df.loc[df['Usage'] == 'PublicTest', 'pixels']
y_test = np.stack(y_test.apply(str2img), axis=0)

y_train.shape, y_valid.shape, y_test.shape
```

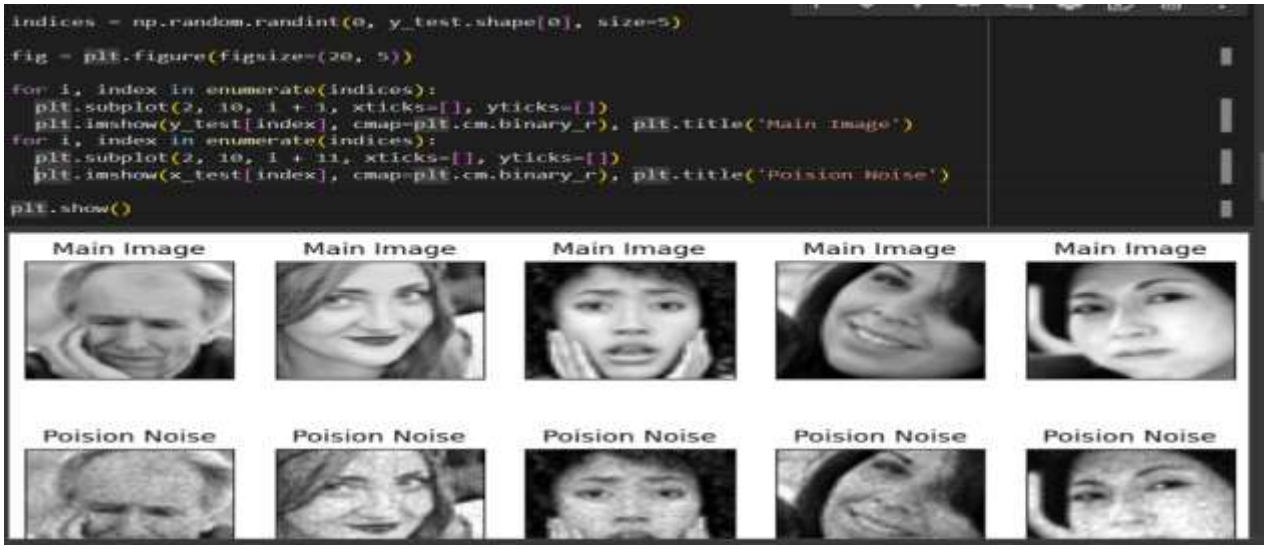
```
((28709, 48, 48), (3589, 48, 48), (3589, 48, 48))
```

تابع add_poisson_noise ابتدا یک کپی از تصویر ورودی می‌سازد، سپس با استفاده از توزیع پواسون به تصویر نویز اضافه می‌کند.نتیجه نویزی شده به محدوده [0, 255] محدود می‌شود تا از مقادیر غیرمجاز جلوگیری شود.سپس این تابع برای هر تصویر در مجموعه‌های آموزشی، اعتبارسنجی و تست (y_train, y_valid, y_test)اعمال می‌شود تا مجموعه‌های نویزی‌شده (x_train, x_valid, x_test)به دست آید.

```
import cv2 as cv
mean=0
sigma=10

def add_poisson_noise(img):
    image = img.copy()
    noisy_image = np.random.poisson(image).astype('float32')
    noisy_image = np.clip(noisy_image, 0, 255)
    return noisy_image

x_train = np.array([add_poisson_noise(img) for img in y_train])
x_valid = np.array([add_poisson_noise(img) for img in y_valid])
x_test = np.array([add_poisson_noise(img) for img in y_test])
```

در این بخش، پنج تصویر از مجموعه تست (y_test) و نسخه نویزی‌شده آنها (x_test) نمایش داده می‌شود:

ابتدا 5 شاخص تصادفی از مجموعه تست انتخاب می‌شود.با استفاده ازplt.subplot ، در یک شبکه 2 سطری و 10 ستونی، تصاویر اصلی (y_test) و تصاویر نویزی‌شده (x_test) به صورت کنار هم قرار می‌گیرند.تصاویر اصلی در ردیف اول و تصاویر نویزی‌شده در ردیف دوم نمایش داده می‌شوند.برای هر تصویر، برچسب "Main Image" و "Poison Noise" در کنار آنها قرار می‌گیرد تا تفاوت‌ها واضح‌تر شوند.در نهایت، با استفاده از plt.show()تصاویر به نمایش در می‌آیند.



در ابتدا، دو مدل اصلی تعریف می‌شوند:

Generatorاین مدل تصاویر جدیدی از نویز تولید می‌کند. با استفاده از لایه‌های کانولوشن و ترنس‌پوز کانولوشن، ویژگی‌های تصاویری با ابعاد 48 48xرا می‌آموزد و از این ویژگی‌ها برای بازسازی تصاویر مشابه به تصاویر واقعی استفاده می‌کند.

Discriminatorاین مدل به تمایز بین تصاویر واقعی (از داده‌های آموزشی) و تصاویر تولیدشده (توسط Generator) می‌پردازد. این مدل از لایه‌های کانولوشن و لایه‌های Dropout برای جلوگیری از overfitting استفاده می‌کند.

سپس، توابع ضرر برای هر دو مدل تعریف می‌شود:

Generator Lossمیزان موفقیت Generator در تولید تصاویر واقعی را اندازه‌گیری می‌کند.

Discriminator Lossمیزان دقت Discriminator در تشخیص تصاویر واقعی و تولیدی را محاسبه می‌کند.

در ادامه، حلقه آموزش (train_step) برای به‌روزرسانی وزن‌های مدل‌ها با استفاده از گرادیان‌ها طراحی شده است. در هر مرحله از آموزش، Generatorتصویری تولید می‌کند و Discriminator با استفاده از تصاویر واقعی و تولیدی اقدام به تمایز می‌کند. سپس، ضررها محاسبه و با استفاده از بهینه‌سازAdam ، وزن‌ها به‌روزرسانی می‌شوند.

در نهایت، داده‌ها برای ورودی به مدل‌ها آماده می‌شوند و مدل‌ها آموزش داده می‌شوند. پس از پایان آموزش، تصاویر تولیدشده از مجموعه آزمایش (x_test) به‌دست آمده و در قالب تصویری با استفاده از Matplotlib نمایش داده می‌شوند تا کیفیت بازسازی تصاویر نویزی‌شده توسط GAN بررسی شود.

```python
def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Input(shape=(48, 48, 1)))
    model.add(layers.Conv2D(64, (5, 5), padding='same', activation='relu'))
    model.add(layers.Conv2D(128, (3, 3), strides=(2, 2), padding='same', activation='relu'))
    model.add(layers.Conv2D(128, (3, 3), strides=(2, 2), padding='same', activation='relu'))
    model.add(layers.Conv2DTranspose(128, (3, 3), strides=(2, 2), padding='same', activation='relu'))
    model.add(layers.Conv2DTranspose(64, (3, 3), strides=(2, 2), padding='same', activation='relu'))
    model.add(layers.Conv2D(1, (5, 5), padding='same', activation='sigmoid'))
    return model

# Define the discriminator model
def make_discriminator_model():
    model = tf.keras.Sequential()
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same', input_shape=[48, 48, 1]))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Flatten())
    model.add(layers.Dense(1))
    return model

# Define the loss functions and optimizers
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
```

```python
def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss

def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)


generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)

# Define the training loop
EPOCHS = 50
noise_dim = 100
num_examples_to_generate = 16

# We will reuse this seed overtime (so it's easier)
# to visualize progress in the animated GIF)
seed = tf.random.normal([num_examples_to_generate, 48, 48, 1])

# Notice the use of `tf.function`
# This annotation causes the function to be "compiled".
@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, 48, 48, 1])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
      generated_images = generator(noise, training=True)
```

```python
@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, 48, 48, 1])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
      generated_images = generator(noise, training=True)

      real_output = discriminator(images, training=True)
      fake_output = discriminator(generated_images, training=True)

      gen_loss = generator_loss(fake_output)
      disc_loss = discriminator_loss(real_output, fake_output)

    gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.trainable_v


# Reshape the data for the GAN
x_train = x_train.reshape(-1, 48, 48, 1)
x_valid = x_valid.reshape(-1, 48, 48, 1)
x_test = x_test.reshape(-1, 48, 48, 1)
y_train = y_train.reshape(-1, 48, 48, 1)
y_valid = y_valid.reshape(-1, 48, 48, 1)
y_test = y_test.reshape(-1, 48, 48, 1)
```

```python
# Create the models
generator = make_generator_model()
discriminator = make_discriminator_model()

# Define batch size
BATCH_SIZE = 64

# Training loop
for epoch in range(EPOCHS):
    for image_batch in tf.data.Dataset.from_tensor_slices(x_train).batch(BATCH_SIZE):
        train_step(image_batch)
    print(f'Epoch {epoch+1} completed.')

# After training, generate images from the test set
generated_images = generator(x_test, training=False)

# Display or save the generated images
# Example using matplotlib
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(20, 5))
for i in range(5):
    plt.subplot(2, 10, i + 1, xticks=[], yticks=[])
    plt.imshow(y_test[i].reshape(48, 48), cmap=plt.cm.binary_r)
    plt.subplot(2, 10, i + 11, xticks=[], yticks=[])
    plt.imshow(generated_images[i].numpy().reshape(48, 48), cmap=plt.cm.binary_r)
```

در نهایت تصاویر چاپ میشوند.

```python
# Display or save the generated images
# Example using matplotlib
fig = plt.figure(figsize=(20, 5))
for i in range(5):
    plt.subplot(2, 10, i + 1, xticks=[], yticks=[])
    plt.imshow(generated_images[i].numpy().reshape(48, 48), cmap=plt.cm.binary_r)
    plt.title(f"denoised Image {i+1}") # Added title to original image
plt.show()
```

سوال سوم)

الف) ایده اصلی معماری مدل‌های دیفیوژن، ایجاد یک فرایند تدریجی برای اضافه کردن نویز به داده‌ها (مثل تصاویر) و سپس یادگیری نحوه بازسازی آن‌ها از حالت نویزی به حالت اصلی است. این فرآیند در دو مرحله صورت می‌گیرد:

مرحله افزایشی نویز :(Forward Process) در این مرحله، به تدریج نویز به داده‌های ورودی اضافه می‌شود تا تصویر به یک توزیع تقریباً یکنواخت از نویز تبدیل شود.

مرحله کاهش نویز :(Reverse Process) پس از یادگیری این فرآیند توسط مدل، در این مرحله مدل سعی می‌کند تا از حالت نویزی به حالت اولیه تصویر بازگردد. این کار با استفاده از یک مدل پیش‌بینی که یاد گرفته چگونه نویز را از تصاویر حذف کند، انجام می‌شود .مدل‌های دیفیوژن معمولاً از شبکه‌های عصبی مثل (U-Net) برای پیش‌بینی و کاهش نویز استفاده می‌کنند.

ب) در فرآیند انتشار (Forward Process) در مدل‌های دیفیوژن، به تدریج نویز به داده‌های ورودی، مثل تصاویر، اضافه می‌شود تا تصویر به یک توزیع نویزی تبدیل شود. این فرآیند در هر گام به گونه‌ای عمل می‌کند که هر تصویر مرحله به مرحله بیشتر به نویز تبدیل می‌شود. این اضافه شدن نویز در طول گام‌های زمانی مختلف (تعداد مشخصی از گام‌ها) صورت می‌گیرد. در مدل‌های دیفیوژن این فرآیند به شرح زیر انجام می‌شود:

نویز به تصویر ورودی به صورت تدریجی در هر گام زمانی اضافه می‌شود. این افزایشی بودن به این معنا است که تصویر در ابتدا جزئیات زیادی دارد و در نهایت به یک تصویر کاملاً نویزی تبدیل می‌شود که دیگر ویژگی‌های خاص تصویر اصلی را ندارد.

1. (Beta schedule)تابع بتا در مدل‌های دیفیوژن معمولاً به صورت یک دنباله خطی یا غیرخطی از مقادیر بتا (که میزان نویز را مشخص می‌کند) تعریف می‌شود. در هر گام از فرآیند انتشار، به تصویر مقداری نویز از توزیع نرمال با واریانس تعیین شده توسط این تابع اضافه می‌شود.
2. **عملکرد تابع بتا** :در هر گام زمانی t، مقدار نویز به تصویر اصلی اضافه می‌شود به طوری که تصویری که در گام‌های قبلی تولید شده، توسط نویز در گام tبه صورت زیر اصلاح می‌شود.

در این فرآیند، به دلیل اضافه شدن تدریجی نویز و متناسب با مقادیر بتا در هر گام، تصویر اولیه به تدریج به یک توزیع تصادفی تبدیل می‌شود که دیگر مشابه تصویر اصلی نیست. این فرآیند معمولاً برای مدت زمان مشخصی (تعداد گام‌ها) ادامه پیدا می‌کند تا تصویر نهایی کاملاً نویزی شود.

پ) در فرآیند معکوس (Reverse Process) در مدل‌های دیفیوژن، هدف این است که داده‌های نویزی به تدریج به داده‌های اصلی (داده‌های اولیه) بازسازی شوند. این فرآیند برخلاف فرآیند انتشار (Forward Process) که به تدریج نویز به داده‌ها اضافه می‌کند، در فرآیند معکوس نویز را از داده‌های نویزی حذف می‌کند و ویژگی‌های اصلی داده را به آن باز می‌گرداند.

در این فرآیند، مدل دیفیوژن از مدل‌های آموزش‌دیده برای تخمین نویزی که باید از هر تصویر حذف شود استفاده می‌کند. با استفاده از مدل‌هایی مانند شبکه‌های عصبی، در هر گام از فرآیند معکوس، مدل سعی می‌کند تصویر نویزی را به شکل تدریجی به تصویر اصلی تبدیل کند. این کار با استفاده از تابعی که نویز را در هر گام کاهش می‌دهد انجام می‌شود.

فرآیند معکوس به این صورت عمل می‌کند:

1. مدل از تصویر نویزی در یک گام زمانی خاص شروع می‌کند.
2. سپس با استفاده از اطلاعات آموزش‌دیده، مدل تخمین می‌زند که نویز باید چقدر از تصویر حذف شود.
3. این فرآیند برای تمامی گام‌های زمانی تکرار می‌شود تا در نهایت تصویر اصلی بازسازی شود.

در حقیقت، فرآیند معکوس مدل دیفیوژن معادل یک بازسازی تصادفی است که داده‌های نویزی را به داده‌های اصلی و قابل شناسایی تبدیل می‌کند.

ت) تابع خطا در مدل‌های دیفیوشن معمولاً برای اندازه‌گیری فاصله بین پیش‌بینی مدل و حقیقت (ground truth) تعریف می‌شود. در بیشتر مدل‌های دیفیوشن، تابع خطا به صورت حداقل‌سازی خطای میانگین مربعات (MSE) تعریف می‌شود که در آن مدل سعی می‌کند نویز پیش‌بینی‌شده را با نویز واقعی (حقیقی) که در فرآیند انتشار به تصویر اضافه شده، مقایسه کند.

در فرآیند آموزش مدل دیفیوشن، هدف این است که مدل بتواند تخمین دقیقی از نویز موجود در تصاویر نویزی بدست آورد تا بتواند در فرآیند معکوس، تصاویر اصلی را بازسازی کند. بنابراین، در هر گام زمانی، مدل باید قادر باشد نویز واقعی را از تصویر نویزی تخمین بزند. برای این منظور، تابع خطا معمولاً به این شکل محاسبه می‌شود:

$$\mathbb{E}\left[\|\epsilon - \hat{\epsilon}\|^2\right] = \text{Loss}$$

یادگیری پارامترهای فرآیند معکوس در مدل‌های دیفیوشن بسیار مهم است زیرا این پارامترها مسئول بازسازی صحیح تصویر از داده‌های نویزی هستند. فرآیند معکوس نیاز دارد که مدل بتواند در هر گام زمانی تخمینی دقیق از نویز موجود در تصویر بدست آورد و آن را از تصویر حذف کند. در صورتی که این پارامترها به درستی آموزش داده نشوند، فرآیند معکوس به درستی عمل نخواهد کرد و مدل قادر به بازسازی صحیح تصاویر اصلی نخواهد بود.

این پارامترها شامل تخمین‌های نویز در هر گام زمانی و همچنین شیب‌های مدل است که برای تطبیق تغییرات در نویز و بازسازی تصاویر از آن استفاده می‌شود. برای بهینه‌سازی مدل و اطمینان از عملکرد صحیح فرآیند معکوس، این پارامترها باید به‌طور دقیق یادگیری شوند.

ث) معماری‌های معمول برای مدل‌های دیفیوشن معمولاً بر اساس شبکه‌های عصبی پیچیده‌ای هستند که به‌طور خاص برای پیش‌بینی نویز و بازسازی داده‌ها از توزیع نویزی طراحی شده‌اند. معماری‌های رایج در این مدل‌ها شامل موارد زیر می‌شوند:

شبکه‌های عصبی کانولوشنی:(CNNs)

شبکه‌های عصبی کانولوشنی (CNNs) به‌طور گسترده در مدل‌های دیفیوشن استفاده می‌شوند زیرا توانایی پردازش تصاویر را به‌طور مؤثر دارند. این شبکه‌ها به‌ویژه برای شبیه‌سازی فرآیندهای پیچیده مانند بازسازی و پیش‌بینی نویز از داده‌های تصویری

بسیار مفید هستندCNN .ها برای اعمال فیلترها روی بخش‌های مختلف تصویر و استخراج ویژگی‌های سطح بالا از آن استفاده می‌شوند.

شبکه‌های عصبی یونت:(UNet)

شبکه‌های UNetبه‌طور ویژه در مدل‌های دیفیوشن برای پیش‌بینی نویز و بازسازی تصاویر مورد استفاده قرار می‌گیرند. این شبکه‌ها دارای معماری **رمزگذار-رمزگشا** هستند که شامل لایه‌های کاهشی و افزایشی می‌باشند. در این معماری، شبکه به‌طور مکرر از ویژگی‌های سطح پایین به سطح بالا و بالعکس استفاده می‌کند. این ساختار به شبکه اجازه می‌دهد تا اطلاعات دقیق‌تر و ویژگی‌های جزئی تصویر را حفظ کند و به دقت پیش‌بینی نویز و بازسازی تصویر کمک کند.

شبکه‌های عصبی مولد:(GANs)

در بعضی مدل‌های دیفیوشن، از **شبکه‌های عصبی مولد** (Generative Adversarial Networks)نیز استفاده می‌شود، اگرچه معمولاًGAN ها در مدل‌های دیفیوشن استفاده نمی‌شوند، اما از آن‌ها برای بهبود کیفیت تصاویر بازسازی‌شده و اطمینان از شباهت بیشتر با داده‌های واقعی می‌توان استفاده کرددر مدل‌های دیفیوشن، شبکه‌های عصبی معمولاً برای پیش‌بینی نویز آموزش داده می‌شوند تا بتوانند فرآیند معکوس انتشار را به‌طور دقیق انجام دهند. این به‌این‌معناست که شبکه‌ها باید یاد بگیرند که چطور نویز واقعی را که به تصویر اضافه شده است، پیش‌بینی کنند و سپس آن را از تصویر حذف کنند.

پیش‌بینی نویز: شبکه عصبی در هر گام از فرآیند معکوس باید تخمینی از نویز اضافه‌شده در آن گام زمانی را ارائه دهد. این کار به‌طور معمول با استفاده از خطای میانگین مربعات (MSE) بین نویز پیش‌بینی‌شده و نویز واقعی انجام می‌شود.

پیش‌بینی توزیع داده‌ها: در بعضی مدل‌ها، به‌جای پیش‌بینی دقیق نویز، مدل ممکن است برای تخمین توزیع داده‌ها آموزش ببیند. به‌طور خاص، برخی مدل‌ها از شبکه‌های مولد برای مدل‌سازی توزیع داده‌ها استفاده می‌کنند.

در کل، اکثر مدل‌های دیفیوشن بر پیش‌بینی نویز تمرکز دارند، زیرا نویز دقیقی که در فرآیند انتشار اضافه می‌شود، عامل اصلی در بازسازی تصاویر از توزیع نویزی به تصویر اصلی است.

ج) تعداد مراحل دیفیوژن تأثیر زیادی بر کیفیت و سرعت بازسازی دارد:

هر مرحله به مدل این امکان را می‌دهد که به تدریج اطلاعات دقیق‌تری از داده‌ها را به دست آورد. افزایش تعداد مراحل باعث می‌شود که مدل قادر به بازسازی دقیق‌تری از تصویر نویزی باشد، زیرا جزئیات بیشتری در طول فرآیند معکوس به‌دست می‌آید. در صورتی که مراحل کم باشند، مدل قادر به بازیابی کامل ویژگی‌های داده نخواهد بود و کیفیت بازسازی کاهش می‌یابد.

با افزایش تعداد مراحل، زمان مورد نیاز برای اجرای فرآیند معکوس بیشتر می‌شود. این امر به دلیل این است که مدل باید گام‌های بیشتری را برای بازسازی داده‌ها طی کند. در مقابل، کاهش تعداد مراحل سرعت فرآیند را افزایش می‌دهد، اما ممکن است در این حالت کیفیت بازسازی کاهش یابد.

در نتیجه، تعداد مراحل باید به‌گونه‌ای تنظیم شود که تعادلی بین کیفیت نهایی و زمان پردازش به‌دست آید. برای استفاده در برنامه‌های عملی، تنظیم این مقدار بر اساس نیاز سیستم و دقت مورد انتظار انجام می‌شود.

چ)

مزایا:

ثبات در آموزش :مدل‌های دیفیوژن به دلیل فرآیند پیوسته خود، نسبت بهGAN ها که ممکن است با مشکلاتی مانند "mode collapse"مواجه شوند، آموزش ثابتی دارند.

کیفیت بالای تصاویر :دیفیوشن‌ها قادر به تولید تصاویر با جزئیات و کیفیت بالاتر هستند.

مدل‌سازی توزیع‌های پیچیده :می‌توانند توزیع‌های پیچیده‌تری را مدل کنند و مستقیماً از نویز داده‌ها تولید کنند.

معایب:

زمان آموزش و تولید طولانی :تعداد گام‌های زیاد برای تولید تصاویر، زمان آموزش و تولید را بالا می‌برد.

نیاز به منابع محاسباتی بیشتر :به دلیل تعداد گام‌های زیاد، منابع محاسباتی بیشتری می‌طلبند.

کنترل کمتر بر ویژگی‌ها :برخلافGAN ها که قابلیت کنترل دقیق‌تر بر ویژگی‌ها دارند، مدل‌های دیفیوژن این ویژگی را کمتر دارند.

مقایسه با GAN وVAE

دیفیوشن‌ها در آموزش پایدارتر ازGAN ها هستند، اما زمان بیشتری برای تولید تصاویر نیاز دارند.

در مقایسه باVAE ها، مدل‌های دیفیوژن تصاویر با کیفیت‌تری تولید می‌کنند، اما منابع بیشتری می‌خواهند.

ابتدا ابزارها و کتابخانه‌های موردنیاز برای پیاده‌سازی پروژه، از جمله PyTorch، NumPy، و Matplotlib، نصب و تنظیم شدند. این مرحله شامل آماده‌سازی محیط برنامه‌نویسی، تعریف وابستگی‌ها، و تنظیم دستگاه CPU یا GPUبرای اجرای کدها بود.

```
[ ]  from datasets import load_dataset
     from PIL import Image
     import torch.nn.functional as F
     import os
     from tqdm.notebook import tqdm
     import torch
     import numpy as np
     import torchvision.transforms as transforms
     import torch.nn as nn
     import torchvision
     import math
     import matplotlib.pyplot as plt
     import torch
     import urllib
     import PIL
     from unet import UNet
```

```
[ ]  device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
     device
```

transformاین بخش ورودی‌های تصویری را برای آموزش مدل آماده می‌کند.

transforms.Resize(IMAGE_SHAPE)ابعاد تصویر را به یک اندازه استاندارد 32*32 تغییر می‌دهد.

transforms.ToTensor()داده‌های تصویر که معمولاً به شکل آرایه‌ای هستند، به فرمت تنسور PyTorch تبدیل می‌شوند.

transforms.Lambda(lambda t: (t * 2) - 1)مقادیر پیکسل‌ها که به‌طور معمول در بازه [0, 1]هستند، به بازه [-1, 1]نگاشت می‌شوند. این کار برای بهبود عملکرد مدل در کار با داده‌ها انجام می‌شود.

reverse_transformاین تبدیل برای برگرداندن داده‌ها از حالت تنسور به تصویر اصلی استفاده می‌شود. (t + 1) / 2مقادیر بازه [-1, 1]را به [0, 1]باز می‌گرداند.

permuteترتیب ابعاد (channel, height, width) به (height, width, channel) تغییر می‌کند که فرمت استاندارد تصاویر است.

t.cpu().numpy().astype(np.uint8)داده‌های تنسور را به آرایه‌ای با مقادیر صحیح (۸ بیتی) تبدیل می‌کند.

```
] IMAGE_SHAPE = (32, 32)

] transform = transforms.Compose([
      transforms.Resize(IMAGE_SHAPE),
      transforms.ToTensor(),
      transforms.Lambda(lambda t: (t * 2) - 1),
  ])


  reverse_transform = transforms.Compose([
      transforms.Lambda(lambda t: (t + 1) / 2),
      transforms.Lambda(lambda t: t.permute(1, 2, 0)),
      transforms.Lambda(lambda t: t * 255.),
      transforms.Lambda(lambda t: t.cpu().numpy().astype(np.uint8)),
      transforms.ToPILImage(),
  ])
```

image_to_tensor : این تابع یک تصویر ورودی فرمت PIL.Imageرا به تنسور PyTorch تبدیل می‌کند.

img.convert('RGB'): تصویر به فضای رنگی RGB تبدیل می‌شود.مقادیر پیکسل‌ها از بازه [0, 255]به [0, 1]نرمال می‌شوند.

permute(2, 0, 1)تغییر ترتیب ابعاد از (height, width, channel) به.(channel, height, width)

unsqueeze(0)یک بعد اضافی به داده‌ها اضافه می‌کند که نشان‌دهنده تعداد نمونه‌ها (batch) است.

tensor_to_image_conversionاین تابع یک تنسور را به تصویر تبدیل می‌کند.

squeeze()حذف ابعاد اضافی. clip(0, 1)مقادیر خارج از محدوده [0, 1]را محدود می‌کندastype(np.uint8) .مقادیر پیکسل‌ها به مقادیر صحیح ۸ بیتی تبدیل می‌شوند.

Image.fromarrayآرایه‌ای از مقادیر پیکسل به تصویر PILتبدیل می‌شود.

```
image_to_tensor(img: Image.Image) -> torch.Tensor:
return torch.tensor(np.array(img.convert('RGB')) / 255.0).permute(2, 0, 1).unsqueeze(0) * 2 - 1


tensor_to_image_conversion(tensor: torch.Tensor) -> Image.Image:
image_array = np.array(((tensor.squeeze().permute(1, 2, 0) + 1) / 2).clip(0, 1) * 255).astype(np.uint
return Image.fromarray(image_array)


gather_tensor_values(constants: torch.Tensor, time: torch.Tensor) -> torch.Tensor:

gathered_values = constants.gather(-1, time)
return gathered_values.reshape(-1, 1, 1, 1)
```

این دستور مجموعه داده CIFAR-10 را که شامل تصاویر کوچک ۳۲x۳۲ از ۱۰ کلاس مختلف است، بارگذاری می‌کندImage.new('RGB')یک بوم جدید با اندازه و فضای رنگی مشخص ایجاد می‌کند.

در اینجا اندازه تصویر ۵ ستون و ۲ ردیف تعریف شده است.با استفاده از حلقه، تصاویر به ترتیب در مکان مناسب روی بوم قرار داده می‌شوند.

موقعیت تصویر بر اساس محاسبه (idx % 5)و (idx // 20)تعیین می‌شود. گرید تصویر به اندازه بزرگ‌تری تغییر داده می‌شود تا جزئیات واضح‌تر باشند.

```
cifar10_data = load_dataset('cifar10')

# Create a new image to display a grid of CIFAR-10 examples
grid_image = Image.new('RGB', size=(32 * 5, 32 * 2))

# Paste images into the grid
for idx in range(40):
    img = cifar10_data['train'][idx]['img']
    grid_image.paste(img, ((idx % 5) * 32,(idx // 20) * 32))

# Resize the image grid for better visibility
grid_image.resize((32 * 5 * 4, 32 * 2 * 4), Image.NEAREST)
```

کلاس DiffusionModelبرای مدل‌سازی فرآیند انتشار در یادگیری ماشین طراحی شده است. این فرآیند شامل افزودن نویز به داده‌ها در مراحل مختلف (*forward*) و بازسازی داده اولیه با حذف نویز در مراحل معکوس (*backward*) است. سازنده کلاس پارامترهای انتشار مانند تعداد مراحل زمانی، مقادیر نویزدهی betas و alphasو ضرب تجمعی آلفاها (alphas_cumprod) را محاسبه و تنظیم می‌کند. متد *forward*با استفاده از ترکیب مقیاس‌یافته داده اولیه و نویز، تصاویر نویزی تولید می‌کند. متد *backward*نویز را با کمک مدل تخمین‌گر حذف کرده و تصویر بازسازی‌شده را مرحله‌به‌مرحله تولید می‌کند. یک متد کمکی نیز برای انتخاب مقادیر مرتبط با هر مرحله زمانی وجود دارد که ساختار داده‌ها را برای محاسبات حفظ می‌کند. این کلاس به طور کلی هسته اصلی فرآیند انتشار در مدل‌های یادگیری عمیق است.

```python
class DiffusionModel:
    def __init__(self, start_schedule=0.0001, end_schedule=0.02, timesteps = 300):
        self.start_schedule = start_schedule
        self.end_schedule = end_schedule
        self.timesteps = timesteps
        self.betas = torch.linspace(start_schedule, end_schedule, timesteps)
        self.alphas = 1 - self.betas
        self.alphas_cumprod = torch.cumprod(self.alphas, axis=0)

    def forward(self, x_0, t, device):

        noise = torch.randn_like(x_0)
        sqrt_alphas_cumprod_t = self.get_index_from_list(self.alphas_cumprod.sqrt(), t, x_0.shape)
        sqrt_one_minus_alphas_cumprod_t = self.get_index_from_list(torch.sqrt(1. - self.alphas_cumprod

        mean = sqrt_alphas_cumprod_t.to(device) * x_0.to(device)
        variance = sqrt_one_minus_alphas_cumprod_t.to(device) * noise.to(device)

        return mean + variance, noise.to(device)

    @torch.no_grad()
    def backward(self, x, t, model, **kwargs):

        betas_t = self.get_index_from_list(self.betas, t, x.shape)
        sqrt_one_minus_alphas_cumprod_t = self.get_index_from_list(torch.sqrt(1. - self.alphas_cumprod
        sqrt_recip_alphas_t = self.get_index_from_list(torch.sqrt(1.0 / self.alphas), t, x.shape)
        mean = sqrt_recip_alphas_t * (x - betas_t * model(x, t, **kwargs) / sqrt_one_minus_alphas_cump
        posterior_variance_t = betas_t

        if t == 0:
```

```python
    if t == 0:
        return mean
    else:
        noise = torch.randn_like(x)
        variance = torch.sqrt(posterior_variance_t) * noise
        return mean + variance

@staticmethod
def get_index_from_list(values, t, x_shape):
    batch_size = t.shape[0]

    return result.reshape(batch_size, *((1,) * (len(x_shape) - 1))).to(t.device)
```

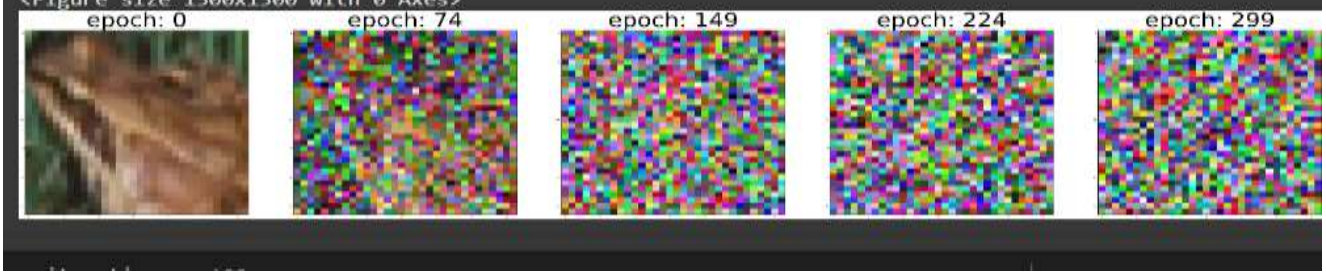
این بخش از کد مربوط به فرآیند نمایش تصاویر نویزی در مراحل مختلف فرآیند انتشار است. به عبارت دیگر، این کد نشان می‌دهد که چگونه یک تصویر خاص از مجموعه داده با گذر از مراحل انتشار، به تدریج نویزی می‌شود.

تصویر شماره ۵۷ از مجموعه داده CIFAR-10 انتخاب می‌شود و به عنوان ورودی اولیه استفاده می‌شود.تصویر انتخاب‌شده ۵ بار کپی می‌شود و به یک دسته (batch) تبدیل می‌شود.
پنج مرحله زمانی از صفر تا انتهای فرآیند انتشار انتخاب می‌شود که نشان‌دهنده شدت‌های مختلف نویزدهی هستند.
مدل انتشار به این تصاویر نویز اضافه می‌کند، به طوری که هر تصویر متناسب با مرحله زمانی خود نویزی متفاوت دارد.

تصاویر نویزی به همراه عنوانی که مرحله زمانی را نشان می‌دهد، در یک گرافیک بصری نمایش داده می‌شوند. تصاویر از تصویر اصلی با کمترین نویز تا تصویری که تقریباً کاملاً نویزی شده است مرتب شده‌اند.

```
NO_DISPLAY_IMAGES = 5
start_im = cifar10_data['train'][57]['img']

torch_image_batch = torch.stack([transform(start_im)] * NO_DISPLAY_IMAGES)

t = torch.linspace(0, diffusion_model.timesteps - 1, NO_DISPLAY_IMAGES).long()
noisy_image_batch, _ = diffusion_model.forward(torch_image_batch, t, device)

plt.figure(figsize=(15,15))
f, ax = plt.subplots(1, NO_DISPLAY_IMAGES, figsize = (100,100))

for idx, image in enumerate(noisy_image_batch):
    ax[idx].imshow(reverse_transform(image))
    ax[idx].set_title(f"epoch: {t[idx].item()}", fontsize = 100)
plt.show()
```

```
<Figure size 1500x1500 with 0 Axes>
```



کلاس SinusoidalPositionEmbeddingsاین کلاس برای تولید جاسازی‌های موقعیتی سینوسی از مقادیر زمانی طراحی شده است. جاسازی‌ها با استفاده از توابع سینوس و کسینوس تولید می‌شوند و به مدل کمک می‌کنند تا اطلاعات مربوط به زمان را در مراحل پردازش تصویر حفظ کند.

```
class SinusoidalPositionEmbeddings(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.dim = dim

    def forward(self, time):
        device = time.device
        half_dim = self.dim // 2
        embeddings = math.log(10000) / (half_dim - 1)
        embeddings = torch.exp(torch.arange(half_dim, device=device) * -embeddings)
        embeddings = time[:, None] * embeddings[None, :]
        embeddings = torch.cat((embeddings.sin(), embeddings.cos()), dim=-1)
        return embeddings
```

کلاس Blockاین کلاس یک واحد بلوکی اصلی برای مدل UNet است که شامل:

جاسازی زمانی با استفاده از کلاس قبلی.

لایه‌های کانولوشنی برای پردازش داده‌ها.

لایه‌های Batch Normalization و ReLU برای بهبود پایداری و یادگیری مدل.

پشتیبانی از اطلاعات برچسب (labels) برای افزایش انعطاف‌پذیری در کاربردهای خاص .بلوک‌ها به دو حالت طراحی شده‌اند : *downsample*کاهش ابعاد تصویر و *upsample*افزایش ابعاد تصویر، که برای مسیرهای پایین‌رونده و بالارونده در UNet استفاده می‌شوند.

```python
class Block(nn.Module):
    def __init__(self, channels_in, channels_out, time_embedding_dims, labels, num_filters = 3, downsa
        super().__init__()

        self.time_embedding_dims = time_embedding_dims
        self.time_embedding = SinusoidalPositionEmbeddings(time_embedding_dims)
        self.labels = labels
        if labels:
            self.label_mlp = nn.Linear(1, channels_out)

        self.downsample = downsample

        if downsample:
            self.conv1 = nn.Conv2d(channels_in, channels_out, num_filters, padding=1)
            self.final = nn.Conv2d(channels_out, channels_out, 4, 2, 1)
        else:
            self.conv1 = nn.Conv2d(2 * channels_in, channels_out, num_filters, padding=1)
            self.final = nn.ConvTranspose2d(channels_out, channels_out, 4, 2, 1)

        self.bnorm1 = nn.BatchNorm2d(channels_out)
        self.bnorm2 = nn.BatchNorm2d(channels_out)

        self.conv2 = nn.Conv2d(channels_out, channels_out, 3, padding=1)
        self.time_mlp = nn.Linear(time_embedding_dims, channels_out)
        self.relu = nn.ReLU()

    def forward(self, x, t, **kwargs):
        o = self.bnorm1(self.relu(self.conv1(x)))
        o_time = self.relu(self.time_mlp(self.time_embedding(t)))
        o = o + o_time[(..., ) + (None, ) * 2]
```

کلاس UNetاین کلاس ساختار اصلی UNet را پیاده‌سازی می‌کند که شامل:

مسیر پایین‌رونده با بلوک‌هایی که اطلاعات ویژگی‌ها را استخراج و ابعاد تصویر را کاهش می‌دهند.

مسیر بالارونده با بلوک‌هایی که ابعاد تصویر را بازیابی و ویژگی‌ها را ادغام می‌کنند.

لایه‌های کانولوشنی اولیه و نهایی برای تبدیل تصاویر ورودی و خروجی .مدل با استفاده از جاسازی‌های زمانی و اتصالات باقیمانده (residual connections)بین مسیرهای پایین‌رونده و بالارونده طراحی شده است تا اطلاعات مهم حفظ شود.

این ساختار، ترکیبی از طراحی کارآمد برای یادگیری ویژگی‌های عمیق و پردازش تصاویر با اطلاعات زمانی است که در مدل‌های تولیدی مانند انتشار یا بازسازی تصاویر به کار می‌رود.

```
class UNet(nn.Module):
    def __init__(self, img_channels = 3, time_embedding_dims = 128, labels = False, sequence_channels
        super().__init__()
        self.time_embedding_dims = time_embedding_dims
        sequence_channels_rev = reversed(sequence_channels)

        self.downsampling = nn.ModuleList([Block(channels_in, channels_out, time_embedding_dims, label
        self.upsampling = nn.ModuleList([Block(channels_in, channels_out, time_embedding_dims, labels,
        self.conv1 = nn.Conv2d(img_channels, sequence_channels[0], 3, padding=1)
        self.conv2 = nn.Conv2d(sequence_channels[0], img_channels, 1)


    def forward(self, x, t, **kwargs):
        residuals = []
        o = self.conv1(x)
        for ds in self.downsampling:
            o = ds(o, t, **kwargs)
            residuals.append(o)
        for us, res in zip(self.upsampling, reversed(residuals)):
            o = us(torch.cat((o, res), dim=1), t, **kwargs)

        return self.conv2(o)
```

این بخش کد مدل UNet را تعریف و برای آموزش آماده می‌کند. ابتدا مدل UNet بدون استفاده از برچسب‌ها (labels=False)ایجاد می‌شود. سپس مدل به دستگاه مناسب منتقل می‌شود تا محاسبات کارآمدتر انجام شوند. در نهایت، یک بهینه‌ساز Adam با نرخ یادگیری 0.001 تعریف می‌شود که وظیفه به‌روزرسانی وزن‌های مدل در طول فرآیند آموزش را بر عهده دارد. این تنظیمات گام اولیه برای شروع آموزش مدل هستند.

یک حلقه آموزش برای 1000 دوره (epoch) تعریف می‌کند که هدف آن بهینه‌سازی مدل UNet است. در هر دوره:

تصویر اولیه (start_im) با استفاده از یک تبدیل از پیش تعریف‌شده آماده و به یک دسته (batch) از 128 تصویر تکراری تبدیل می‌شود.زمان‌های تصادفی (t) برای افزودن نویز به تصاویر نمونه‌برداری و به دستگاه مناسب GPU) یا (CPU منتقل می‌شوند.مدل انتشار (Diffusion Model) برای تولید تصاویر نویزی و نویز واقعی به کار گرفته می‌شود.مدل UNet تلاش می‌کند نویز پیش‌بینی‌شده را برای تصاویر نویزی تولید کند.

از تابع زیان MSE برای محاسبه تفاوت بین نویز واقعی و پیش‌بینی‌شده استفاده شده و مقدار زیان محاسبه می‌شود.زیان محاسبه‌شده برای به‌روزرسانی وزن‌های مدل با استفاده از الگوریتم Adam استفاده می‌شود.

هر چند دوره یک‌بار بر اساس PRINT_FREQUENCYمقدار زیان میانگین چاپ می‌شود تا پیشرفت آموزش قابل پیگیری باشد.

```python
unet = UNet(labels=False)
unet.to(device)
optimizer = torch.optim.Adam(unet.parameters(), lr=0.001)

for epoch in range(1000):
    mean_epoch_loss = []

    # Apply transform to start_im and then stack
    transformed_image = transform(start_im)
    batch = torch.stack([transformed_image] * 128)

    t = torch.randint(0, diffusion_model.timesteps, (128,)).long().to(device)

    batch_noisy, noise = diffusion_model.forward(batch, t, device)
    predicted_noise = unet(batch_noisy, t)

    optimizer.zero_grad()
    loss = torch.nn.functional.mse_loss(noise, predicted_noise)
    mean_epoch_loss.append(loss.item())
    loss.backward()
    optimizer.step()

    if epoch % PRINT_FREQUENCY == 0:
        print('---')
        print(f"Epoch: {epoch} | Train Loss {np.mean(mean_epoch_loss)}")
```

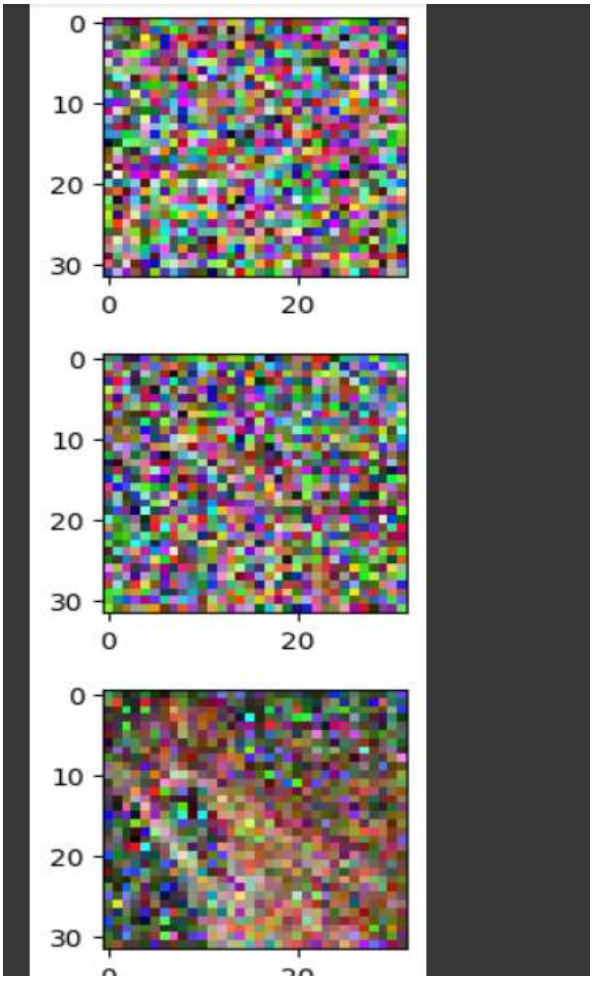
این کد فرآیند تولید تصویر از نویز تصادفی را بدون به‌روزرسانی وزن‌ها با torch.no_gradانجام می‌دهد: یک تصویر نویزی اولیه با ابعاد مشخص و مقادیر تصادفی تولید شده و به دستگاه GPU یا CPU منتقل می‌شود.

یک حلقه معکوس از مراحل انتشار (diffusion) اجرا می‌شود که از زمان آخرین مرحله به اولین مرحله برمی‌گردد.در هر مرحله، مدل انتشار (Diffusion Model) و مدل UNet برای کاهش نویز و بازسازی تصویر استفاده می‌شوند.

هر 50 مرحله، تصویر فعلی با استفاده از تبدیل معکوس (reverse_transform) به نمایش درمی‌آید تا پیشرفت فرآیند دیده شود.

هدف این بخش، بازسازی یک تصویر تمیز از نویز خالص است .

```python
with torch.no_grad():
    img = torch.randn((1, 3) + IMAGE_SHAPE).to(device)
    for i in reversed(range(diffusion_model.timesteps)):
        t = torch.full((1,), i, dtype=torch.long, device=device)
        img = diffusion_model.backward(img, t, unet.eval())
        if i % 50 == 0:
            plt.figure(figsize=(2,2))
            plt.imshow(reverse_transform(img[0]))
            plt.show()
```



در این بخش، فرآیند آموزش مدل درون یک حلقه انجام می‌شود و برخی اصلاحات برای پردازش دسته‌ای (batch processing)صورت گرفته است:

اندازه دسته داده‌ها (BATCH_SIZE) برابر 128 تنظیم شده است و تعداد دفعات چاپ وضعیت (PRINT_FREQUENCY) به 100 تنظیم شده است.

برای هر اپوک، یک حلقه برای پردازش دسته‌های داده (که از trainloaderگرفته می‌شود) آغاز می‌شود.داده‌ها و برچسب‌ها (هدف‌ها) به دستگاه CPU) یا (GPU منتقل می‌شوند.

در هر دسته، ابتدا نویز به داده‌ها اضافه می‌شود و سپس مدل UNet برای پیش‌بینی نویز از تصویر نویزی استفاده می‌شود. تابع خسارت (loss) به‌وسیله خطای میانگین مربعات (MSE) محاسبه می‌شود. گرادیان‌ها محاسبه و وزن‌های مدل با استفاده از الگوریتم Adamبه‌روز می‌شوند.

از tqdmبرای نمایش پیشرفت آموزش استفاده می‌شود. وضعیت حلقه و میانگین خسارت آموزش برای هر دسته به‌روزرسانی می‌شود تا پیشرفت مدل نشان داده شود.

در نهایت، هدف این بخش آموزش مدل با استفاده از دسته‌های داده و به‌روزرسانی پارامترهای مدل است.

```
BATCH_SIZE = 128
PRINT_FREQUENCY = 100

# Training loop (modified for batch processing)
for epoch in range(22):
    mean_epoch_loss = []
    avg_tr_loss = 0
    loop_train = tqdm(enumerate(data_loader, 1), total=len(data_loader), desc="Train", position=0, lea

    for batch_idx, (data, target) in enumerate(trainloader):
        data, target = data.to(device), target.to(device)
        t = torch.randint(0, diffusion_model.timesteps, (BATCH_SIZE,)).long().to(device)
        batch_noisy, noise = diffusion_model.forward(data, t, device)
        predicted_noise = unet(batch_noisy, t)
        optimizer.zero_grad()
        loss = torch.nn.functional.mse_loss(noise, predicted_noise)
        mean_epoch_loss.append(loss.item())
        avg_tr_loss += loss.item()
        loss.backward()
        optimizer.step()

        loop_train.set_description(f"Train - iteration : {epoch}")
        loop_train.set_postfix(
          avg_train_loss="{:.4f}".format(avg_tr_loss / index),
          refresh=True,
      )
```

در این بخش، تصاویری که از مدل استخراج شده‌اند به صورت یک گالری تصویر در یک تصویر بزرگ‌تر ترکیب می‌شوند:

یک حلقه برای 200 تصویر اجرا می‌شود و هر تصویر از x[i]گرفته می‌شود. هر تصویر به صورت تک بُعدی (unsqueeze) تبدیل و به tensor_imageیک تابع برای تبدیل تنسور به تصویر ارسال می‌شود و به لیست imsافزوده می‌شود. یک تصویر جدید با اندازه 32 *10در 32 *10پیکسل ساخته می‌شود (این ابعاد به عنوان گرید برای تصاویر استفاده می‌شوند).

هر تصویر به مکان مناسب خود در این گرید 10 *10با استفاده از pasteقرار داده می‌شود.

هدف این بخش ایجاد یک گالری تصویری است که تصاویری که توسط مدل تولید شده‌اند را به صورت یک شبکه 10 x10در کنار هم قرار دهد

```
ims = []
for i in range(200):
    ims.append(tensor_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%5)*32, 32*(i//10)))
image.resize((32*4*10, 32*4*10), Image.NEAREST)
```