تمرین دوم

نام درس: یادگیری عمیق

استاد درس: دکتر محمدرضا محمدی

نام: محمد حقیقت

شماره دانشجویی: 403722042

گرایش: هوش مصنوعی

دانشکده: مهندسی کامپیوتر

نیم سال دوم 1403-1404

سوال چهارم

آ)

محل پیاده‌سازی هدایت بدون دسته بند (Classifier-Free Guidance - CFG) در ریپازیتوری گیتهاب

پیاده‌سازی این تکنیک بین منطق آموزش (برای یادگیری مدل) و منطق نمونه‌برداری (برای اعمال هدایت) تقسیم شده است.

این کدها عمدتا در نسخه شرطی (conditional) ریپازیتوری قرار دارند.

در بخش آموزش: حذف تصادفی برچسب (Conditional Dropout)

فایل: ddpm_conditional.py

تابع: train()

در این بخش، مدل یاد می‌گیرد که با نبودِ برچسب نیز کار کند.

کد:

```
if np.random.random() < 0.1:
    labels = None

predicted_noise = model(x_t, t, labels)
```

در ۱۰٪ از مراحل آموزش، متغیر labels با مقدار None جایگزین می‌شود. به این ترتیب، مدل هم با برچسب‌های واقعی و هم با ورودی بدون برچسب تغذیه می‌شود و پیش‌نیاز اصلی CFG را فرا می‌گیرد.

در بخش نمونه‌برداری (sampling): اعمال فرمول هدایت

اینجا همان جایی است که فرمول CFG در عمل هنگام تولید تصویر پیاده‌سازی می‌شود.

فایل: ddpm_conditional.py

تابع: sample()

کد:

```python
def sample(self, model, n, labels, cfg_scale=3):
    ...
    for i in tqdm(reversed(range(1, self.noise_steps)), position=0):
        t = (torch.ones(n) * i).long().to(self.device)

        # 1. Get the conditional prediction
        predicted_noise = model(x, t, labels)

        # 2. If guidance is on, get the unconditional prediction and combine them
        if cfg_scale > 0:
            uncond_predicted_noise = model(x, t, None)
            predicted_noise = torch.lerp(uncond_predicted_noise, predicted_noise, cfg_scale)

        # 3. Use the final 'predicted_noise' for the standard denoising step
```

دریافت پیش‌بینی شرطی

اگر هدایت فعال باشد، پیش‌بینی غیرشرطی نیز دریافت و با هم ترکیب می‌شوند

استفاده از 'predicted_noise' نهایی در گام استاندارد نویز زدایی

ابتدا مدل با labels مشخص فراخوانی می‌شود تا پیش‌بینی شرطی به دست آید.

سپس، برای بار دوم با labels=None فراخوانی می‌شود تا پیش‌بینی غیرشرطی حاصل شود.

دستور torch.lerp(...) به شکلی هوشمندانه فرمول CFG را پیاده‌سازی می‌کند. این تابع کار درون‌یابی خطی را انجام می‌دهد ((start + weight * (end - start)) که از نظر ریاضی معادل همان فرمول CFG است.

ب)

روش اعمال شرط در این ریپازیتوری

در این پروژه، مدل برای تولید تصاویر از مجموعه داده CIFAR-10 آموزش داده می‌شود. شرط ما در اینجا کلاس تصویر است (مثلاً: کلاس ۰ برای هواپیما، ۱ برای ماشین، و غیره). روشی که در این کد برای اعمال این شرط به کار رفته، ترکیبی از Embedding و جمع برداری است.

مراحل کار به این صورت است:

تبدیل کلاس به یک بردار (Embedding): مدل نمی‌تواند عدد خام کلاس (مثلاً عدد ۵ برای "سگ") را مستقیماً درک کند. به همین دلیل، ابتدا این عدد به یک بردار عددی معنادار به نام "Embedding"

تبدیل می‌شود. این کار شبیه این است که برای هر کلاس، یک "اثر انگشت دیجیتالی" منحصربه‌فرد بسازیم. این بردار، اطلاعات مربوط به آن کلاس را در خود دارد.

تبدیل مرحله زمانی به بردار (Time Embedding):

همانطور که می‌دانید، مدل دیفیوژن در هر مرحله از حذف نویز، باید بداند در کدام "مرحله زمانی" (t) قرار دارد. این عدد t نیز به یک بردار Embedding جداگانه تبدیل می‌شود تا اطلاعات مربوط به میزان نویز موجود در تصویر را به مدل بدهد.

ترکیب اطلاعات (جمع کردن بردارها):

اینجا بخش کلیدی ماجراست. برای اینکه مدل همزمان از کلاس هدف و مرحله زمانی آگاه باشد، این دو بردار با هم جمع می‌شوند:

امبدینگ_نهایی = امبدینگ_کلاس + امبدینگ_زمان

بردار نهایی حاصل، حاوی اطلاعات ترکیبی است: هم به مدل می‌گوید که "در چه مرحله‌ای از فرآیند هستیم" و هم به او یادآوری می‌کند که "هدف نهایی ساختن یک تصویر از فلان کلاس است".

تزریق اطلاعات به مدل U-Net:

این بردار ترکیبی نهایی به لایه‌های مختلف مدل U-Net (که وظیفه پیش‌بینی نویز را دارد) تزریق می‌شود. به طور مشخص، در این کد، این بردار به بلوک‌های SelfAttention در معماری U-Net داده می‌شود. این کار به مدل اجازه می‌دهد تا در حین پردازش تصویر نویزی، توجه خود را روی ویژگی‌هایی متمرکز کند که به کلاس هدف مرتبط هستند.

آیا این تنها روش برای اعمال شرط است؟

خیر، این فقط یکی از روش‌های ممکن است. روش‌های قدرتمند و متنوع دیگری نیز وجود دارند که بسته به نوع شرط (متن، تصویر دیگر، نقشه سگمنتیشن) و پیچیدگی مدل استفاده می‌شوند. در ادامه به چند روش مهم دیگر اشاره می‌کنم:

الحاق یا چسباندن (Concatenation)

در این روش، شرط مستقیما به ورودی مدل "چسبانده" می‌شود.

حالا چطور کار می‌کند؟ فرض کنید تصویر ورودی شما ۳ کانال رنگی (RGB) دارد. می‌توانیم یک یا چند کانال جدید به آن اضافه کنیم که اطلاعات شرط را در خود دارند. مثلاً اگر شرط ما یک نقشه

سگمنتیشن باشد، آن نقشه را به عنوان یک کانال جدید به تصویر ورودی اضافه کرده و یک ورودی ۴ کاناله به مدل می‌دهیم.

مثال: در مدل‌های Image-to-Image، تصویر شرط به عنوان یک کانال اضافی به تصویر نویزی متصل می‌شود.

مزیت: روشی ساده و مستقیم، به خصوص برای شرط‌هایی که ساختار فضایی دارند (مثل نقشه).

توجه متقاطع (Cross-Attention)

این روش یکی از قدرتمندترین و رایج‌ترین تکنیک‌هاست، به خصوص در مدل‌های تبدیل متن به تصویر مانند Stable Diffusion و DALL-E.

حالا چطور کار می‌کند؟ در این روش، شرط (مثلاً یک جمله متنی) ابتدا به مجموعه‌ای از بردارها (توکن‌ها) تبدیل می‌شود. سپس در لایه‌های مختلف مدل U-Net، یک مکانیزم "توجه" (Attention) پیاده‌سازی می‌شود که به مدل اجازه می‌دهد به بخش‌های مختلف بردار شرط "نگاه کند". مدل یاد می‌گیرد که برای تولید هر قسمت از تصویر، به کدام کلمات یا مفاهیم در متن شرط بیشتر توجه کند.

مثال: وقتی مدل Stable Diffusion می‌خواهد "یک فضانورد در حال اسب‌سواری" را نقاشی کند، مکانیزم Cross-Attention به آن کمک می‌کند تا ویژگی‌های "فضانورد" را با ویژگی‌های "اسب" ترکیب کرده و در جای مناسب قرار دهد.

مزیت: بسیار انعطاف‌پذیر و قدرتمند است و برای شرط‌های پیچیده و طولانی مانند متن عالی عمل می‌کند.

نرمال‌سازی لایه‌ای تطبیقی (Adaptive Layer Normalization - AdaLN)

این یک روش ظریف‌تر است که در آن، شرط مستقیما وارد داده‌ها نمی‌شود، بلکه رفتار خود مدل را کنترل می‌کند.

چطور کار می‌کند؟ بردار شرط (مثلاً Embedding کلاس) برای پیش‌بینی پارامترهای scale و shift در لایه‌های نرمال‌سازی (مانند LayerNorm یا BatchNorm) درون U-Net استفاده می‌شود. با تغییر این پارامترها، شرط می‌تواند به صورت غیرمستقیم بر فعال‌سازی‌های نورون‌ها در سراسر شبکه تأثیر بگذارد و فرآیند تولید را به سمت هدف هدایت کند.

مثال: این تکنیک در مدل‌های معروفی مانند StyleGAN و DiT (Diffusion Transformers) به کار رفته است.

مزیت: به مدل اجازه می‌دهد تا شرط را به شیوه‌ای عمیق‌تر و بنیادی‌تر در فرآیند تولید ادغام کند.

ج)

برای این کار، ماژول‌های اصلی کد باید به شکل زیر تغییر کنند

معرفی یک انکودر متن (Text Encoder): برای این کار به یک جز حیاتی جدید نیاز داریم: یک انکودر متن از پیش‌آموزش‌دیده (مانند انکودر مدل CLIP) که بتواند متن را به بردار عددی (Embedding) تبدیل کند. این انکودر آموزش داده نمی‌شود و فقط برای استخراج ویژگی از متن استفاده می‌شود.

تغییر در ماژول modules.py (مدل U-Net): مدل U-Net باید به جای دریافت شناسه‌ی کلاس (یک عدد)، یک بردار امبدینگ متن را به عنوان ورودی بپذیرد. در نتیجه، لایه nn.Embedding مخصوص کلاس‌ها حذف خواهد شد.

تغییر در ماژول ddpm.py (کلاس اصلی دیفیوژن): این کلاس وظیفه مدیریت انکودر متن و اجرای منطق CFG را بر عهده می‌گیرد. یعنی در هر مرحله، دو بار از مدل U-Net استفاده می‌کند: یک بار با امبدینگ متن اصلی (شرطی) و یک بار با امبدینگ متن خالی (غیرشرطی).

تغییر در اسکریپت cifar.py (اسکریپت آموزش و نمونه‌سازی): این اسکریپت بیشترین تغییر را خواهد داشت. باید بتواند با یک مجموعه داده متنی (شامل جفت‌های (تصویر, کپشن)) کار کند، از یک توکنایزر (Tokenizer) استفاده کند و به جای برچسب کلاس، پرامپت‌های متنی را به مدل بدهد.

جزئیات تغییرات در کد

بخش جدید: انکودر متن

ابتدا باید یک انکودر و توکنایزر متن مانند CLIP را به پروژه اضافه کنیم. این کار معمولا در اسکریپت اصلی آموزش (cifar.py) انجام می‌شود.

```
# This would be initialized in the main training script (cifar.py)
from transformers import CLIPTextModel, CLIPTokenizer

# Load pre-trained model and tokenizer
text_encoder = CLIPTextModel.from_pretrained("openai/clip-vit-large-patch14").cuda().eval()
tokenizer = CLIPTokenizer.from_pretrained("openai/clip-vit-large-patch14")

# Freeze its weights, we don't train it
for param in text_encoder.parameters():
    param.requires_grad = False
```

تغییر در ماژول modules.py (مدل U-Net)

مدل U-Net دیگر نیازی به ساختن امبدینگ از روی شناسه کلاس ندارد و باید یک امبدینگ از پیش ساخته شده را دریافت کند.

در حالت فعلی (Unet.forward):

مدل خودش از روی شناسه کلاس (labels) یک امبدینگ می‌سازد.

```
# modules.py (Unet class)
class Unet(nn.Module):
    def __init__(self, c_in=3, c_out=3, time_dim=256, num_classes=None, **kwargs):
        ...
        if num_classes is not None:
            self.label_emb = nn.Embedding(num_classes, time_dim)
        ...

    def forward(self, x, t, labels=None):
        ...
        # Time embedding
        t = self.pos_encoding(t, self.time_dim)

        if self.label_emb is not None:
            if labels is None:
                # Set to zeros for unconditional training
                labels = torch.zeros(x.shape[0], dtype=torch.long).to(x.device)
            label_emb = self.label_emb(labels)
            t = t + label_emb # Additive conditioning
```

پس از تغییر (Unet.forward):

مدل یک بردار امبدینگ آماده به نام context_emb را به عنوان ورودی می‌پذیرد.

```python
# modules.py (Unet class)
class Unet(nn.Module):
    # num_classes is no longer needed. We might want context_dim instead.
    def __init__(self, c_in=3, c_out=3, time_dim=256, context_dim=768, **kwargs):
        super().__init__()
        # self.label_emb is REMOVED.
        # We need a linear layer to project context embedding to the time dimension
        self.context_proj = nn.Linear(context_dim, time_dim)
        ...

    # The signature changes from 'labels' to 'context_emb'
    def forward(self, x, t, context_emb=None):
        ...
        # Time embedding
        t = self.pos_encoding(t, self.time_dim)

        if context_emb is not None:
            # Project the text embedding and add it to the time embedding
            context_emb = self.context_proj(context_emb)
            t = t + context_emb # Additive conditioning
```

تغییر در ماژول ddpm.py (کلاس اصلی دیفیوژن)

این ماژول منطق CFG را با استفاده از انکودر متن پیاده‌سازی می‌کند. متد sample آن به شکل زیر تغییر می‌کند:

در حالت فعلی (DDPM.sample):

مدل را با labels و None فراخوانی می‌کند.

```python
# ddpm.py
def sample(self, n, labels, cfg_scale=3):
    ...
    for i in tqdm(reversed(range(1, self.noise_steps)), position=0):
        t = (torch.ones(n) * i).long().to(self.device)
        predicted_noise = self.model(x, t, labels) # Conditional
        if cfg_scale > 0:
            uncond_predicted_noise = self.model(x, t, None) # Unconditional
            predicted_noise = torch.lerp(uncond_predicted_noise, predicted_noise, cfg_scale)
```

پس از تغییر (DDPM.sample):

پرامپت‌های متنی را دریافت کرده، آن‌ها را انکود می‌کند و امبدینگ‌ها را به مدل می‌دهد.

```python
# ddpm.py
class DDPM:
    # We add the text encoder and tokenizer as members
    def __init__(self, ..., text_encoder, tokenizer):
        ...
        self.text_encoder = text_encoder
        self.tokenizer = tokenizer

    # The sample signature changes from labels to prompts (list of strings)
    def sample(self, n, prompts, cfg_scale=3):
        logging.info(f"Sampling with prompts: {prompts}...")
        self.model.eval()
        with torch.no_grad():
            x = torch.randn((n, 3, self.img_size, self.img_size)).to(self.device)

            # Your pseudo-code implemented here:
            # 1. Prepare prompts for conditional and unconditional passes
            uncond_prompts = [""] * n

            # 2. Tokenize and encode
            # Conditional embeddings
            cond_inputs = self.tokenizer(prompts, padding="max_length", max_length=self.tokenizer.model_max_length, truncation=True, return_tensors="pt")
            cond_embeddings = self.text_encoder(cond_inputs.input_ids.to(self.device)).pooler_output
```

```python
            # Unconditional embeddings
            uncond_inputs = self.tokenizer(uncond_prompts, padding="max_length", max_length=self.tokenizer.model_max_length, truncation=True,
return_tensors="pt")
            uncond_embeddings = self.text_encoder(uncond_inputs.input_ids.to(self.device)).pooler_output

            for i in tqdm(reversed(range(1, self.noise_steps)), position=0):
                t = (torch.ones(n) * i).long().to(self.device)

                # Predict noise for both conditional and unconditional
                cond_predicted_noise = self.model(x, t, cond_embeddings)
                uncond_predicted_noise = self.model(x, t, uncond_embeddings)

                # Apply CFG
                predicted_noise = torch.lerp(uncond_predicted_noise, cond_predicted_noise, cfg_scale)

                # Denoising step (same as before)
                ...
        ...
        return x
```

تغییر در اسکریپت cifar.py (اسکریپت آموزش)

این اسکریپت باید برای کار با داده‌های متنی اصلاح شود و منطق "حذف شرط" را برای متن پیاده‌سازی کند.

در حالت فعلی (حلقه آموزش):

از برچسب‌های عددی استفاده کرده و به صورت تصادفی آن‌ها را None قرار می‌دهد.

```
# cifar.py (train function)
for i, (images, labels) in enumerate(pbar):
    ...
    if np.random.random() < 0.1:
        labels = None # Conditional dropout
    predicted_noise = ddpm.model(x_t, t, labels)
    ...
```

پس از تغییر (حلقه آموزش):

از پرامپت‌های متنی استفاده کرده و برای حذف شرط، به صورت تصادفی آن‌ها را با یک رشته خالی ("") جایگزین می‌کند.

```
# Let's rename it to train_text_to_image.py
# Assume you have a dataloader that yields (images, prompts)
# Also assume text_encoder and tokenizer are initialized as shown in step 1.

# Inside the train function:
# Pass text_encoder and tokenizer to the DDPM constructor
ddpm = DDPM(..., text_encoder=text_encoder, tokenizer=tokenizer)

for i, (images, prompts) in enumerate(pbar):
    ...
    # Conditional dropout for text
    for j in range(len(prompts)):
        if np.random.random() < 0.1:
            prompts[j] = "" # Replace with empty string for unconditional training

    # Tokenize and get embeddings
    inputs = tokenizer(prompts, padding="max_length", max_length=tokenizer.model_max_length, truncation=True, return_tensors="pt")
    embeddings = text_encoder(inputs.input_ids.to(device)).pooler_output

    # The forward pass in DDPM needs to be updated to accept embeddings
    # (let's assume ddpm.forward(images, context) is the new signature)
    loss = ddpm(images, context=embeddings)
```

د)

Stable Diffusion یک مدل واحد نیست، بلکه یک سیستم هوشمندانه متشکل از سه جزء اصلی است که برای کارایی و قدرت بسیار بالا طراحی شده. این مدل به خانواده‌ای از مدل‌ها به نام Latent Diffusion Models (مدل‌های دیفیوژن در فضای نهان) تعلق دارد.

نوآوری کلیدی آن این است که فرآیند دیفیوژن (حذف نویز) به جای اینکه روی خود تصاویر بزرگ انجام شود، روی یک نسخه بسیار کوچک‌تر و فشرده‌شده از تصویر به نام "فضای نهان" (Latent Space) اتفاق می‌افتد.

این سه جزء اصلی عبارتند از:

یک مدل Autoencoder (از نوع VAE): این مدل مانند یک ابزار فشرده‌سازی و بازگشایی فوق پیشرفته برای تصاویر عمل می‌کند.

انکودر (Encoder): یک تصویر با وضوح بالا (مثلا 512x512 پیکسل) را گرفته و آن را به یک نمایش نهان کوچک (مثلاً 64x64) فشرده می‌کند. این فضای نهان همچنان تمام اطلاعات ضروری تصویر را در خود حفظ می‌کند.

دیکودر (Decoder): یک نمایش نهان را گرفته و آن را دوباره به یک تصویر با وضوح بالا تبدیل می‌کند.

یک انکودر متن (Text Encoder مدل CLIP): تنها وظیفه این مدل، درک مفهوم پرامپت متنی است. یک رشته متنی (مثلاً "یک قلعه باشکوه در کوهستان") را گرفته و آن را به یک بردار عددی غنی (امبدینگ) تبدیل می‌کند که معنای کلمات را در خود دارد.

مدل U-Net (مدل اصلی دیفیوژن): این قلب سیستم است، اما با دو تفاوت اساسی نسبت به مدلی که در ریپازیتوری وجود دارد:

در فضای نهان کار می‌کند: این مدل یاد می‌گیرد که نویز را از روی نمایش نهان کوچک حذف کند، نه از روی تصویر پیکسلی بزرگ. این کار از نظر محاسباتی بسیار بهینه‌تر است.

از Cross-Attention برای شرط‌گذاری استفاده می‌کند: به جای اینکه فقط امبدینگ متن را به امبدینگ زمان اضافه کند، از یک مکانیزم قدرتمند به نام توجه متقاطع (Cross-Attention) استفاده می‌کند. این مکانیزم به U-Net اجازه می‌دهد تا در هر مرحله، برای هدایت فرآیند تولید تصویر، به بخش‌های خاصی از پرامپت متنی "نگاه کند". برای مثال، هنگام ساختن بخش قلعه در فضای نهان، می‌تواند به کلمه "قلعه" در متن توجه بیشتری کند.

فرآیند کامل به این شکل است:

آموزش: یک تصویر توسط VAE به فضای نهان انکود می‌شود. به این فضای نهان نویز اضافه می‌شود. سپس مدل U-Net یاد می‌گیرد که با راهنمایی امبدینگ متن (از طریق Cross-Attention)، این نویز را پیش‌بینی کند.

تولید تصویر (Inference):

با یک فضای نهان کاملاً تصادفی (پر از نویز) شروع می‌کنیم.

با استفاده از U-Net و امبدینگ متن از CLIP، این فضای نهان را مرحله به مرحله نویززدایی می‌کنیم (این همان فرآیند دیفیوژن با راهنمایی CFG است).

پس از پایان نویززدایی، فضای نهان تمیز و نهایی را به دیکودر VAE می‌دهیم تا آن را به یک تصویر جدید و با وضوح بالا تبدیل کند.

خب حالا باید برای تبدیل شدن به Stable Diffusion کد را تغییر بدهیم

اجزای جدیدی که باید اضافه شوند

شما باید دو مدل جدید و از پیش‌آموزش‌دیده را به پروژه اضافه کنید. این مدل‌ها در طول فرآیند آموزش دیفیوژن، آموزش داده نمی‌شوند و وزن‌هایشان ثابت می‌ماند.

یک VAE (Variational Autoencoder): برای انکود و دیکود کردن تصاویر به/از فضای نهان.

```
# To be added in the main script (e.g., 'train_sd.py')
from diffusers import AutoencoderKL

vae = AutoencoderKL.from_pretrained("runwayml/stable-diffusion-v1-5", subfolder="vae").cuda()
# Freeze the VAE
vae.requires_grad_(False)
```

یک انکودر متن CLIP: برای پردازش پرامپت‌های متنی.

```
# To be added in the main script
from transformers import CLIPTextModel, CLIPTokenizer

text_encoder = CLIPTextModel.from_pretrained("openai/clip-vit-large-patch14").cuda()
tokenizer = CLIPTokenizer.from_pretrained("openai/clip-vit-large-patch14")
# Freeze the text encoder
text_encoder.requires_grad_(False)
```

تغییرات در ماژول modules.py (مدل U-Net)

این ماژول به بزرگترین تغییرات معماری نیاز دارد.

قبل (U-Net ساده):

- روی تصاویر ۳ کاناله RGB کار می‌کند.
- شرط‌گذاری با جمع ساده امبدینگ کلاس انجام می‌شود.
- فقط دارای بلوک‌های SelfAttention است.

بعد (U-Net مدل Stable Diffusion):

تغییر کانال‌های ورودی/خروجی: U-Net اکنون روی فضای نهان ۴ کاناله که توسط VAE تولید شده کار خواهد کرد.

در سازنده Unet، مقادیر c_in و c_out باید از 3 به 4 تغییر کنند.

استفاده از Cross-Attention برای شرط‌گذاری: باید لایه‌های CrossAttention را به معماری اضافه کنیم. این لایه‌ها حالت داخلی U-Net و امبدینگ متن (context_emb) را به عنوان ورودی می‌گیرند.

باید یک لایه CrossAttention تعریف کنیم (یا از کتابخانه‌ها وارد کنید).

در متد forward کلاس Unet، ورودی به forward(self, x, t, context_emb=None) تغییر می‌کند.

جمع ساده t = t + label_emb به طور کامل حذف می‌شود.

امبدینگ context_emb به بلوک‌های ترانسفورمر جدیدی در U-Net پاس داده می‌شود که هم شامل SelfAttention و هم CrossAttention هستند.

تغییرات در ماژول ddpm.py (کلاس اصلی دیفیوژن)

منطق این کلاس از نظر مفهومی مشابه باقی می‌ماند (مدیریت فرآیند نویززدایی)، اما اکنون به طور کامل در فضای نهان عمل می‌کند.

قبل:

- متد forward یک دسته تصویر را به عنوان ورودی می‌گرفت.
- متد sample مستقیماً تصاویر پیکسلی تولید می‌کرد.

بعد:

- متد forward باید به forward(self, latents, context_emb) تغییر نام دهد و فضای نهان نویزی و امبدینگ متن را بگیرد.
- متد sample دقیقاً همان منطق CFG که در پاسخ قبلی توضیح داده شد را اجرا می‌کند (فراخوانی مدل با امبدینگ‌های شرطی و غیرشرطی)، اما متغیر x که در هر مرحله به‌روزرسانی می‌شود، یک فضای نهان است، نه یک تصویر. نویز تصادفی اولیه نیز در ابعاد فضای نهان ((n, 4, 64, 64)) ایجاد می‌شود.
- خروجی نهایی متد sample یک فضای نهان نویززدایی شده خواهد بود، نه یک تصویر.

تغییرات در اسکریپت cifar.py (اسکریپت آموزش و تولید)

این اسکریپت به مدیر اصلی پروژه تبدیل می‌شود که هر سه جزء را به هم متصل می‌کند. نام آن را به train_sd.py تغییر می‌دهیم.

قبل (حلقه آموزش):

- بارگذاری (تصویر, برچسب).
- ارسال به ddpm.forward(تصویر).

بعد (حلقه آموزش در train_sd.py):

- بارگذاری (تصویر, پرامپت) از دیتا لودر.
- انکود کردن تصویر: latents = vae.encode(image).latent_dist.sample().
- انکود کردن متن: context_emb = text_encoder(tokenizer(prompt, ...)).last_hidden_state
- اجرای گام دیفیوژن: loss = ddpm.forward(latents, context_emb).

قبل (نمونه‌سازی):

- فراخوانی ddpm.sample().
- ذخیره تصاویر حاصل.

بعد (نمونه‌سازی در train_sd.py):

- دریافت پرامپت متنی از کاربر.
- فراخوانی ddpm.sample() برای دریافت فضای نهان نهایی و نویززدایی شده:
  denoised_latent = ddpm.sample(prompts=[prompt], ...)
- دیکود کردن فضای نهان: final_image = vae.decode(denoised_latent).sample.
- پردازش نهایی و ذخیره final_image.

برای رفع برخی ایرادات و ابهامات از AI استفاده شده است.