گزارش پیاده‌سازی ReNO

ابتدا توضیحی مختصر درمورد مقاله و نحوه پیاده سازی آن داده، سپس بخش های مختلف کد را توضیح داده و درنهایت نمونه‌ای خروجی از آن را نشان می‌دهیم. تمامی کدهای این گزارش در یک jupyter notebook به نام ReNO.ipnyb قرار دارد.

**نکته: در فولدر pics تمام تصاویر مربوط به prompt ها به همراه یک فایل txt تحلیل آنها وجود دارد. در انتهای این گزارش هم جمع بندی مختصری در رابطه با مقایسه CLIP و BLIP آمده است.

توضیح مقاله

در این مقاله به یک روش که در زمان inference کار می‌کند برای بهبود عملکرد مدل های text to image طراحی شده. ایده این روش بر این اساس است که ماتریس نویز که به مدل داده می‌شود تا از روی آن تصویر مربوط به prompt را بنویسد، معمولا کاملا رندوم است و خیلی اوقات نمی‌توان از روی آن تصویر مطابق با prompt را ساخت. این موضوع در prompt هایی که اشیای مختلف و با صفت‌های مختلف دارد مشهود است. برای همین در این روش، ابتدا با کمک یک مدل از پیش آموزش دیده (pre-trained) تصویر با ماتریس نویز اولیه کاملا رندوم ساخته شده، سپس یک loss function با کمک یک VLM طراحی می‌شود که وظیفه‌اش پیدا کردن میزان شباهت feature های prompt و تصویر ساخته شده است، و هدف پیدا کردن ماتریس نویز اولیه‌ای است که این loss function را مینیمم می‌کند. در این پیاده سازی، از stable diffusion turbo به عنوان مدل pre-trained و از CLIP به عنوان VLMی که feature های prompt و تصویر ساخته شده را مینیمم می‌کند استفاده شده است.

توضیح بخش‌های مختلف کد

ابتدا loss function را با کمک CLIP تعریف می‌کنیم:

```python
def compute_clip_loss(clip_model, tokenizer, prompt, image):
    image_features = clip_model.get_image_features(image)

    prompt_token = tokenizer(
        prompt, return_tensors="pt", padding=True, max_length=77, truncation=True
    ).to(image.device)
    text_features = clip_model.get_text_features(**prompt_token)

    image_features = image_features / image_features.norm(dim=-1, keepdim=True)
    text_features = text_features / text_features.norm(dim=-1, keepdim=True)

    return 100 - (image_features @ text_features.T).mean() * clip_model.logit_scale.exp()
```

مطابق توضیحات بخش اول گزارش، ابتدا feature های تصویر و prompt را پیدا کرده، سپس آنها را نرمالایز کرده تا بتوانیم راحت تر فاصله کسینوسی آنها را پیدا کرده و درنهایت با محاسبه ضرب داخلی آنها (حالا که نرمالایز شده اند) فاصله کسینوسی آنها را به دست می‌آوریم.

سپس حلقه train که ماتریس نویز بهینه را پیدا می‌کند را به شکل زیر تعریف می‌کنیم:

```python
def optimize_noise(pipe, metric_model, tokenizer, prompt, noise, iterations=50, lr=0.05, compute_metric_loss=compute_clip_loss):
    noise.requires_grad_(True)
    optimizer = torch.optim.Adam([noise], lr=lr)
    preprocess = Compose(
        [
            Resize(224, interpolation=InterpolationMode.BICUBIC),
            CenterCrop(224),
            Normalize(
                (0.48145466, 0.4578275, 0.40821073),
                (0.26862954, 0.26130258, 0.27577711),
            ),
        ]
    )

    for i in tqdm(range(iterations)):
        optimizer.zero_grad()

        latents = 1 / 0.18215 * noise
        decoded_latents = pipe.vae.decode(latents).sample
        generated_images = (decoded_latents / 2 + 0.5).clamp(0, 1)
        images = torch.nn.functional.interpolate(generated_images, size=(224, 224), mode="bilinear", align_corners=False)
        images = preprocess(images)
        torch.cuda.empty_cache()

        loss = compute_metric_loss(metric_model, tokenizer, prompt, images)
        loss = torch.autograd.Variable(loss, requires_grad=True)
        torch.cuda.empty_cache()
        print(f"Iteration {i}, Loss: {loss.item()}")

        loss.backward()
        optimizer.step()

    return noise
```

در این تابع، ابتدا یک transform برای تبدیل تصویر خروجی مدل به ورودی مناسب برای CLIP (بر اساس سایز ورودی CLIP و میانگین و انحراف معیار کانال‌های آن) تعریف کرده. سپس latent که همان نویز با ضریب

مقیاس کننده است را به decoder مدل می‌دهیم و آن را بین 0 تا 1 scale می‌کنیم. می‌دانیم که بخش decoder در این مدل‌ها وظیفه رسم feature ها روی نویز را دارد، پس خروجی این بخش، گزینه مناسبی برای ورودی loss function است. اگر نویز اولیه به گونه ای باشد که decoder بتواند به خوبی feature ها را روی آن بسازد، ما به هدف ReNO رسیده‌ایم.

در نهایت خروجی decoder به loss function داده شده و طبق مشتق آن نویز اولیه آپدیت می‌شود. (توجه شود که تمام پارامترها به جز نویز، freeze شده اند و تغییری نمی‌کنند)

حال که توابع کمکی را پیاده سازی کردیم، نوبت تعریف مدل‌ها است:

```
pipe = StableDiffusionPipeline.from_pretrained(
    "stabilityai/sd-turbo",
    variant="fp16",
)
pipe = pipe.to("cuda")
```

```
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it as secret in your Google Cola
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
Loading pipeline components...: 100% 5/5 [00:13<00:00,  3.81s/it]
You have disabled the safety checker for <class 'diffusers.pipelines.stable_diffusion.pipeline_stable_diffusion.StableDiffusionPipeline'> by passing `safety
```

```
def freeze_params(params):
    for param in params:
        param.requires_grad = False

freeze_params(pipe.vae.parameters())
freeze_params(pipe.unet.parameters())
freeze_params(pipe.text_encoder.parameters())
```

```
clip_model = CLIPModel.from_pretrained("openai/clip-vit-base-patch32").to("cuda")
tokenizer = CLIPProcessor.from_pretrained("openai/clip-vit-base-patch32")
freeze_params(clip_model.parameters())
```

برای مدل pre-trained ساخت تصویر، مطابق خواسته از stable diffusion turbo استفاده شده است و تمام پارامترهایش freeze شده اند. برای متریک CLIP از "openai/clip-vit-base-patch32" استفاده شده است. البته استفاده از "laion/CLIP-ViT-H-14-laion2B-s32B-b79K" که در پیاده سازی نویسندگان مقاله ReNO هم استفاده شده بود گزینه بهتری است، اما به دلیل اینکه حافظه زیادی از رم cuda می‌گیرد و سایت‌هایی مثل colab و Kaggle رم کافی نداشتند (و من هم به سیستم یا سرور قدرتمندی دسترسی نداشتم) از مدل قبلی که تعداد پارامترهای کمتری دارد استفاده کردم.

درنهایت پارامترهای ورودی مساله را مقداردهی کرده و خروجی مدل در قبل و بعد از بهینه‌سازی ReNO را ذخیره می‌کنیم:

برای متریک BLIP هم loss function را مطابق کد (که خیلی شبیه به loss function متریک CLIP عمل کرده) تعریف می‌کنیم.

بقیه توابع و کارهای مورد نیاز برای متریک BLIP مشابه CLIP است و آن را دوباره توضیح نمی‌دهم. اما به دلیل اینکه با BLIP میزان مصرف رم cuda بیشتر از رم google colab بود، نتوانستم از آن خروجی بگیرم، صرفا کد آن را در فایل jupyter notebook قرار دادم تا بتوان با سیستم مناسب آن را اجرا کرد.

بررسی خروجی کد

نکته: صرفا با تغییر prompt که در بالا وجود داشت، می‌توان خروجی کد با prompt های مختلف بررسی کرد. اما من چون محدودیت استفاده از GPU در google colab را روی هر دو اکانت ایمیلم تمام کرده بودم، فقط خروجی یکی از آنها که هنگام تست کد به دست آوردم را دارم. آن در زیر تحلیل شده است:

Prompt برای آن خروجی "a gold bench and a green clock" است. همانطور که می‌توان دید، این prompt شامل دو جسم مختلف با دو صفت مختلف است (هر دو صفت رنگ هستند، اما چون gold هم به معنای رنگ طلایی و هم به معنای جنس طلا است و در prompt توضیح اضافه تری که راهنمایی کند کدام معنی مد نظر است نیامده، مدل می‌تواند هرکدام را خروجی دهد و خروجی اش درست باشد)

بدون استفاده از تکنیک ReNO خروجی کد به صورت زیر است:

همانطور که می‌توان دید، مدل توانسته است یکی از اجسام را به خوبی به تصویر بکشد (ساعت سبز)، اما جسم دیگر (نیمکت طلایی) را درست نشان نداده و فقط یک نیمکت/مبل سبز رنگ را نشان داده است. اگر با ReNO مطابق کد بالا در 50 تا iteration و با learning rate مساوی با 0.01 را با همان نویز به عنوان نویز اولیه اجرا کنیم، به خروجی زیر می‌رسیم:

همانطور که می‌توان دید، ساعت سبز همچنان در همان مکان قبلی وجود دارد (با شکلی کمی متفاوت به خاطر تغییرات جزئی نویز در آن ناحیه)، ولی نیمکت طلایی در این تصویر اضافه شده است (که نشان می‌دهد نویز در آن ناحیه تغییرات قابل توجهی داشته است).

بقیه عکس ها (که ماتریس نویز اولیه در آنها با seed اولیه 42 ساخته شده) در فولدر pics به همراه آنالیز مختصری از آن prompt وجود دارد.

جمع بندی تحلیل تصاویر

در اکثر موارد، بهینه سازی ReNO به بهتر شدن تصاویر کمک کرد. در مجموع، به نظر متریک CLIP بهتر از BLIP بود. دلیل آن چند چیز می‌تواند باشد:

1. مدل BLIP استفاده شده ساده‌تر از مدل CLIP استفاده شده بوده، و درنتیجه توانایی تحلیل تصاویر و متن را به خوبی آن مدل به خصوص از CLIP نداشته است.
2. Hyperparameters برای بهینه سازی با CLIP بهتر از BLIP تنظیم شده اند. طبق بررسی هایی که با هایپرپارامترهای مختلف داشتم، 50 تا iteration به همراه learning rate = 0.05 برای CLIP و learning rate = 0.01 برای BLIP نتیجه بهتری می‌داد. هرچه learning rate بیشتر باشد، تصویر نهایی شباهت کمتری با تصویر اولیه خواهد داشت، اما ممکن است خیلی کیفیت تصویر هم پایین بیاید.
3. در کل معماری BLIP به نسبت به CLIP ضعیف‌تر است و نمی‌تواند به خوبی آن تصویر و نوشته ها را تحلیل کند.

طبق این prompt ها که تمرکز زیادی روی رنگ اشیا داشتند، عملکرد CLIP معمولا به این صورت بود که سعی می‌کرد تمام اشیا مورد نظر در تصویر باشند، اما توجه کمتری به صفات آنها (در این مورد رنگ) دارد. اما BLIP در مقایسه با CLIP بهتر رنگ های موجود در prompt را تشخیص داده و سعی می‌کند آنها را در تصویر بگذارد، اما به خوبی CLIP نمی‌تواند هر صفت را به اشیا مناسب ارتباط دهد.