E4: Multimodal Diffusion Models

Group: 11
Colab link: https://colab.research.google.com/drive/1xtJNUBeDspImojGWmT-dlH4qZ9CTZ0tI0?usp=sharing

11: Viktor Maximilian Lehnhausen, Mikael Alves Brito, Georg Matthes

TECHNISCHE UNIVERSITÄT DARMSTADT

Question Part (10 Pts.)

1. ✍ **Energy-based Models** (4 Pts.)

   a) Summary (2 Pts.):

   Energy-Based Models (EBMs) define a probability distribution using an energy function, which assigns low energy to likely things and high energy to unlikely ones. EBMs don't require explicit normalization of the distribution, which makes them, in comparison to other models, more flexible. A key idea of EBMs is, instead of computing the probability itself, they learn the gradient of the log probability. This gradient is called the score function and can be used in score-based DMs to iteratively denoise the data and to generate output. The connection of EBMs and DMs is this score-based approach of modeling the data distributions.

   b) Mathematical Relationship (2 Pts.):

$$p_\theta(x) = \frac{\exp(-e_\theta(x))}{Z_\theta}$$

$$\log p_\theta(x) = -e_\theta(x) - \log Z_\theta$$

$$\nabla_x \log p_\theta(x) = -\nabla_x e_\theta(x) - \nabla_x \log Z_\theta$$

Since $\nabla_x \log Z_\theta = 0$, as Z_θ does not depend on x,

$$\nabla_x \log p_\theta(x) = -\nabla_x e_\theta(x) = s_\theta(x)$$

2. ✍ **Inference Efficiency** (2 Pts.)

CFG increases the computational cost during inference by requiring two separate model evaluations per timestep: one with the prompt (conditional) and one without (unconditional). For a batch of B prompts over T timesteps, this leads to a total of $2 \cdot B \cdot T$ forward passes. This doubles the workload compared to standard conditional diffusion. However, this overhead can be reduced if the unconditional input is identical across the batch. In such cases, diffusion models can pre-compute and cache the unconditional scores once per timestep. If all B prompts share the same noise schedule and use the same unconditional context, the unconditional pass only needs to be computed once per timestep and can be reused, resulting in $B \cdot T + T$ forward passes.

3. ✍️ **Learn about Textual Inversion** (4 Pts.)

Textual Inversion goal is to improve fine-tuning of image generation models. The at that time state of the art models suffered from various limitations during and after fine-tuning, (i.e. forgetting). The authors present a way to encoperate new pictures in the model with low overhead.
With Textual Inversion already trained text-to-image diffusion models can learn new concepts/objects that were not seen during training, without needing to learn the whole model again. Instead only the new word is learned.
The model is given a new word (e.g. Pokeball) and several example images of that object. A new vector is added to the text embedding, and only this vector is optimized during training. All other vectors are frozen.
In each training step, a noisy image is used and the model tries to remove the noise and reconstruct an image similiar to the examples. The new vector is updated so that the output images become closer to the input images.
Limitations: It could be difficult to learn complex objects, because only one vector is used to represent the whole object.
If embeddings are frozen during fine-tuning, the model can only adapt poorly, because it keeps the pre-trained parameters at their original values. This is helpful if the new data is similar but can limit performance when the new data is very different.
Futhermore Textual Inversion does not update the pre-trained model. This means that when adding many new concepts, the model might struggle to represent them well, especiallly if they conflict with existing knowledge. In that case training a new model might be more effective than adding objects.

Coding Part (15 Pts.)

1. 💻 **Loss Function** (2 Pts.)

```
# --------------- YOUR CODE STARTS HERE ---------------------------------
    noise = torch.randn_like(latents)

    timestep = wrapper.scheduler.timesteps[timestep_idx].unsqueeze(0)
    timestep = timestep.to(wrapper.pipe._execution_device)

    noisy_img = wrapper.scheduler.add_noise(latents, noise, timestep)
```

```
    noise_predicted = wrapper.predict_noise(timestep_idx, noisy_img, embeddings)

    loss = torch.nn.functional.mse_loss(noise_predicted, noise, reduction="mean")

# ---------------- YOUR CODE ENDS HERE --------------------------------
```

2. 💻 **Optimization Setup** (5 Pts.)

```
tokenizer = wrapper.tokenizer
text_encoder = wrapper.text_encoder
placeholder_tokens = config.placeholder_tokens
initializer_token = config.initializer_token
prompt_templates = config.prompt_templates
lr = config.learning_rate

# (1) store copy of original embeddings
wrapper._original_embeddings = text_encoder.get_input_embeddings().weight.data.clone()

num_added_tokens = tokenizer.add_tokens(placeholder_tokens)
if num_added_tokens != len(placeholder_tokens):
    raise ValueError(
        f"The␣tokenizer␣already␣contains␣some␣of␣the␣placeholder␣tokens.\n"
        f"Please␣choose␣different␣tokens."
    )


text_encoder.resize_token_embeddings(len(tokenizer))
embeddings = text_encoder.get_input_embeddings()

placeholder_token_ids = tokenizer.convert_tokens_to_ids(placeholder_tokens)

with torch.no_grad():
    initalizer_token_id = tokenizer.convert_tokens_to_ids(initializer_token)
    initalizer_embeddings = embeddings.weight[initalizer_token_id].clone()

    for token_id in placeholder_token_ids:
        embeddings.weight[token_id] = initalizer_embeddings.clone()

# Freeze vae, unet and text encoder except embeddings
for param  in wrapper.vae.parameters():
    param.requires_grad = False

for param  in wrapper.unet.parameters():
    param.requires_grad = False

for param in text_encoder.parameters():
    param.requires_grad = False

embeddings = text_encoder.get_input_embeddings()
embeddings.weight.requires_grad = True

wrapper.placeholder_token_ids = placeholder_token_ids

optimizer = torch.optim.AdamW([embeddings.weight], lr=lr)
```

```
og_step = optimizer.step

def patched_step(closure=None):
    og_step(closure)

    # set not placeholder tokens to _original_embeddings
    with torch.no_grad():
        current_step_embeddings = wrapper.text_encoder.get_input_embeddings()
        for i in range(len(current_step_embeddings.weight)):
            if i not in wrapper.placeholder_token_ids:

                if i < len(wrapper._original_embeddings): #unsure ?
                    current_step_embeddings.weight[i] = wrapper._original_embeddings[i]

optimizer.step = patched_step
```

3. 💻 **Main Algorithm** (8 Pts.)

   a) 💻 **Implementation**

```
optimizer = prepare_for_textual_inversion(wrapper, config)
wrapper.vae.eval()
wrapper.unet.eval()
wrapper.text_encoder.train()

dataset = UnlabeledImageFolderDataset(config.target_folder,
                                      config.max_num_target_images,
                                      train_transform
                                      )
dataloader = torch.utils.data.DataLoader(dataset, batch_size=1, shuffle=True)

prompt_templates = []
for template in config.prompt_templates:
    for placeholder in config.placeholder_tokens:
        prompt_templates.append(template.format(placeholder))

for step in range(config.num_inversion_steps):
    try:
        image = next(iter(dataloader))
    except:
        dataloader = torch.utils.data.DataLoader(dataset,
                                                 batch_size=1,
                                                 shuffle=True
                                                 )
        image = next(iter(dataloader))

    with torch.no_grad():
        image = image.to(wrapper.pipe._execution_device)
        latents = wrapper.encode_image_to_latents(image)

    prompt_idx = torch.randint(0, len(prompt_templates), (1,)).item()
    prompt = prompt_templates[prompt_idx]

    c_p_e = wrapper.encode_prompt(prompt)

    timestep_idx = torch.randint(0, len(wrapper.scheduler.timesteps), (1,)).item()
```

```
    optimizer.zero_grad()
    loss = compute_standard_diffusion_loss(wrapper, latents, timestep_idx, c_p_e)
    loss.backward()
    optimizer.step()

    if step % config.show_every == 0:
        print(f"Step: {step}/{config.num_inversion_steps}, Loss: {loss.item():.4f}")

        validation_sampling(wrapper, config, step)

print("Textual inversion fine tuning done!")
```

b) 💻 **Evaluation**

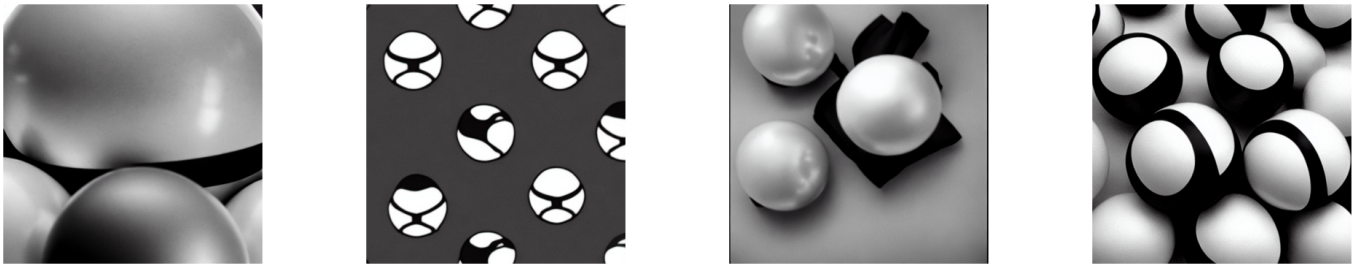
```
#3 b) i.
for i in range(4):
    images = wrapper(["black and white photo of placeholder_1"],
    generator=torch.Generator().manual_seed(42+i))
    show_images(images)
    images = wrapper(["oil paining of placeholder_1"], generator=torch.Generator
        ().manual_seed(42+i))
    show_images(images)
    images = wrapper(["a kid drawing placeholder_1"], generator=torch.Generator
        ().manual_seed(42+i))
    show_images(images)
    images = wrapper(["a photo of placeholder_1 at the top of a mountain"],
        generator=torch.Generator().manual_seed(42+i))
    show_images(images)
own_config = TextualInversionConfig(target_folder="/kaggle/input/keith-haring/
    keith_haring", placeholder_tokens=["placeholder_2"], initializer_token="
    style")
textual_inversion(wrapper, own_config)
images = wrapper(["photo of a cat in style of placeholder_2"], generator=torch.
    Generator().manual_seed(42))
show_images(images)
# 3 b) ii.
for i in range(4):
    images = wrapper(["black and white photo of placeholder_2"], generator=torch
        .Generator().manual_seed(42+i))
    show_images(images)
for i in range(4):
    images = wrapper(["oil paining of placeholder_2"], generator=torch.Generator
        ().manual_seed(42+i))
    show_images(images)
for i in range(4):
    images = wrapper(["a kid drawing placeholder_2"], generator=torch.Generator
        ().manual_seed(42+i))
    show_images(images)
for i in range(4):
    images = wrapper(["a photo of placeholder_2 at the top of a mountain"],
        generator=torch.Generator().manual_seed(42+i))
    show_images(images)
# 3 b) iii.
print("a photo of placeholder_1 in the style of placeholder_2")
for i in range(0,4):
    images = wrapper(["a photo of placeholder_1 in the style of placeholder_2"],
```

```
        generator=torch.Generator().manual_seed(42+i))
    show_images(images)

print("placeholder_1 and placeholder_2 at the beach")
for i in range(0,4):
    images = wrapper(["placeholder_1 and placeholder_2 at the beach"], generator
        =torch.Generator().manual_seed(42+i))
    show_images(images)
```

c) ✍️ **Write-Up**

The pokeballs can easily be represented with a prompt for a black and white image. This can be seen in fig. 1. Obviously it is a bit harder to spot the pokeball, as it lives from its red and white parts, but the brigther and darker halfs of a round ball can be reasembled to a pokeball. The other placeholder which were pictures of a artwork is also easily reproduced in the black and white prompt fig. 4. A more challanging prompt is a kid drawing the placeholders. The four images in fig. 2 at least contain some parts of chilrdren. Image 1 and 2 some realistic pictures, where in 3 there seems to be a confusion, as a kids head is drawn on a pokeball. The fourth Image from this prompt looks like a drawn kid with a pokeball in its hands. Our own visual concept the first three images fail to include some children fig. 5. There even is a fail, where the model produced potential NSFW output and it is rightfully censored. Only in the fourth image we can identify some childrens sitting at a desk. The prompt where we put the placeholder on top of a mountain fail completly for our own concept fig. 6. The pokeball differs for this prompt fig. 3. The foruth image at least contains some hilly lawn, but on the other tries the model missunderstood something. placeholder_2 is very dominant and other concepts of the prompt struggle to seek through. placeholder_1 is not that dominant and can connect better with the rest of the prompt. In a abstract way the model can connect the learned concepts. It is picking up the style of like intended (placeholder_2) and uses the right color concept of placeholder_2. It lacks of the chaotic steal of placeholder_2 and the concept of placeholder_1 is in some representations not more established than in the color scheme. But in none of the pictures an pokeball is recognizable, that could mean that the model has problems to connect those to objects but it could also be because placeholder_2 is very abstract.



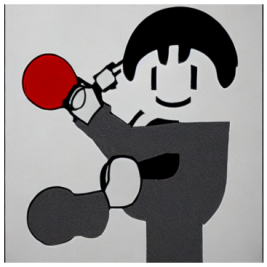
(a) BW Image 1 of p1 (b) BW Image 2 of p1 (c) BW Image 3 of p1 (d) BW Image 4 of p1

Figure 1: Four black and white (BW) images using placeholder_1 (p1) representing the pokeballs. The prompt used was: ”black and white photo of placeholder_1”

(a) Image 1

(b) Image 2

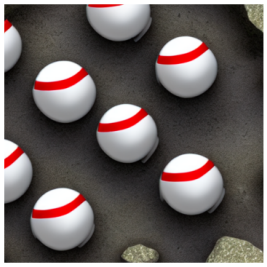
(c) Image 3



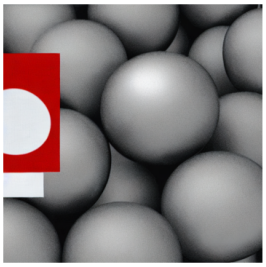
(d) Image 4

Figure 2: Four images using placeholder_1 representing the pokeballs using the prompt: ”a kid drawing placeholder_1”

(a) Image 1



(b) Image 2



(c) Image 3

(d) Image 4

Figure 3: Four image of placeholder_1 at the top of a mountain. The prompt used was: ”a photo of placeholder_1 at the top of a mountain”

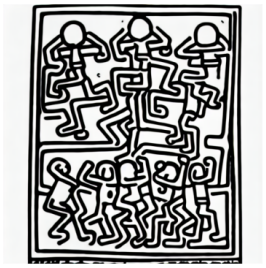
(a) Image 1

(b) Image 2

(c) Image 3

(d) Image 4

Figure 4: Four black and white images of placeholder_2. The prompt used was: ”black and white photo of placeholder_2”.



(a) Image 1

(b) Image 2

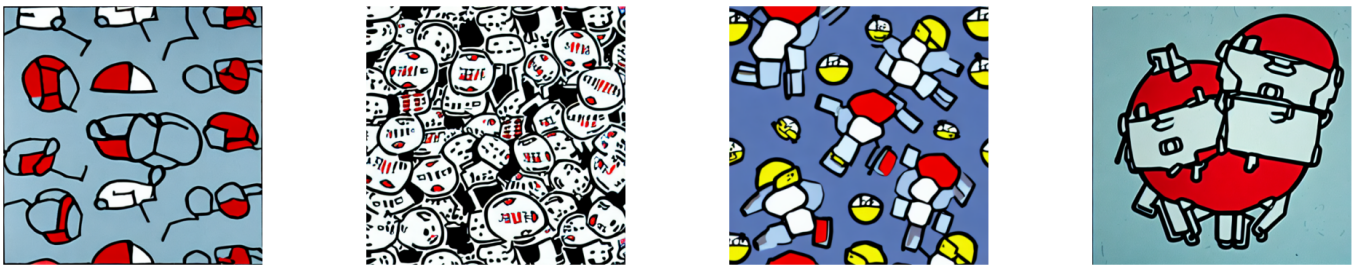
(c) Image 3

(d) Image 4

Figure 5: Four images of placeholder_2 drawn by a kid. The prompt used was: ”a kid drawing placeholder_2”.

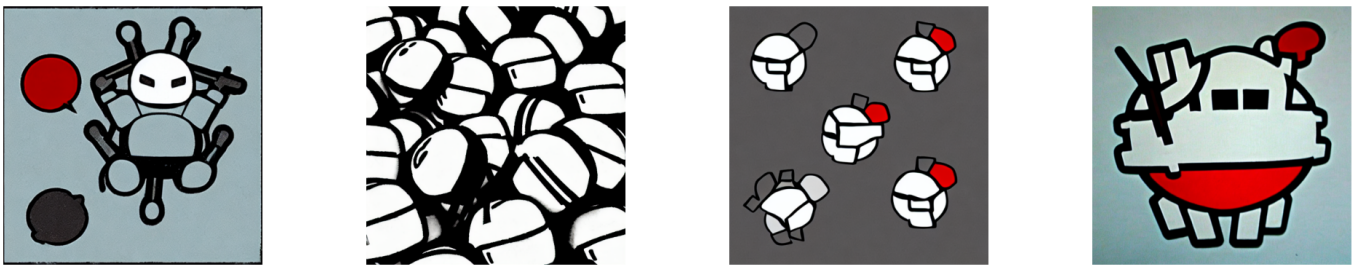
(a) Image 1 (b) Image 2 (c) Image 3 (d) Image 4

Figure 6: Four images of placeholder_2 on top of a mountain. The prompt used was: ”a photo of placeholder_2 at the top of a mountain”.



(a) Image 1 (b) Image 2 (c) Image 3 (d) Image 4

Figure 7: Four images using placeholder_1 and placeholder_2. The prompt used was: ”placeholder_1 and placeholder_2 at the beach”.



(a) Image 1 (b) Image 2 (c) Image 3 (d) Image 4

Figure 8: Four images using placeholder_1 and placeholder_2. The prompt used was: ”a photo of placeholder_1 in the style of placeholder_2”.