VG

Optimize:

```
import argparse
import os
import glob
from PIL import Image
from tqdm import tqdm
import numpy as np
import torch
from main import instantiate_from_config
from ldm.models.diffusion.ddim import DDIMSampler
from ldm.invoke.devices import choose_torch_device
def make_batch(image, mask, device):
    image = np.array(Image.open(image).convert("RGB"))
    image = image.astype(np.float32) / 255.0
    image = image[None].transpose(0, 3, 1, 2)
    image = torch.from_numpy(image)
    mask = np.array(Image.open(mask).convert("L"))
    mask = mask.astype(np.float32) / 255.0
    mask = mask[None, None]
    mask[mask < 0.5] = 0
    mask[mask >= 0.5] = 1
    mask = torch.from_numpy(mask)
    masked_image = (1 - mask) * image
    batch = {"image": image, "mask": mask, "masked_image": masked_image}
    for k in batch:
        batch[k] = batch[k].to(device=device)
        batch[k] = batch[k] * 2.0 - 1.0
    return batch
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "--indir",
        type=str,
        nargs="?",
        help="dir containing image-mask pairs (`example.png` and `example_mask.png`)",
    )
    parser.add_argument(
        "--outdir",
        type=str,
        nargs="?",
        help="dir to write results to",
    )
    parser.add_argument(
        "--steps",
        type=int,
        default=50,
        help="number of ddim sampling steps",
    )
    opt = parser.parse_args()
    masks = sorted(glob.glob(os.path.join(opt.indir, "*_mask.png")))
    images = [x.replace("_mask.png", ".png") for x in masks]
    print(f"Found {len(masks)} inputs.")
    config = OmegaConf.load("models/ldm/inpainting_big/config.yaml")
    model = instantiate_from_config(config.model)
    model.load_state_dict(torch.load("models/ldm/inpainting_big/last.ckpt")["state_dict"],
    )
    device = choose_torch_device()
    model = model.to(device)
    sampler = DDIMSampler(model)
    os.makedirs(opt.outdir, exist_ok=True)
    with torch.no_grad():
        with model.ema_scope():
            for image, mask in tqdm(zip(images, masks)):
                outpath = os.path.join(opt.outdir, os.path.split(image)[1])
                batch = make_batch(image, mask, device=device)
                c = model.cond_stage_model.encode(batch["masked_image"])
                cc = torch.nn.functional.interpolate(batch["mask"],
                                                     size=c.shape[-2:])
                c = torch.cat((c, cc), dim=1)
                shape = (c.shape[1]-1,)+c.shape[2:]
                samples_ddim, _ = sampler.sample(S=opt.steps,
                                                 conditioning=c,
                                                 batch_size=c.shape[0],
                                                 shape=shape,
                                                 verbose=False)
                x_samples_ddim = model.decode_first_stage(samples_ddim)
                image = torch.clamp((batch["image"]+1.0)/2.0,
                                    min=0.0, max=1.0)
                mask = torch.clamp((batch["mask"]+1.0)/2.0,
                                   min=0.0, max=1.0)
                predicted_image = torch.clamp((x_samples_ddim+1.0)/2.0,
                                              min=0.0, max=1.0)
                inpainted = (1-mask)*image+mask*predicted_image
                inpainted = inpainted.cpu().numpy().transpose(0,2,3,1)[0]*255
                Image.fromarray(inpainted.astype(np.uint8)).save(outpath)
```

I see several ways you could optimize the code you've provided:

1. Use the `functools.partial` method to create a new function that has some of the parameters of the `make_batch` function fixed. This will allow you to avoid passing in the same values for those parameters each time you call `make_batch`, and can make your code more readable. For example, you could write:

```
from functools import partial

def make_batch(image, mask, device):
    ...

make_batch_with_device = partial(make_batch, device=device)
```

Then, you can use `make_batch_with_device` just like you would use `make_batch`, but you don't have to pass in the `device` parameter each time.

2. Use the `map` function to apply the `make_batch_with_device` function to each of the `images` and `masks` in parallel, rather than using a `for` loop. This can speed up your code if you have a large number of images and masks. For example:

```
batches = map(make_batch_with_device, images, masks)
```

3. Use the `torch.no_grad` context manager to avoid storing gradient information when you call the `sample` method of the `DDIMSampler` class. This can reduce the memory usage of your code and make it run faster. For example:

```
with torch.no_grad():
    samples_ddim, _ = sampler.sample(...)
```

4. Use the `torch.Tensor.to` method to move the data for each batch to the device that you're using for inference, rather than using the `make_batch` function to do this. This can reduce the amount of data that needs to be transferred between the CPU and the device, and can make your code run faster. For example:

```
image = torch.from_numpy(image)
image = image.to(device=device)
mask = torch.from_numpy(mask)
mask = mask.to(device=device)
```

5. Use the `torch.Tensor.permute` method to transpose the dimensions of the `image` tensor, rather than using the `numpy.transpose` method and converting the tensor to a numpy array. This can avoid the overhead of converting between numpy arrays and PyTorch tensors, and can make your code run faster. For example:

```
image = torch.from_numpy(image)
image = image.permute(0, 3, 1, 2)
```

6. Use the `torch.nn.functional.interpolate` method to resize the `mask` tensor, rather than using the `PIL.Image.resize` method and converting the tensor to a numpy array. This can avoid the overhead of converting between numpy arrays and PyTorch tensors, and can make your code run faster. For example:

```
mask = torch.from_numpy(mask)
mask = torch.nn.functional.interpolate(mask, size=c.shape[-2:])
```

7. Use the `torch.cat` method to concatenate the `c` and `cc` tensors along the first dimension, rather than using the `