In [ ]:
```
# # installations
# !pip install --upgrade pip

# !python --version
# !pip3 install jedi==0.16

# !pip3 uninstall cvxpy -y > /dev/null
# !pip3 install setuptools==65.5.0 pip==22.2 > /dev/null
# # hack for gym==0.21.0 https://github.com/openai/gym/issues/3176
# !pip3 install torch==1.13.1 torchvision==0.14.1 diffusers==0.11.1 \
# scikit-image==0.19.3 scikit-video==1.1.11 zarr==2.12.0 numcodecs==0.10.
# pygame==2.1.2 pymunk==6.2.1 gym==0.21.0 shapely==1.8.4 dnn \
# &> /dev/null # mute output
# !pip install --upgrade "jax[cuda12_pip]"==0.4.23 -f https://storage.goo
# # This will take a while, and do not worry if you get some warnings

## Issues: numcodecs==0.10.2, pygame==2.1.2, gym==0.21.0, upgrade jax num
```

In [ ]:
```
# from google.colab import drive
# drive.mount('/content/drive')

%load_ext autoreload
%autoreload 2

import os
#change to desired path to the part2-DiffusionModels folder
# os.chdir("./drive/MyDrive/hw4/DiffusionModels")
os.chdir("./part2_DiffusionModels")
%pwd
```

Out[ ]:
```
'/home/anascimento7/CS7643/Assignments/hw4_code_student_version/part2_Di
ffusionModels'
```

In [ ]:
```
from part2_DiffusionModels.tests import TestCases
unit_tester = TestCases()
```

Homework 3: Part 2 - Diffusion Models [21 points]

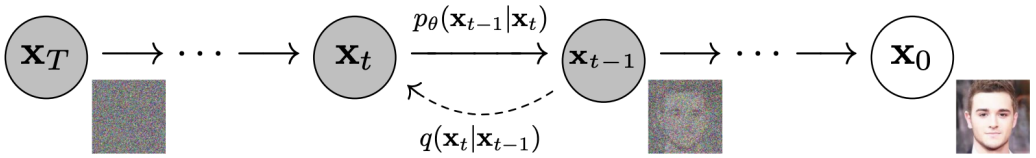
Denoising Diffusion Probabilistic Models (DDPM) - https://arxiv.org/abs/2006.11239/
Classifier Free Guidance (CFG) - https://arxiv.org/abs/2207.12598

In this part of the assignment you will be implementing a diffusion model and applying it in an image generation task. Denoising Diffusion Probabilistic Models (DDPMs), also known as Diffusion Models, are state of the art generative models that train a network to itteratively denoise random gaussian noise. If you have not done so already, please read through the DDPM paper linked above in order to get an understanding of the math behind DDPMs. The code for this part of the assignment is not particularly difficult, but you will have a difficult time if you do not understand the theory behind diffusion models. You will also be implementing classifier free guidance (CFG). While the CFG paper does not have to be read as thoroughly as the DDPM paper, it is definitely a useful read if you are not familiar with CFG.

Note: (you "may" have to revisit this part several times)

- Algorithm1 and Algorithm2 in the paper use 1 indexing for timesteps. We are using 0 indexing.
- In the paper they talk about using either β or $\tilde{\beta}$ for variance. We will always use β, even for the thresholding section
- For thresholding, when t = 0, just set $\bar{\alpha}_{t-1} = 1$
- Use noise_pred_net(sample = x, timestep = t, global_cond = c) to generate an output from the noise prediction network (alternatively you can just call noise_pred_net(x, t, c))
- For CFG, please use zeros_like

Creating the DDPM



For the sake of this assignment, we are breaking the Diffusion Model into two parts. A noise scheduler and a noise prediction net. The noise scheduler handles the noise addition during the forward process and the noise removal during the reverse process. The noise prediction net is a neural network that predicts the amount of noise that should be removed at each denoising step. Transformers, like the one you implemented in the last assignment, can be used as this noise prediction network. For this assignment we are using a U-Net as opposed to a transformer, but you will not be implementing this model, just training it.

IMPORTANT NOTE FOR IMPLEMENTATION!

You will be generating random numbers in various sections of this project. To keep the outputs deterministic for unit test and auto grader purposes, you must use our prebuilt randomizer function. This randomizer is defined in `DiffusionModels/randomizer.py`. Please take a look at that function before you start coding. The function that we define is sufficient for generating any random numbers that you will need for this project.

IMPORTANT NOTE 2

The equations in the DDPM paper describe an unconditional model, you will be implementing a conditional model. This does not significantly change the equations, you just need to pass in the conditioning term along with x and t. Make sure to watch the first lecture on generative models if you are running into confusion about this or other aspects of Diffusion Models.

IMPORTANT NOTE 3

Algorithm 1 - Training, and Algorithm 2 - Sampling in the DDPM paper use 1 indexing for timesteps. However, you should use 0 indexing (index start from 0).

Part 2.1 - Noise Scheduler [6 points]

2.1.1 - Forward Process (Adding Noise) [3 points] [.5 writeup points]

During the forward process, we take an input and gradually add Gaussian noise to the data according to a variance schedule. This is described in section 2, background, of the DDPM paper. You must read that section to complete this part of the assignment.

TODO: Complete initialization function in `DiffusionModels/noise_scheduler.py`

TODO: Implement `add_noise` function in `part2-DiffusionModel/noise_scheduler.py`

Hint: Pay attention to equation 4

Question: what does $q(x_t|x_o)$ represent for a diffusion model? (read the paper)

Answer: the (Gaussian) noise generation process for any t conditioned on x_0. That is the distribution used for the noising (forward) process.

In [ ]:
```
# test initialization
unit_tester.test_noise_scheduler_init()
# test add noise
unit_tester.test_add_noise()
```

```
NoiseScheduler() initialization test case passed!
add_noise() test case passed!
```

2.1.2 - Reverse Process (Removing Noise) [2.5 point]

When taking a step during the backwards process, we have access to a partially noised sample x_t, the denoising timestep t, and our model's noise prediction. We remove the predicted noise to get a slightly less noisy sample x_{t-1}. The completely uncorrupted sample is recovered at x_0. This process is described in detail in section 3 of the DDPM paper, please read this section before completing this part of the assignment.

TODO: Implement `denoise_step` function in `DiffusionModels/noise_scheduler.py` (do not implement the thresholding part yet)

Hint: Pay attention to algorithm 2 - Sampling

In [ ]:
```
# test denoise step
unit_tester.test_denoise_step()
```

```
denoise_step() test case passed!
```

Part 2.2 - Constructing the Diffusion Model [7

Points]

We have provided you with a noise prediction network, and you have just written a noise scheduler. With these two parts you are able to put together a completed diffusion model.

2.2.1 Compute Loss for Training [2 points] [.5 writeup points]

For this part of the assignment, you will be implementing the `compute_loss_on_batch` function of the diffusion model. We have implemented the rest of the training logic for you. Keep in mind, the only part of the diffusion model that needs to be trained is the noise prediction network. Your loss function should capture how well the noise_pred_net estimates the added noise. We expect you to implement the simplified variant of the variational lower bound as described in section 3.4 of the DDPM paper.

TODO: Implement loss computation section of `compute_loss_on_batch` function in `DiffusionModels/diffusion_model.py`

Hint: Use your noise scheduler, refer to equation 14 and Algorithm 1 - Training
Hint: For each item in the batch, you should be selecting a random timestep t to compute the loss

Question: What is $\epsilon_\theta(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, t)$?

Answer: It is the learned noise, which we are optimizing for (i.e., updating θ so that we can correctly predict ϵ).

In a more detailed manner:

- ϵ is the ground truth that we have access to
- $\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}$ is the process of jumping to timestep t conditioned on x_0, which yields some noisy data
- That noisy data is ran though the neural net with the timestep t, which will predict $\epsilon_\theta(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, t)$ itself

In [ ]:
```
# test loss computation
# print(os.getcwd())
unit_tester.test_compute_loss_on_batch()
```

```
compute_loss_on_batch() test case passed!
```

2.2.2 Classifier Free Guidance [1 Point]

We want our diffusion model to make use of classifier free guidance. If you are not familiar with classifier free guidance, please take a look at the CFG paper now. Classifier free guidance allows us to variably combine noise predictions from a conditional and an unconditional noise prediction network at inference time. This has the effect of enhancing class specific features and is critical for high quality image

generation. In practice, we train the unconditional noise prediction net at the same time as the conditional noise prediction net. Section 3.2 of the CFG paper describes this process.

We expect you to use 0s as the null tokens.

TODO: Implement classifier free guidance section of `compute_loss_on_batch` function in `DiffusionModels/diffusion_model.py`

Hint: This should only be a few lines, use self.p_uncond

In [ ]:
```
# test classifier free guidance training
unit_tester.test_compute_loss_with_cfg()
```

```
compute_loss_with_cfg() test case passed!
```

2.2.3 Sample Generation (with classifier free guidance) [3.5 points]

Before we can actually use our diffusion model, we need to write the code for generating a sample. To generate an output, we draw a sample x_T from random gaussian noise. We then itteratively denoise the random output for T denoising timesteps until we get our uncorrupted output x_0. You will be implementing sampling with classifier-free guidance. Do not worry about replicating algorithm 2 of the CFG paper, we only expect you to calculate the updated noise prediction from equation 6 of the CFG paper:

$$\bar{\epsilon_\theta}(x_t, t, c) = (1 + w) * \epsilon_\theta(x_t, t, c) - w * \epsilon_\theta(x_t, t, \text{empty})$$

where ϵ_θ is the predicted noise from your noise prediction network and w is the guidance weight

TODO: Implement `generate_sample` function in `DiffusionModels/diffusion_model.py`

Hint: Use your noise scheduler and refer to Algorithm 2 - Sampling of the DDPM paper
Hint: Dont forget that this is a conditional model and to include classifier free guidance

In [ ]:
```
# test generate sample
unit_tester.test_generate_sample()
```

```
generate_sample_step() test case passed!
generate_sample_step() with guidance test case passed!
```

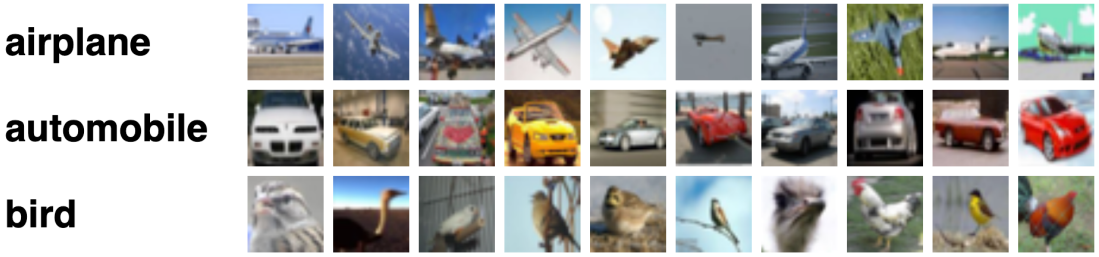
In [ ]:
```
## Trying to test threshold
unit_tester.test_threshold_denoise_step()
```

```
threshold_denoise_step() test case passed!
```

Part 2.3 - Diffusion Model Applications (Computer Vision) [8 points]

Diffusion models are mostly known in popular culture because of image generation

tools such as Dall-e and Midjourney. These models are able to take in text based prompts and output extremely realistic photos. For the sake of this assignment we will not be implementing anything that is nearly as visually impressive, but we hope to shed some insight on a few critical aspects to state of the art diffusion models. Namely classifier free guidance and thresholding. You will be working with a subset of the CIFAR-10 dataset. The CIFAR-10 dataset consists of 60000 32x32 color images in 10 classes, with 6000 images per class. We will be working with the first three classes of this dataset - airplanes, automobiles, and birds.



In [ ]:
```
# Imports and global function definitions
from part2_DiffusionModels.computer_vision.image_gen import ImageGenerato
from matplotlib import pyplot as plt

def plot_image(output):
    B, H, W, C = output.shape
    # batch size divided by number of classes
    num_cols = B // 3
    assert B % 3 == 0
    fig, axes = plt.subplots(3, num_cols, figsize=(12, 8))
    # Loop through the rows
    for i in range(3):
        # Loop through the columns
        for j in range(B // 3):
            image = output[i * (B // 3) + j]
            if num_cols == 1:
                axes[i].imshow(image)
                axes[i].axis('off')
            else:
                axes[i, j].imshow(image)
                axes[i, j].axis('off')

    # label the first row "airplane", the second row "automobile", and th
    axes[0, 0].set_title('airplane')
    axes[1, 0].set_title('automobile')
    axes[2, 0].set_title('bird')

    plt.show()
```

In [ ]:
```
# # Run this cell to download the dataset, only needs to be done once eve
# %cd computer_vision
# !wget https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
# !tar -xvzf cifar-10-python.tar.gz
# !rm cifar-10-python.tar.gz
# %cd ..
```

2.3.1 Initialization and Data Augmentation [2 writeup points]

In parts 2.1 and 2.2 you implemented diffusion model training and sample generation, so the majority of the work for generating images is done! The computer_vision/image_gen.py file handles the rest of the logic for image generation. Your first task is to complete the initialization function, this is only a few lines of code.

TODO: Complete initialization function in `DiffusionModels/computer_vision/image_gen.py`

In [ ]:
```
generator = ImageGenerator()
```

After initializing the image generator, take a look through the rest of the file to understand how the data is loaded and how the images are generated. Your next task is to perform data augmentation on the CIFAR dataset (for the sake of training time we do not require you to actually add augmenations and increase the size of your dataset, but go through this exersize with us anyway). This is a common technique in the computer vision world as it allows us to increase the size of our dataset and train the model to be invariant to certain perturbations. You may have done this before when training a classification model, but now we are training a generator. The types of augmentations that we want our generator to be invariant to are not necessariliy the same as for a classifier.

Question: *Why might we want a classifier to be invariant to different augmentations then a generator? Name 2 augmentations that we might perform on a classification dataset that we don't want to perform on a dataset for a generative model. Name 2 augmentations that you want to perform on a dataset for a generative model (its okay if you want also want to do the same augmentations on a classification dataset).*

Your Answer Here: We want classifiers need to be robust to minor variations (out-of-distribution data, but somewhat bounded) and perform the class classifcation correctly; hence, be invariant to those augmentations. On the other hand, we want generators to be more "creative" and give us diverse outputs.

- Two augmentations that we might perform on a classification dataset that we don't want to perform on a dataset for a generative model:
  - Addition of blurr only in the background, not the objects (class specific occlusion)
  - Addition of noise that preserve only the labels (label preserving noise injection)
- Two augmentations that you want to perform on a dataset for a generative model:
  - Addition of blurr (classifiers might want)
  - Addition of flips and rotations (classifiers might want)

OPTIONAL: Implement two augmentations in the `load_dataset` function of `DiffusionModels/computer_vision/image_gen.py` , (these should be the augmentations you described above). Increasing the dataset size will slow down your training, so we do not require you to actually implement these augmentations.

However, we have left room for you to add the code for augmentation if you have the time or compute.
OPTIONAL: Display an image from the CIFAR dataset and show it after the augmentations (3 images total: original, augmentation 1, augmentation 2)

In [ ]:
```
# Feel free to add more cells or generate the images separately and displ
```

2.3.2 Image Generation and Classifier Free Guidance [1 writeup point]

Now it is time to train the model - if this fails to run, go and check over your previous work. We are implementing a small model, but even still, this will take a while to train! Please use google colab if you do not have access to a GPU. The remainder of this assignment will be infeasable for you if you do not have a GPU. That being said, we understand that you have time constraints and we do not expect you to let this model train for hours (although your results would look much better if you did).

We ask that you budget 30 minutes for the training of this image generation model, you can run it for longer if you prefer though. Do not stress too much about the specific images generated, we know the outputs will be poor given your limited computation budget. If your loss isnt decreasing (should get below 0.1 after ~7 epochs of training and below 0.05 by the end of training) or you are generating random noise, then you may have a serious issue. But if your airplane is a smudge on a blue-ish background, that is a great output. The goal is for you to learn, we will be paying much more attention to your analysis than your images.

In [ ]:
```
generator.load_dataset(dataset_paths=[
    'computer_vision/cifar-10-batches-py/data_batch_1',
    'computer_vision/cifar-10-batches-py/data_batch_2',
    'computer_vision/cifar-10-batches-py/data_batch_3',
    'computer_vision/cifar-10-batches-py/data_batch_4',
    'computer_vision/cifar-10-batches-py/data_batch_5',
    ], batch_size = 64
     )
```

In [ ]:
```
# You can just rerun this cell to keep training the model

# Keep the number of epochs at 25 for now (if you need it to be smaller t
# You will have a chance to increase the training time and generate your
generator.train_policy(epochs=25)
generator.policy.save_weights('model_pths/image_weights.pth')

# Load the model from the checkpoint
# generator.policy.load_weights('model_pths/image_weights.pth')
# generator.load_dataset(dataset_paths=[
#     'computer_vision/cifar-10-batches-py/data_batch_1',
#     'computer_vision/cifar-10-batches-py/data_batch_2',
#     'computer_vision/cifar-10-batches-py/data_batch_3',
#     'computer_vision/cifar-10-batches-py/data_batch_4',
#     'computer_vision/cifar-10-batches-py/data_batch_5',
#     ], batch_size = 64
#      )
```

```
Batch:   0%|          | 0/235 [00:00<?, ?it/s]
```

```
Batch:   0%|          | 0/235 [00:00<?, ?it/s]
Training Progress:   0%|          | 0/25 [00:00<?, ?it/s]
```

```
---------------------------------------------------------------------------
-
RuntimeError                              Traceback (most recent call las
t)
Cell In[17], line 5
      1 # You can just rerun this cell to keep training the model
      2
      3 # Keep the number of epochs at 25 for now (if you need it to be sm
aller that is okay as well)
      4 # You will have a chance to increase the training time and generat
e your best images at the end of this section
----> 5 generator.train_policy(epochs=25)
      6 generator.policy.save_weights('model_pths/image_weights.pth')
      8 # Load the model from the checkpoint
      9 # generator.policy.load_weights('model_pths/image_weights.pth')
     10 # generator.load_dataset(dataset_paths=[
   (...)
     16 #     ], batch_size = 64
     17 #      )

File ~/CS7643/Assignments/hw4_code_student_version/part2_DiffusionModels/c
omputer_vision/image_gen.py:90, in ImageGenerator.train_policy(self, epoch
s)
     89 def train policy(self, epochs = 10):
---> 90     self.policy.train(train_epochs = epochs, data_loader=self.data
_loader)

File ~/CS7643/Assignments/hw4_code_student_version/part2_DiffusionModels/d
iffusion_model.py:94, in DiffusionModel.train(self, data_loader, train_epo
chs)
     92 data = data.to(self.device)
     93 cond = cond.to(self.device)
---> 94 loss = self.compute_loss_on_batch(data, cond)
     95 loss.backward()
     96 self.optimizer.step()

File ~/CS7643/Assignments/hw4_code_student_version/part2_DiffusionModels/d
iffusion_model.py:179, in DiffusionModel.compute_loss_on_batch(self, data,
cond, seed)
    173 noisy_data = self.noise_scheduler.add_noise(data, epsilon, t) # x_
t
    174 # print('noisy_data:', noisy_data.shape)
    175
    176 # # loss = (torch.norm(noise_GT - noisy_data))^2
    177
    178 ## From the notebook: use noise_pred_net(sample = x, timestep = t,
global_cond = c) to generate an output from the noise prediction network
(alternatively you can just call noise pred net(x, t, c))
--> 179 ep_theta = self.noise_pred_net(noisy_data.to(self.device), t.to(se
lf.device), cond.to(self.device))
    181 loss = torch.nn.functional.mse_loss(epsilon.to(self.device), ep_th
eta.to(self.device))
    184 ########################################
    185 #             END OF YOUR CODE         #
    186 ########################################

File ~/anaconda3/envs/cs7643-a3/lib/python3.11/site-packages/torch/nn/modu
les/module.py:1511, in Module._wrapped_call_impl(self, *args, **kwargs)
   1509     return self._compiled_call_impl(*args, **kwargs)  # type: igno
re[misc]
```

```
   1510 else:
-> 1511     return self._call_impl(*args, **kwargs)

File ~/anaconda3/envs/cs7643-a3/lib/python3.11/site-packages/torch/nn/modu
les/module.py:1520, in Module._call_impl(self, *args, **kwargs)
   1515 # If we don't have any hooks, we want to skip the rest of the logi
c in
   1516 # this function, and just call forward.
   1517 if not (self._backward_hooks or self._backward_pre_hooks or self._
forward_hooks or self._forward_pre_hooks
   1518         or _global_backward_pre_hooks or _global_backward_hooks
   1519         or  global forward hooks or  global_forward_pre_hooks):
-> 1520     return forward_call(*args, **kwargs)
   1522 try:
   1523     result = None

File ~/CS7643/Assignments/hw4_code_student_version/part2_DiffusionModels/n
oise_prediction_net.py:623, in ConditionalUnet2D.forward(self, x, t, y)
    621     y_emb = self.label_emb(y)
    622     t = t + y emb
--> 623 return self.unet_forwad(x, t)

File ~/CS7643/Assignments/hw4_code_student_version/part2_DiffusionModels/n
oise_prediction_net.py:581, in UNet.unet_forwad(self, x, t)
    580 def unet_forwad(self, x, t):
--> 581     x1 = self.inc(x)
    582     x2 = self.down1(x1, t)
    583     x2 = self.sa1(x2)

File ~/anaconda3/envs/cs7643-a3/lib/python3.11/site-packages/torch/nn/modu
les/module.py:1511, in Module._wrapped_call_impl(self, *args, **kwargs)
   1509     return self._compiled_call_impl(*args, **kwargs)  # type: igno
re[misc]
   1510 else:
-> 1511     return self._call_impl(*args, **kwargs)

File ~/anaconda3/envs/cs7643-a3/lib/python3.11/site-packages/torch/nn/modu
les/module.py:1520, in Module._call_impl(self, *args, **kwargs)
   1515 # If we don't have any hooks, we want to skip the rest of the logi
c in
   1516 # this function, and just call forward.
   1517 if not (self._backward_hooks or self._backward_pre_hooks or self._
forward_hooks or self._forward_pre_hooks
   1518         or _global_backward_pre_hooks or _global_backward_hooks
   1519         or  global forward hooks or  global_forward_pre_hooks):
-> 1520     return forward_call(*args, **kwargs)
   1522 try:
   1523     result = None

File ~/CS7643/Assignments/hw4_code_student_version/part2_DiffusionModels/n
oise_prediction_net.py:486, in DoubleConv.forward(self, x)
    484     return F.gelu(x + self.double_conv(x))
    485 else:
--> 486     return self.double_conv(x)

File ~/anaconda3/envs/cs7643-a3/lib/python3.11/site-packages/torch/nn/modu
les/module.py:1511, in Module._wrapped_call_impl(self, *args, **kwargs)
   1509     return self._compiled_call_impl(*args, **kwargs)  # type: igno
re[misc]
   1510 else:
```

```
-> 1511     return self._call_impl(*args, **kwargs)

File ~/anaconda3/envs/cs7643-a3/lib/python3.11/site-packages/torch/nn/modu
les/module.py:1520, in Module._call_impl(self, *args, **kwargs)
   1515 # If we don't have any hooks, we want to skip the rest of the logi
c in
   1516 # this function, and just call forward.
   1517 if not (self._backward_hooks or self._backward_pre_hooks or self._
forward_hooks or self._forward_pre_hooks
   1518         or _global_backward_pre_hooks or _global_backward_hooks
   1519         or  global forward hooks or  global_forward_pre_hooks):
-> 1520     return forward_call(*args, **kwargs)
   1522 try:
   1523     result = None

File ~/anaconda3/envs/cs7643-a3/lib/python3.11/site-packages/torch/nn/modu
les/container.py:217, in Sequential.forward(self, input)
    215 def forward(self, input):
    216     for module in self:
--> 217         input = module(input)
    218     return input

File ~/anaconda3/envs/cs7643-a3/lib/python3.11/site-packages/torch/nn/modu
les/module.py:1511, in Module._wrapped_call_impl(self, *args, **kwargs)
   1509     return self._compiled_call_impl(*args, **kwargs)  # type: igno
re[misc]
   1510 else:
-> 1511     return self._call_impl(*args, **kwargs)

File ~/anaconda3/envs/cs7643-a3/lib/python3.11/site-packages/torch/nn/modu
les/module.py:1520, in Module._call_impl(self, *args, **kwargs)
   1515 # If we don't have any hooks, we want to skip the rest of the logi
c in
   1516 # this function, and just call forward.
   1517 if not (self._backward_hooks or self._backward_pre_hooks or self._
forward_hooks or self._forward_pre_hooks
   1518         or _global_backward_pre_hooks or _global_backward_hooks
   1519         or  global forward hooks or  global_forward_pre_hooks):
-> 1520     return forward_call(*args, **kwargs)
   1522 try:
   1523     result = None

File ~/anaconda3/envs/cs7643-a3/lib/python3.11/site-packages/torch/nn/modu
les/conv.py:460, in Conv2d.forward(self, input)
    459 def forward(self, input: Tensor) -> Tensor:
--> 460     return self._conv_forward(input, self.weight, self.bias)

File ~/anaconda3/envs/cs7643-a3/lib/python3.11/site-packages/torch/nn/modu
les/conv.py:456, in Conv2d._conv_forward(self, input, weight, bias)
    452 if self.padding_mode != 'zeros':
    453     return F.conv2d(F.pad(input, self._reversed_padding_repeated_t
wice, mode=self.padding_mode),
    454                     weight, bias, self.stride,
    455                     pair(0), self.dilation, self.groups)
--> 456 return F.conv2d(input, weight, bias, self.stride,
    457                 self.padding, self.dilation, self.groups)

RuntimeError: cuDNN error: CUDNN_STATUS_INTERNAL_ERROR
```

First, you will evaluate the output of unguided image generation. The generate_images

will produce num_samples generations of each class.

In [ ]:
```
image = generator.generate_images(guidance=0, num_samples=8)
plot_image(image)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[16], line 1
----> 1 image = generator.generate_images(guidance=0, num_samples=8)
      2 plot_image(image)

File ~/CS7643/Assignments/hw4_code_student_version/part2_DiffusionModels/computer_vision/image_gen.py:119, in ImageGenerator.generate_images(self, guidance, num_samples, threshold, class_label)
    113     all_labels = [class_label] * num_samples * 3
    115 # The size of this conditioning tensor is different than what you would expect given the function signature of generate_sample
    116 # If you are looking at this before implementing generate_sample, the first dimension in this cond tensor is still the batch size
    117 # The generate_sample function signature reflects the shape of the conditioning tensor for the 1D prediction network (part 2.4)
    118 # If none of this makes sense to you, that is fine, you do not need to worry about any of this
--> 119 cond = torch.tensor(all_labels).to(self.policy.device)
    121 # generate the image
    122 image = self.policy.generate_sample(cond, guidance_weight = guidance, threshold=threshold)

RuntimeError: CUDA error: device-side assert triggered
CUDA kernel errors might be asynchronously reported at some other API call, so the stacktrace below might be incorrect.
For debugging consider passing CUDA_LAUNCH_BLOCKING=1.
Compile with `TORCH_USE_CUDA_DSA` to enable device-side assertions.
```

Question: What do you notice about the quality of unguided image generation? How effectively is the model learning each class? This is very early in the training process, but do you notice any features that the model is learning for each class?

Your Answer Here: Unfortunatelly, I am running into CUDA errors, the most frequent being `RuntimeError: cuDNN error: CUDNN_STATUS_INTERNAL_ERROR`, therefore I can not see the images nor draw conclusions about it. Below are some of the things I have tried, but it is also important to note that I did try many different numbers for `self.condition_dim`, both in the int format and tuple of ints.

I am running locally, in VSCode and already tried restarting the kernel, closing and reopening the program, and even restarting my machine. Unfortunatelly I don't have time to dig any further in the web or talk about this in OH. I made a post on piazza about this, apparently some people had the same issue.

Now, we will evaluate image generation with different guidance weights. To do this, we dont actually need to retrain the model at all, this guidance can be done purely at inference time. Great! Explain why we dont have to retrain for image guidance.

Your Answer Here: Unfortunatelly, I am running into CUDA errors, the most frequent being `RuntimeError: cuDNN error: CUDNN_STATUS_INTERNAL_ERROR`, therefore I can not see the images nor draw conclusions about it. Below are some of the things I have tried, but it is also important to note that I did try many different numbers for `self.condition_dim`, both in the int format and tuple of ints.

I am running locally, in VSCode and already tried restarting the kernel, closing and reopening the program, and even restarting my machine. Unfortunatelly I don't have time to dig any further in the web or talk about this in OH. I made a post on piazza about this, apparently some people had the same issue.

In [ ]:
```
image = generator.generate_images(guidance=0.5, num_samples=8)
plot_image(image)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[18], line 1
----> 1 image = generator.generate_images(guidance=0.5, num_samples=8)
      2 plot_image(image)

File ~/CS7643/Assignments/hw4_code_student_version/part2_DiffusionModels/computer_vision/image_gen.py:119, in ImageGenerator.generate_images(self, guidance, num_samples, threshold, class_label)
    113     all_labels = [class_label] * num_samples * 3
    115 # The size of this conditioning tensor is different than what you would expect given the function signature of generate_sample
    116 # If you are looking at this before implementing generate_sample, the first dimension in this cond tensor is still the batch size
    117 # The generate_sample function signature reflects the shape of the conditioning tensor for the 1D prediction network (part 2.4)
    118 # If none of this makes sense to you, that is fine, you do not need to worry about any of this
--> 119 cond = torch.tensor(all_labels).to(self.policy.device)
    121 # generate the image
    122 image = self.policy.generate_sample(cond, guidance_weight = guidance, threshold=threshold)

RuntimeError: CUDA error: device-side assert triggered
CUDA kernel errors might be asynchronously reported at some other API call, so the stacktrace below might be incorrect.
For debugging consider passing CUDA_LAUNCH_BLOCKING=1.
Compile with `TORCH_USE_CUDA_DSA` to enable device-side assertions.
```

In [ ]:
```
image = generator.generate_images(guidance=1, num_samples=8)
plot_image(image)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[19], line 1
----> 1 image = generator.generate_images(guidance=1, num_samples=8)
      2 plot_image(image)

File ~/CS7643/Assignments/hw4_code_student_version/part2_DiffusionModels/computer_vision/image_gen.py:119, in ImageGenerator.generate_images(self, guidance, num_samples, threshold, class_label)
    113     all_labels = [class_label] * num_samples * 3
    115 # The size of this conditioning tensor is different than what you would expect given the function signature of generate_sample
    116 # If you are looking at this before implementing generate_sample, the first dimension in this cond tensor is still the batch size
    117 # The generate_sample function signature reflects the shape of the conditioning tensor for the 1D prediction network (part 2.4)
    118 # If none of this makes sense to you, that is fine, you do not need to worry about any of this
--> 119 cond = torch.tensor(all_labels).to(self.policy.device)
    121 # generate the image
    122 image = self.policy.generate_sample(cond, guidance_weight = guidance, threshold=threshold)

RuntimeError: CUDA error: device-side assert triggered
CUDA kernel errors might be asynchronously reported at some other API call, so the stacktrace below might be incorrect.
For debugging consider passing CUDA_LAUNCH_BLOCKING=1.
Compile with `TORCH_USE_CUDA_DSA` to enable device-side assertions.
```

In [ ]:
```
image = generator.generate_images(guidance=2, num_samples=8)
plot_image(image)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[20], line 1
----> 1 image = generator.generate_images(guidance=2, num_samples=8)
      2 plot_image(image)

File ~/CS7643/Assignments/hw4_code_student_version/part2_DiffusionModels/computer_vision/image_gen.py:119, in ImageGenerator.generate_images(self, guidance, num_samples, threshold, class_label)
    113     all_labels = [class_label] * num_samples * 3
    115 # The size of this conditioning tensor is different than what you would expect given the function signature of generate_sample
    116 # If you are looking at this before implementing generate_sample, the first dimension in this cond tensor is still the batch size
    117 # The generate_sample function signature reflects the shape of the conditioning tensor for the 1D prediction network (part 2.4)
    118 # If none of this makes sense to you, that is fine, you do not need to worry about any of this
--> 119 cond = torch.tensor(all_labels).to(self.policy.device)
    121 # generate the image
    122 image = self.policy.generate_sample(cond, guidance_weight = guidance, threshold=threshold)

RuntimeError: CUDA error: device-side assert triggered
CUDA kernel errors might be asynchronously reported at some other API call, so the stacktrace below might be incorrect.
For debugging consider passing CUDA_LAUNCH_BLOCKING=1.
Compile with `TORCH_USE_CUDA_DSA` to enable device-side assertions.
```

In [ ]:
```
image = generator.generate_images(guidance=5, num_samples=8)
plot_image(image)
```

```
---------------------------------------------------------------------------
-
RuntimeError                              Traceback (most recent call las
t)
Cell In[21], line 1
----> 1 image = generator.generate_images(guidance=5, num_samples=8)
      2 plot_image(image)

File ~/CS7643/Assignments/hw4_code_student_version/part2_DiffusionModels/c
omputer_vision/image_gen.py:119, in ImageGenerator.generate_images(self, g
uidance, num_samples, threshold, class_label)
    113     all_labels = [class_label] * num_samples * 3
    115 # The size of this conditioning tensor is different than what you
would expect given the function signature of generate_sample
    116 # If you are looking at this before implementing generate_sample,
the first dimension in this cond tensor is still the batch size
    117 # The generate_sample function signature reflects the shape of the
conditioning tensor for the 1D prediction network (part 2.4)
    118 # If none of this makes sense to you, that is fine, you do not nee
d to worry about any of this
--> 119 cond = torch.tensor(all_labels).to(self.policy.device)
    121 # generate the image
    122 image = self.policy.generate_sample(cond, guidance_weight = guidan
ce, threshold=threshold)

RuntimeError: CUDA error: device-side assert triggered
CUDA kernel errors might be asynchronously reported at some other API call
, so the stacktrace below might be incorrect.
For debugging consider passing CUDA_LAUNCH_BLOCKING=1.
Compile with `TORCH_USE_CUDA_DSA` to enable device-side assertions.
```

Question : What do you notice about the generations with classifier free guidance? What features does the model pick up as being most indicattive of each class? Does performance degrade as guidance increases? If so, how?

Your Answer Here : Unfortunatelly, I am running into CUDA errors, the most frequent being `RuntimeError: cuDNN error: CUDNN_STATUS_INTERNAL_ERROR`, therefore I can not see the images nor draw conclusions about it. Below are some of the things I have tried, but it is also important to note that I did try many different numbers for `self.condition_dim`, both in the int format and tuple of ints.

I am running locally, in VSCode and already tried restarting the kernel, closing and reopening the program, and even restarting my machine. Unfortunatelly I don't have time to dig any further in the web or talk about this in OH. I made a post on piazza about this, apparently some people had the same issue.

2.3.3 Thresholding [2 points] + [3 writeup points]

You may have noticed that with high guidance weights, the images tend to oversaturate. This is because the model is predicting values outside of the range [-1, 1] (our input is normalized to this range before being passed to the model). To help mitigate this, it is common to threshold sample predictions at each denoising step. As our denoising step is currently implemented, we can't actually use this technique. This is because we directly estimate x_{t-1}. Why shouldn't we just threshold x_{t-1} to be within

[-1, 1]? Additionally, why is thresholding particularly useful when dealing with large guidance weights? Will thresholding have an effect even if we don't have a large guidance weight?

Your Answer Here: For the coding answer, you can see that I run this above, and it works (I'll try to link that page in Gradescope).

For high quality, we need to play with the values of the bounds for thresholding, so I see it as hyperparameters: it might have a good or a bad effect depending on the values you choose. If we do constrain it to [-1, 1] though, we might make the noise prediction outside what is within the true distribution that we are trying to learn, and maybe also lose data (whatever is outside that interval) in the process.

Although, when dealing with large guidance weights, thresholding will do some clipping to outputs that are probably not wanted, which is a heuristic to deal with outputs that are non-representative of the distribution we are learning.

Even if we don't have a large guidance weight, threshold can still have an effect of preventing the generation of pixel values that are off (too dark or too bright), and/or outliers of the image distribution ("artifacts").

Disclosure: Unfortunatelly, I am running into CUDA errors, the most frequent being `RuntimeError: cuDNN error: CUDNN_STATUS_INTERNAL_ERROR` , therefore I can not see the images nor draw conclusions about it. Below are some of the things I have tried, but it is also important to note that I did try many different numbers for `self.condition_dim` , both in the int format and tuple of ints.

I am running locally, in VSCode and already tried restarting the kernel, closing and reopening the program, and even restarting my machine. Unfortunatelly I don't have time to dig any further in the web or talk about this in OH. I made a post on piazza about this, apparently some people had the same issue.

The answer above is what makes sense to me + some research that I did.

So we need to compute x_{t-1} such that we estimate x_0 as an intermediate step. This way we can threshold our estimate of x_0.
This is shown in equations (6) and (7) of the DDPM paper. The equation for computing an estimate of x_0 is shown in equation (15). Make sure to threshold the prediction of x_0!

TODO: Implement denoising with thresholding in the `denoise_step` function in `DiffusionModels/noise_scheduler.py`

In [ ]:
```
# test denoise step with thresholding
unit_tester.test_threshold_denoise_step()
```

Now that you have completed the thresholding, evaluate how this changes image generation with different guidance weights.

In [ ]:
```
image = generator.generate_images(guidance=0, num_samples=8, threshold=Tr
plot_image(image)
```

In [ ]:
```
image = generator.generate_images(guidance=0.5, num_samples=8, threshold=
plot_image(image)
```

In [ ]:
```
image = generator.generate_images(guidance=1, num_samples=8, threshold=Tr
plot_image(image)
```

In [ ]:
```
image = generator.generate_images(guidance=2, num_samples=8, threshold=Tr
plot_image(image)
```

In [ ]:
```
image = generator.generate_images(guidance=5, num_samples=8, threshold=Tr
plot_image(image)
```

Question: How do the results after thresholding compare to the results before thresholding? With thresholding and high guidance weight, which class specific features are most prominant? Which set of parameters gave you the best output?

Your Answer Here: I guess with thresholding it would perform better.

Unfortunatelly, I am running into CUDA errors, the most frequent being `RuntimeError: cuDNN error: CUDNN_STATUS_INTERNAL_ERROR`, therefore I can not see the images nor draw conclusions about it. Below are some of the things I have tried, but it is also important to note that I did try many different numbers for `self.condition_dim`, both in the int format and tuple of ints.

I am running locally, in VSCode and already tried restarting the kernel, closing and reopening the program, and even restarting my machine. Unfortunatelly I don't have time to dig any further in the web or talk about this in OH. I made a post on piazza about this, apparently some people had the same issue.

TODO: Experiment with different training times and guidance weights! Try and get the best image possible (or show something else interesting). You need at least two more generations, feel free to do more are welcome though.

In [ ]:
```
# room for experimentation
```

Collect Submission

Run the following cell to collect `assignment3_part2_submission.zip`. You will submit this to HW3 Code - Part 2 on gradescope. Make sure to also export a PDF of this jupyter notebook and attach that to the end of your theory section. This PDF must show your answers to all the questions in the document, please leave in all the photos that you generate as well. You will not be given credit for anything that is not visible to us in this PDF.

In [ ]:
```
%%bash collect_submission.sh
```

Contributers

- Matthew Bronars (Lead)
- Manav Agrawal
- Mihir Bafna