DeepLense Tests

ML4SCI, Google Summer of Code

Namritha Maddali

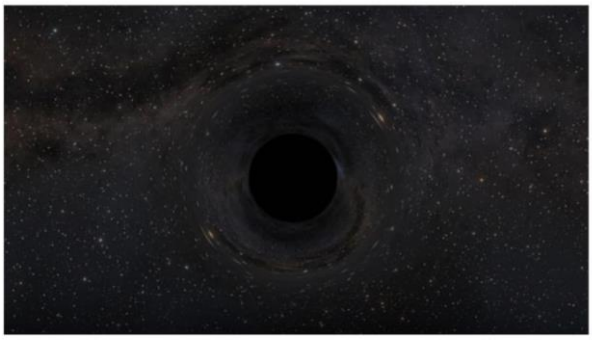
namritha.maddali@gmail.com

Test IV – Classification of Gravitational Lensing Images

https://github.com/namritha-maddali/ Generation-of-Gravitational-Lensing-Images

Introduction

Gravitational Lensing occurs when an extremely massive celestial body causes curvature in spacetime, visibly bending the light around it. Gravitational Lens is the term used to refer to these bodies. Basically, the **light from sources like other galaxies and stars are redirected around the gravitational lens, making the light look "bent"**. A famous example of this can be seen in the theorized black hole images



The Task at hand: Train a diffusion model to simulate gravitational lensing

Dataset: 10,000 Strong Lensing Images

Solution:

The Idea

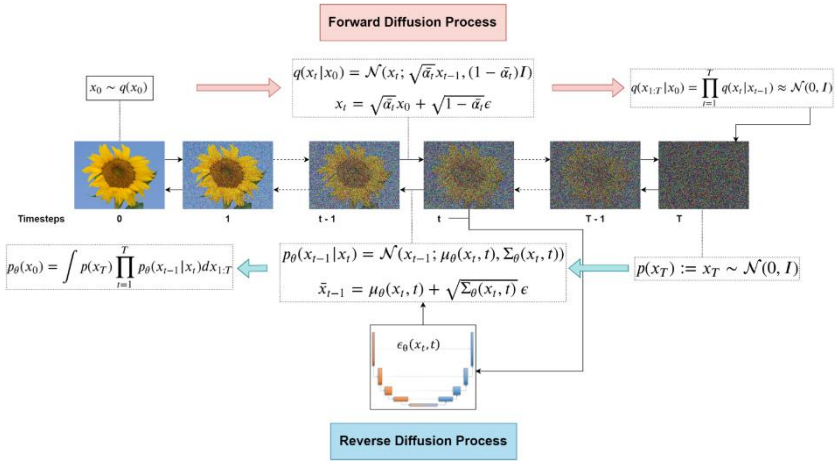
- Diffusion probabilistic models are a form of generative models (like GANs, autoregressive models, etc). They are capable of generating high quality images, sometimes images of much better quality than the training images.
- It is based on Markov Chain
  - This is a machine learning model where the future state depends on the current state only and not any of the past states, and is widely used for probabilistic modelling.
- **Forward Diffusion Process:**

  $$X_0 \text{ to } X_T$$

  - The process starts with a clear image of the target
  - As the steps increase, Gaussian noise is gradually added to the input
- **Reverse Diffusion Process:**

  $$X_T \text{ to } X_0$$

  - A neural network (here U-Net) is trained to remove the noise and reconstruct the image.



The Trained Model: `denoising_diffusion_pytorch`

- The model architecture chosen here uses UNet as the base neural network to reconstruct images from noise, and Gaussian Noising for the forward process.
- U-Net is a convolution neural network that was designed for image segmentation
  - The encoder captures features and the decoder restores spatial resolution of the image.
  - It learns and predicts the noise to reconstruct the clean image.

```
model = Unet(
    dim = 64,
    dim_mults = (1, 2, 4, 8),
    channels = 1,
    flash_attn = True
)

diffusion = GaussianDiffusion(
    model,
    image_size = 128,    # images should be of size 128x128
    timesteps = 1000     # number of steps
)
```

- Algorithm 1 is the forward diffusion process, where the gradient descent step happens on the Gaussian Noise function.
- Algorithm 2 on the other hand is the sampling process, that is the reverse diffusion process

Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \text{Uniform}(\{1, \dots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on
$$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$$
6: **until** converged

Algorithm 2 Sampling

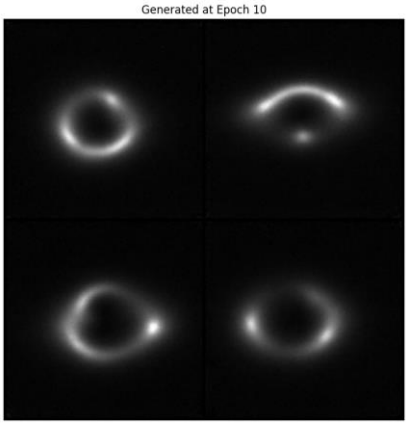
1: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2: **for** $t = T, \dots, 1$ **do**
3: $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
4: $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t) \right) + \sigma_t \mathbf{z}$
5: **end for**
6: **return** $\mathbf{x}_0$

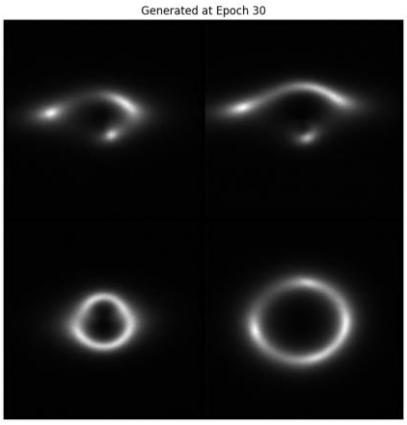
The Training Process

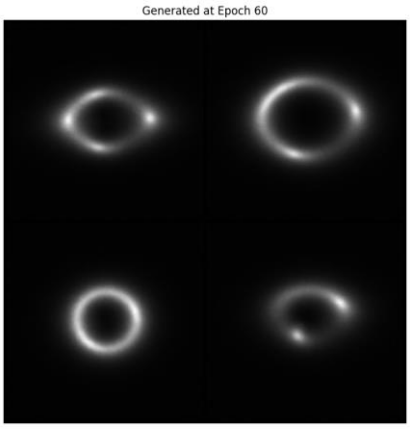
- The 10,000 images are divided into 24 classes and have been resized to 128x128, to accommodate for the restrictions on compute.
- And this is how the diffusion model is applied to the data (in image on the right)
- Optimization techniques like gradient clipping and Adam optimizer have been implemented as well.

```
for batch in tqdm(dataloader):
    batch = batch.to(device)
    optimizer.zero_grad()  # reset gradients
    loss = diffusion(batch) # forward
    loss.backward() # backprop
```

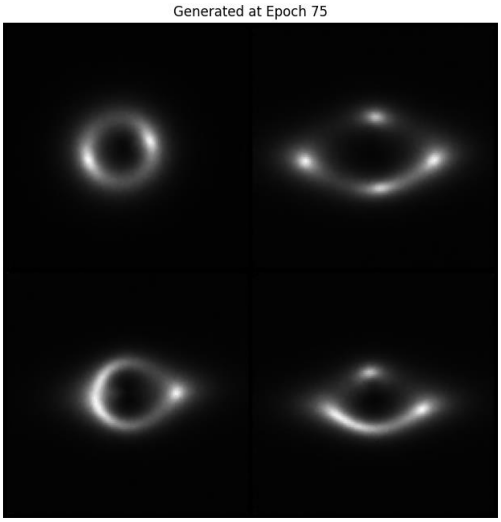
- During the training process, along with the loss incurred at every epoch, the model's performance after every 10 epochs is visualized by generating a few images at these checkpoints.







- The loss observed in the first 10 epochs was actually very low, close to 0.007. And as the epochs increased, the loss kept reducing steadily.
- From the 60th epoch to the 75th epoch, the reduction in loss was very minimal, but the training was continued as it is essential for the model to capture as many features as possible.
- The images generated by the model after 75 epochs were incredible! They were identical to the original testing images, if not even better in clarity



- The model was then saved into a pth file (which is available in the github repository)

Quantifying Realism of Generated Images

- Once the model has been trained, now it is time to test out how good the generated images are.
- In another notebook, about 10,000 new images were generated using the trained model in order to draw reliable results from any comparisons drawn. (the output of the image generation has been redacted from the notebook as it was too long)

  `Codebase for generation:` https://www.kaggle.com/code/nomviktor/generating-data-using-diffusion-model?scriptVersionId=230391215
  `version 1: data collection process, output contains the 10k images generated`

- Once the images were generated and loaded into pytorch dataloaders, multiple comparison metrics were used on the real and generated images

Fréchet Inception Distance (FID)

- This is used to quantify the realism, diversity and quality of the images generated by a model like a GAN or diffusion model.
- This uses real images as the ground truth and performs the comparative study for the generated images.
- A low score of FID means that the generated images and the real images are very close to each other in terms of similarity.
- For the FID to work, image features must be extracted. And to do this, a custom ResNet architecture has been used (from the common test docs)
- **The FID score obtained is close to 0.0009, showing that our generated images are very close to the real images (if not too close!)**

```
[170]  evaluator.run(zip(generated_loader, real_loader))
```

```
State:
        iteration: 157
        epoch: 1
        epoch_length: 157
        max_epochs: 1
        output: <class 'NoneType'>
        batch: <class 'NoneType'>
        metrics: <class 'dict'>
        dataloader: <class 'zip'>
        seed: <class 'NoneType'>
        times: <class 'dict'>
```

Renyi Kernel Entropy

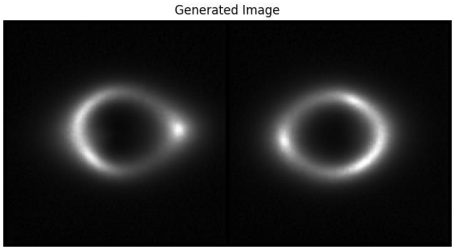
- Another metric used is Renyi Kernel Entropy, that was referenced in the Awesome Evaluation of Visual Generation github repository.
- This is an evaluation score based on quantum information theory to measure the number of modes in generated samples

```
[230]  print(kernel.compute_rke_mc(generated_features))
       print(kernel.compute_rrke(real_features, generated_features))
```

```
{0.2: 1.0261691817562502, 0.3: 1.0116372548484103, 0.4: 1.0065493815757378}
{0.2: 0.025132607651035647, 0.3: 0.0113372504605527, 0.4: 0.0064108677585926815}
```

- From the obtained Renyi Kernel Entropy values at different bandwidth,
  - RKE or Renyi Kernel Entropy **values for the generated features** is close to 1 so there is high entropy
  - RRKE or Renyi Kernel Entropy between the real and generated features is very small, showing that they are fairly similar, and as the bandwidth increases the entropy reduces

From these two metrics, we can see that our generated images resemble gravitational lensing very well, and our model has learnt well.



Conclusion

We have trained a model on 10k image to simulate gravitational lensing in images, generated some more images, and performed quantifying metrics on it

References:

- https://arxiv.org/pdf/2006.11239
- https://github.com/lucidrains/denoising-diffusion-pytorch
- https://github.com/ziqihuangg/Awesome-Evaluation-of-Visual-Generation
- https://github.com/mjalali/renyi-kernel-entropy
- https://pytorch-ignite.ai/blog/gan-evaluation-with-fid-and-is/
- https://pytorch.org/ignite/generated/ignite.metrics.FID.html