2024

Generative Models for Visual Signals

F74094017 資訊 113 李昆翰 (GITHUB: RANDYUNCLE)

內容

Abstract

In the contemporary landscape of artificial intelligence, generative models have demonstrated remarkable capabilities, particularly in image generation and transformation tasks. Denoising Diffusion Probabilistic Models (DDPMs) have emerged as a powerful framework for generating high-quality images through a progressive denoising process. However, the iterative nature of DDPMs, involving a large number of diffusion steps, often results in prolonged training and sampling times. This research explores an innovative approach to expedite the DDPM training process by leveraging Deep Image Prior (DIP), a technique known for its ability to capture rich image structures without requiring pre-training.

The proposed method involves initializing the DDPM with a prior generated by a DIP model, trained briefly on target images to capture high-level structural information. This DIP-based initialization provides a more informative starting point for the DDPM, potentially reducing the number of diffusion steps needed for convergence. I conduct extensive experiments using the CIFAR-10 dataset, applying custom transformations to adapt the images for the DIP model and subsequently for the DDPM. The effectiveness of this approach is evaluated through quantitative metrics Root Mean Squared Error (RMSE), as well as qualitative assessments of the generated samples.

You could find my implementations and experiments in the following GitHub repository: https://github.com/randyuncle/Generative-Models-for-Visual-Signals

Deep Image Prior (DIP)

Introduction

Deep Image Prior (DIP) is a concept introduced to leverage the inherent structure and capabilities of convolutional neural networks (CNNs) for image restoration tasks, without the need for pre-training on large datasets. Unlike traditional approaches that require extensive training on external data, DIP demonstrates that the architecture of a CNN itself can act as a powerful prior for natural images.

The fundamental idea behind DIP is that a randomly initialized CNN, when trained to map from **a random noise input** to a given corrupted image, can effectively capture the underlying structure of the image before overfitting to the noise. This process is possible because CNNs are inherently biased towards generating natural images due to their convolutional nature, which favors the smoothness and continuity present in most real-world images.

Structure and the hyperparameters that I used

The structure of Deep Image Prior (DIP) is implemented in a U-net-like hourglass neural network, which is a fully convolutional encoder-decoder architecture often combined with skip connections. The authors illustrate this structure in Figure 21 of the original paper.

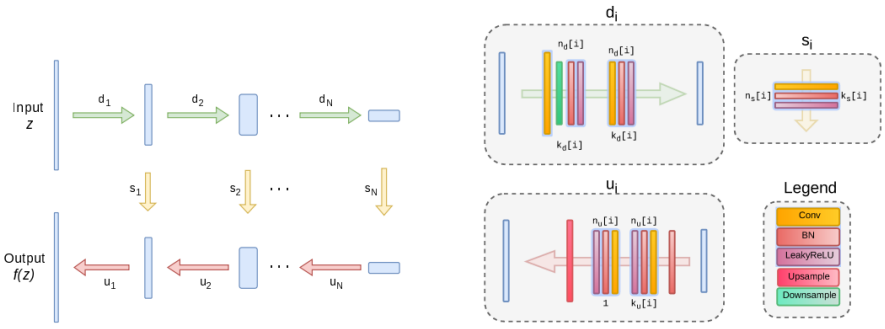


Fig. 21: **The architecture used in the experiments.** We use "hourglass" (also known as "decoder-encoder") architecture. We sometimes add skip connections (yellow arrows). $n_u[i]$, $n_d[i]$, $n_s[i]$ correspond to the number of filters at depth i for the upsampling, downsampling and skip-connections respectively. The values $k_u[i]$, $k_d[i]$, $k_s[i]$ correspond to the respective kernel sizes.

圖表 1

The concrete elements of the structure, such as the U-net, skip architecture, and other implementations, can be found in the source code provided by the authors, which is implemented in PyTorch. My implementation follows this model, specifically the hourglass network with skip connections, and can be seen in the `main.ipynb` file in my GitHub repository.

In my implementation, I adhered to the descriptions in Section 4 of the original paper. The LeakyReLU function is used for non-linearity, and the downsampling technique employed involves strides (as default) integrated within the convolutional modules. Both nearest neighbor and bilinear upsampling is implemented in the code. Reflective padding is applied in all convolutional layers, and random noise for the input ***z*** is generated using Bernoulli sampling. The ADAM optimizer is used for optimization.

Furthermore, the original source code is not only challenging to configure for different upsampling, downsampling, and other structures but also difficult to debug. Therefore, I implemented my work with reference to RPraneetha's team on GitHub. They implement each structure of the network in individual classes and construct the skip architecture of the hourglass neural network in a class with layer block-based initialization. This approach makes the code easier to read and debug, which influenced my implementation. However, for the convolutional layers, and the other utilities in `utils.py` in my source code, I used the method from the original source code to allow easy switching between different convolutional layer architectures.

For the training code, I adopted most of the original source code and made adjustments to accommodate different versions of some Python packages. The hyperparameters for the network architecture in this assignment follow the super-resolution task, which is the default architecture for DIP, as described in the original paper.

Super-resolution (*default architecture*).

$z \in \mathbb{R}^{32 \times W \times H} \sim U(0, \frac{1}{10})$
$n_u = n_d =$ `[128, 128, 128, 128, 128]`
$k_u = k_d =$ `[3, 3, 3, 3, 3]`
$n_s =$ `[4, 4, 4, 4, 4]`
$k_s =$ `[1, 1, 1, 1, 1]`
$\sigma_p = \frac{1}{30}$
`num_iter` $= 2000$
`LR` $= 0.01$
`upsampling` $=$ `bilinear`

圖表 2

Denoising Diffusion Probabilistic Model (DDPM)

Introduction

Denoising Diffusion Probabilistic Models (DDPM) represent a class of generative models that have shown exceptional performance in producing high-quality images. Unlike traditional generative models like GANs or VAEs, DDPMs generate images through a process of gradual denoising of a noisy image.

The core idea behind DDPMs involves two main stages: a forward diffusion process and a reverse denoising process. The “forward diffusion process” gradually adding Gaussian noise to given clean images over step of step, which is been called “timesteps”; on the other hand, the “reverse denoising process” is the step that trains the neural network by the task of reversing the forward diffusion process by staring from pure noises, effectively reconstructing the original image.

The strengths of DDPMs include their ability to generate highly diverse and high-fidelity images, thanks to the stable training process that avoids many of the pitfalls encountered in GAN training, such as mode collapse. Additionally, the probabilistic framework allows for better theoretical understanding and analysis.

However, a notable drawback is the computational inefficiency due to the large number of diffusion steps required for generating an image making DDPMs slower compared to other generative models (though it runs normally with CIFAR10 dataset in my device). Despite this, recent advancements and optimizations continue to enhance their efficiency and applicability in various domains of generative AI, and continue presenting new frameworks that are inspired by this method.

Structure of DDPM

According to the additional information provided in the original DDPM paper, the authors implemented their DDPM using a U-net architecture based on Wide Residual Networks, replacing weight normalization with group normalization. All models have two convolutional residual blocks per resolution level, as recommended in the Wide Residual Networks paper, and self-attention blocks at the 16x16 resolution between convolutional blocks. Additionally, the diffusion time $\boldsymbol{t}$ is specified by adding the Transformer sinusoidal positional embedding into each residual block. For 32x32 models, they used four feature map resolutions, which decompose from 32x32 to 4x4, while their 256x256 models use six feature map resolutions.

The source code was built using the TensorFlow framework, as the DDPM experiments were conducted with TPU. However, I am not familiar with writing neural networks in TensorFlow, and the original implementation of DIP was in PyTorch. Therefore, I searched on GitHub for any PyTorch implementations and found many created by individuals. To understand how the neural network layers and DDPM architecture were implemented in those repositories and the original source code, I read through the referenced paper and examined the original and other individual implementations.

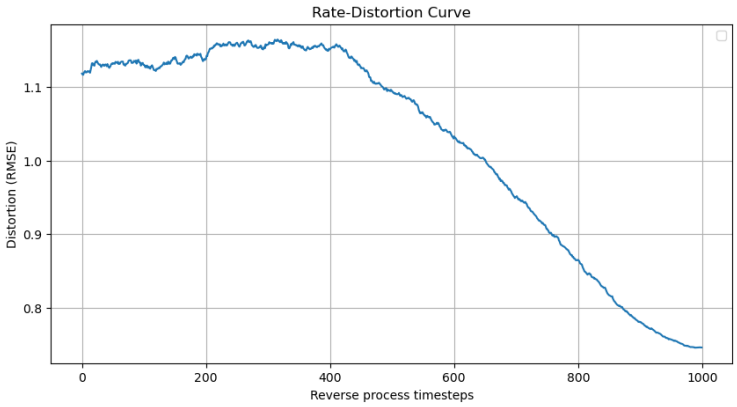
My final implementation can be found in my GitHub repository. Most of my work is inspired by mattroz's repository, which structured the DDPM in a PyTorch-based pattern and constructed neural network layers with easily readable classes and comments. In this repository, the U-net for DDPM training was constructed with five feature map resolutions for 256x256 models. The main differences between my work and mattroz's are the simplification and symbolization of some network layer blocks, the introduction of pre-calculations from the original source code in the initialization of the denoising diffusion probabilistic framework, increased usage of the reverse diffusion function implemented in the framework, and the incorporation of the DIP output in the forward diffusion function.

The selected testing in DDPM

The training and experiments I would done in DDPM is the CIFAR10 datasets. The network is optimized by Adam with learning rate in $2x10^{-4}$, and the training images would be transform with random horizontal flips. The beta setting would be schedule between 10^{-2} to 10^{-4} with learning epoch 100. Since the selected dataset is CIFAR10, I also set the batch size to 128.

For the pure DDPM training, I saved the images at every 5 epochs in the directory `output_img/1000T`. To facilitate comparison between different DDPM implementations, I also created code for generating a rate-distortion vs. time plot, similar to the leftmost plot in Fig. 5 of the original paper. This includes RMSE calculations in the notebook `main.ipynb`.

The result of the pure DDPM training is reflected in the following rate-distortion vs. time plot.



圖表 3

The trend of this plot closely matches the one presented in the original paper.

Ideas and Experiments on accelerating DDPM by DIP

Most of the experiments are documented with well-written git commit messages, which I would recommend readers to read the following descriptions with the git commit messages.

Replace gaussian noise in forward diffusion of DDPM by DIP

Description of the Idea

This thought is to replace the gaussian noise in the forward diffusion process in the DDPM framework into the output of DIP with the pure noise input, hoping to contract the high resolution of the noise in the prior during forwarding.

The code of this idea could be seen in commit 674c7b9. The following is the screen shot of the idea.

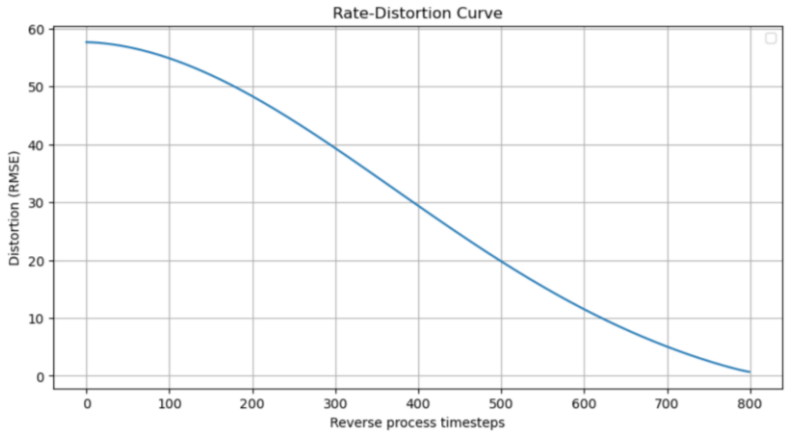
```python
def forward_diffusion(self, images, timesteps, dip_model, batch_size) -> tuple[torch.Tensor, torch.Tensor]:
    """The forward function of diffusion probabilistic model

    Args:
        images: The input data
        timesteps: Current forwarding time
    """
    noise = dip_model(get_noise(32, "noise", (32, 32)).to(images.device).detach()).repeat(batch_size, 1, 1, 1).to(images.device) \
            if dip_model else torch.randn(images.shape).to(images.device)
    self.alphas_hat = self.alphas_hat.to(images.device)
    alpha_hat = self.alphas_hat[timesteps]
    # alphas_sprt = self.alphas_sprt[timesteps].to(images.device)
    # alphas_one_minus_sqrt = self.alphas_one_minus_sqrt[timesteps].to(images.device)

    alpha_hat = broadcast(alpha_hat, images)

    # The calculation of the forward loss function `Lsimple(0)`,
    # which is the equation (14) in the original paper
    return torch.sqrt(alpha_hat) * images + torch.sqrt(1 - alpha_hat) * noise, noise
```

圖表 4

Experiments

I experiment with the 5 epoch of DIP training, and 800 time steps for DDPM training with the input noise and sampling noise for evaluation be the output of DIP.

I saved the images at every 5 epochs in the directory `output_img/ 800T_dip5`. The following is the quantitative result in rate-distortion vs. time plot of this experiment.



圖表 5 (Note that the timesteps of the plot is actually opposite, due to some coding errors)

From the output images and the plot, it is evident that this approach is not effective for accelerating the DDPM training process. The images are entirely white, and the RMSE in this plot is quite large, indicating increased difficulty in training with such noise.

Make the output of DIP be the target of loss function

Description of the Idea

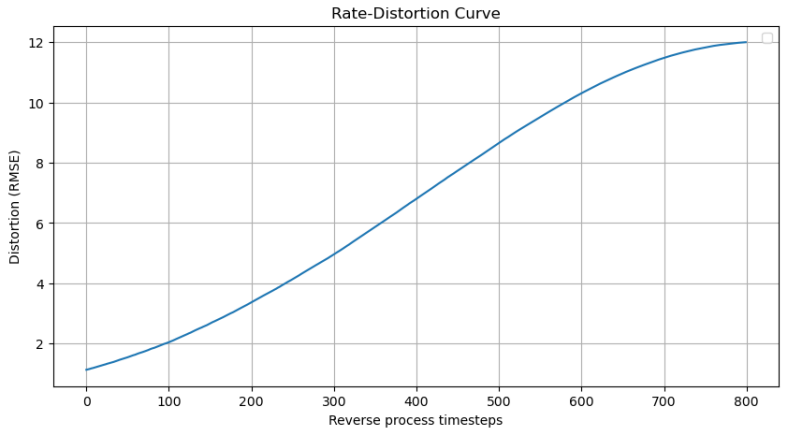
Rather than making the output of DIP to be the input noise of DDPM in forward diffusion and evaluation sampling, I make the output of DIP to be the target noise for the loss function while training the neural network of DDPM.

The code of this thought could be seen in commit 4bd09b3.

Experiments

I experiment with the 1000 epoch of DIP training, and 800 time steps for DDPM training with the input noise and sampling noise for evaluation be the output of DIP.

I saved the images at every 5 epochs in the directory `output_img/ 800T_dip1000_new`. The following is the quantitative result in rate-distortion vs. time plot of this experiment.



圖表 6 (In contrast to the previous experiment, timesteps in this plot is correct)

From the output images and the plot, it is evident that this approach, like the previous one, is not effective for our goal. The output images are filled with noise, and the plot results indicate that this method performs worse than the previous one. The lowest RMSE is higher than that of the previous approach, demonstrating its inferiority.

Conclusions and Thoughts for this assignment

In conclusion, the current thoughts on accelerating DDPM by DIP aren't the better way to generate the well and clean image, and they also slows the DDPM training time even if I reduce the diffusion steps of DDPM. From the rate-distortion v.s. time plot and the resulting images during training I uploaded to the repository, it indicates that those methods might need more training epochs, or even some are the wrong implementations.

For me, the biggest challenge in this assignment was constructing the two frameworks, which consumed most of my time and left little for experimentation. This was somewhat disappointing. The primary reason for this difficulty is my limited experience in building neural networks; I mostly rely on API calls in my bachelor learning, which felt strange in this context that a CS student didn't code for such tasks. Consequently, I struggled to understand how to implement, simplify, and interpret the source code or code from other GitHub repositories. This led me to read numerous papers on specific structures, such as Wide Residual Networks, and learn how self-attention and partial embedding techniques were implemented in the original papers and source code, as well as by others on GitHub who built them in PyTorch. This experience highlighted my lack of coding practice for neural networks in Python, providing a valuable opportunity to improve. Although some of my code was still copied from GitHub sources, this process helped me better understand

neural network construction and gain hands-on experience in evaluating GAI through research, implementation, and code reading.

References

[1] Ulyanov, D., Vedaldi, A., Lempitsky, V. (2018). Deep Image Prior. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 9446–9454).

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. In Advances in Neural Information Processing Systems, pages 5998–6008, 2017.

[3] Sergey Zagoruyko and Nikos Komodakis. Wide Residual Networks. arXiv preprint arXiv:1605.07146, 2016.

[4] Jascha Sohl-Dickstein, Eric A. Weiss, Niru Maheswaranathan, Surya Ganguli. Deep Unsupervised Learning using Nonequilibrium Thermodynamics. arXiv:1503.03585, 2015.

[5] Jonathan Ho and Ajay Jain and Pieter Abbeel. Denoising Diffusion Probabilistic Models. arXiv preprint arxiv:2006.11239, 2020.

[6] RPraneetha and debadeep93. Reproducing Inpainting and Image Restoration from Deep Image Prior. https://github.com/RPraneetha/DL-RP41-Deep-Image-Prior/tree/master

[7] mattroz. diffusion-DDPM. https://github.com/mattroz/diffusion-ddpm/tree/main

[8] lucidrains, et al.. denoising-diffusion-pytorch. https://github.com/lucidrains/denoising-diffusion-pytorch/tree/main

[9] tqch. PyTorch Implementation of Denoising Diffusion Probabilistic Models. https://github.com/tqch/ddpm-torch/tree/master?tab=readme-ov-file