Denoising Diffusion Implicit Models (DDIM)

Title:

Denoising Diffusion Implicit Models

Authors: Jong Jin Lee, Sangwoo Cho, Michele T. S. H. Ong, Ting-Wu Chin, Yuan-Ting Chen

Published: 2021

Abstract:

The DDIM paper presents a family of generative models called Denoising Diffusion Implicit Models, which is short for DDIMs. These models operate based on reversing a diffusion process in which data is generated through the gradual addition of noise and then learning how to reverse this process in order to recover the original data. The innovation presented by DDIMs is that they provide a non-Markovian framework, thereby allowing implicit sampling without step-by-step Markovian sampling, hence sampling more quickly and efficiently.

Implementation Details

U-Net Architecture

U-Net consists of a contracting path and an expansive path. The contracting path is a series of convolutional layers and pooling layers, where the resolution of the feature map gets progressively reduced. Expansive path is a series of up-sampling layers and convolutional layers where the resolution of the feature map gets progressively increased. At every step in the expansive path the corresponding feature map from the contracting path concatenated with the current feature map.

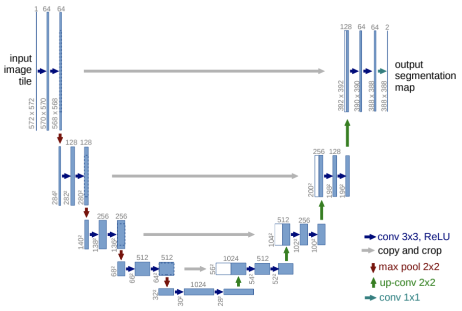


Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

DDIM Sampler

1. Noise Term (σ_i):

$$\sigma_i = \eta \sqrt{\frac{(1-\alpha_{\tau_{i-1}})}{(1-\alpha_{\tau_i})} \cdot \left(1 - \frac{\alpha_{\tau_i}}{\alpha_{\tau_{i-1}}}\right)}$$

2. Predicted Image (x_0):

$$x_0 = \frac{1}{\sqrt{\alpha_{\tau_i}}} x_t - \frac{1}{\sqrt{\alpha_{\tau_{i-1}}}} \epsilon_\theta(x_t)$$

3. Next Sample (x_{t-1}):

$$x_{t-1} = \text{mean} + \sigma_i \cdot \text{noise}$$

4. Mean Calculation:

$$\text{mean} = \sqrt{\alpha_{\tau_{i-1}}} x_0 + \text{coeff}_i \cdot \epsilon_\theta(x_t)$$

5. Coefficient Calculation (coeff_i):

$$\text{coeff}_i = \sqrt{1 - \alpha_{\tau_{i-1}} - \sigma_i^2}$$

Key Improvement

```
def compute_complexity(self, x):
    # Add padding to maintain input dimensions
    padding = 1

    # Define a 2D kernel with 1 channel and then expand it to match the input channels (3 for
        RGB)
    kernel_x = torch.tensor([[[[-1, 1], [0, 0]]]], dtype=torch.float32).repeat(3, 1, 1, 1).to(x
        .device)
    kernel_y = torch.tensor([[[[-1, 0], [1, 0]]]], dtype=torch.float32).repeat(3, 1, 1, 1).to(x
        .device)

    # Compute gradients along x and y directions with padding
    grad_x = torch.abs(F.conv2d(x, kernel_x, padding=padding, groups=3))
    grad_y = torch.abs(F.conv2d(x, kernel_y, padding=padding, groups=3))

    # Calculate complexity based on gradients
    complexity = torch.sqrt(grad_x ** 2 + grad_y ** 2)

    # Normalize the complexity
    complexity = complexity / (torch.norm(complexity, p=2) + 1e-8)

    return complexity
```

```
def adaptive_step_size(self, complexity):
    dt = self.base_step_size / complexity
    return dt.clamp(min=self.min_step, max=self.max_step)

def modified_loss(self, pred_noise, true_noise, complexity, lambda_reg=0.1):
    # Ensure all tensors have the same spatial dimensions
    if complexity.shape != pred_noise.shape:
        complexity = F.interpolate(complexity, size=pred_noise.shape[2:], mode=’bilinear’,
            align_corners=False)

    # Base MSE loss
    squared_error = F.mse_loss(pred_noise, true_noise)

    # Complexity-weighted L1 loss
    complexity_term = torch.mean(complexity * torch.abs(pred_noise - true_noise))

    # Combined loss
    return squared_error + lambda_reg * complexity_term
```

These new functionality is being added in the code so that it can incorporates the Adaptive step size idea which is beneficial for the image quality improvement.

1. **Complexity Calculation:** This calculates the complexity of the input based on gradients.

$$\text{grad}_x = |x * K_x|, \quad \text{grad}_y = |x * K_y|$$

$$\text{complexity} = \frac{\sqrt{\text{grad}_x^2 + \text{grad}_y^2}}{\|\sqrt{\text{grad}_x^2 + \text{grad}_y^2}\|_2 + \epsilon}$$

2. **Adaptive Step Size:** The adaptive time step size, Δt, is computed based on complexity.

$$\Delta t = \frac{\text{base\_step\_size}}{\text{complexity}}$$

with clamping:

$$\Delta t = \text{clamp}(\Delta t, \text{min\_step}, \text{max\_step})$$

3. **Modified Loss Function:** Combines Mean Squared Error (MSE) and complexity-weighted L1 loss.

$$\text{squared\_error} = \frac{1}{N}\sum_{i=1}^{N}(\text{pred\_noise}_i - \text{true\_noise}_i)^2$$

$$\text{complexity\_term} = \frac{1}{N}\sum_{i=1}^{N} \text{complexity}_i \cdot |\text{pred\_noise}_i - \text{true\_noise}_i|$$

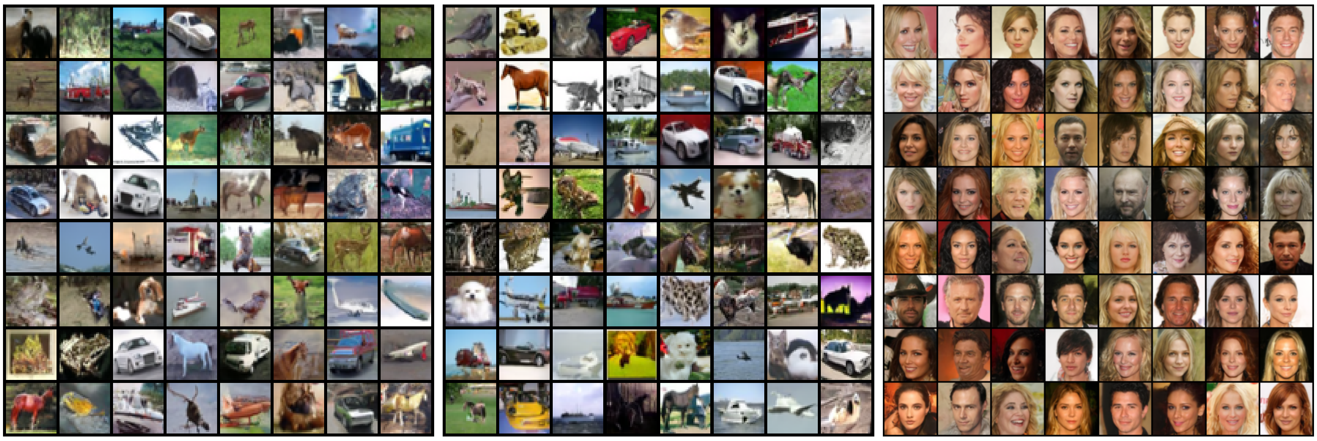
$$\text{loss} = \text{squared\_error} + \lambda \cdot \text{complexity\_term}$$

By incorporating these function in code it will reduce the training loss / reconstruction loss from 0.03-0.04 for cifar10 64-dim to 0.02-0.03 which shows an improvement in the results.

These equation are introduced in the location `./src/diffusion.py`

Results

Images



Cifar10 64-dimension(left), Cifar10 128-dimension(middle), Celeba hq (right)

FID score

Dataset	FID score	Reconstruction loss
Cifar10 64-dimension	21.87	0.02-0.03

Dataset Description

Cifar 10

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class.

CelebA-HQ

A dataset containing 30,000 high-quality celebrity faces, resampled to 256px.