DLCV Assignment 3

Harsh Vishwakarma

MTech, CSA

21532

Diffusion Model:

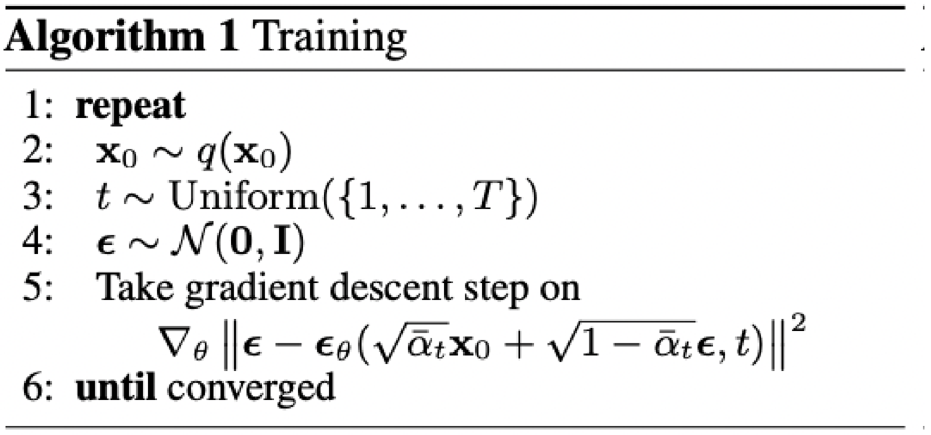
Denoising Diffusion Probabilistic Models (DDPMs) is a state-of-the-art generative model for image synthesis, proposed by Jonathan Ho and Ajay Jain in their paper titled "Denoising Diffusion Probabilistic Models" published in 2021. The model is built on the concept of diffusion processes, which are stochastic processes that describe the spread of particles in a medium. DDPMs utilize the denoising diffusion process to learn a generative model that can produce high-quality images.

The main objective of DDPMs is to learn a model that can generate high-quality images that are indistinguishable from real images. The model works by iteratively adding noise to an image and then attempting to denoise it. In each iteration, the model learns to estimate the distribution of the noise that is added to the image. This distribution is then used to generate a new image by sampling from it. This process is repeated multiple times to generate high-quality images.

Dataset:

The dataset is provided a series of human body key points, i.e., 20 key points in each dimension. There are 895 pairings in total, from which I have taken a clip of N length, which I have indicated below, and also illustrates the results using that N.

Training:



```
Algorithm 1 Training
1: repeat
2:   $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3:   $t \sim \mathrm{Uniform}(\{1, \dots, T\})$
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
```

The above algorithm is implemented to generate a N(0,I) noise in an iterative fashion, for each iteration a time step is takes uniformly at random from (1-T) and a noise is added to is, the loss function basically signifies that the L2 norm between the epsilon ~N(0,I) and a learnable noise for that time step minimizes.

MDM-UNet:

MDM-UNet model, a type of diffusion model used in image generation and other generative modeling tasks. The MDM-UNet model is a variant of the UNet architecture that uses multi-head attention mechanisms to better capture long-range dependencies and improve the model's ability to generate high-quality images.

The model takes several input arguments, including image size, input channels, output channels, and the number of residual blocks and attention resolutions to use. The model also has several optional arguments, such as dropout, channel multiplication factors.

The architecture consists of several blocks, including input blocks, output blocks, and an output module. The input blocks contain a series of residual blocks, each with its own multi-head attention mechanism to capture long-range dependencies in the input. The output blocks mirror the input blocks, but with each block's residual connections reversed. Finally, the output module applies normalization and activation functions before producing the final output image.

Code Implementation details:

Here are brief explanations of the main components of the code:

- The **MDM_UNetModel** class is defined, inheriting from **nn.Module**.

- The **__init__** method is defined, which initializes the parameters of the network, including the image size, number of input channels, number of output channels, number of residual blocks, and attention resolutions, among others.
- A time embedding layer is defined using a two-layer MLP with a SiLU activation function.
- An input block is defined, consisting of a TimestepEmbedSequential module with a convolutional layer of kernel size 3 and specified padding, followed by **num_res_blocks** ResBlock modules.
- The number of output channels is determined by the **channel_mult** parameter, which specifies the multiplier for each resolution level.
- The input block is repeated for each resolution level, with the output of the previous resolution level downsampled using either a convolutional resample or a ResBlock with a downsample option.
- An output block is defined for each resolution level, consisting of **num_res_blocks** ResBlock modules and a convolutional resample or a ResBlock with an upsample option to merge the features from the input block with the features from the corresponding output block.
- The final output is obtained by applying normalization, a SiLU activation function, and a convolutional layer of kernel size 3 with specified padding and padding mode.
- A **clip_dim** parameter is defined to clip the dimensionality of the output features, although it is currently hardcoded and not used.

The **MDM_UNetModel** module outputs a tensor representing the predicted segmentation masks for the input image.

Sampling:

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 sampling algorithm helps in the denoising process, in each iteration from T--1, x_{t-1} using x_t and epsilon (learned noise in that step during forward process).

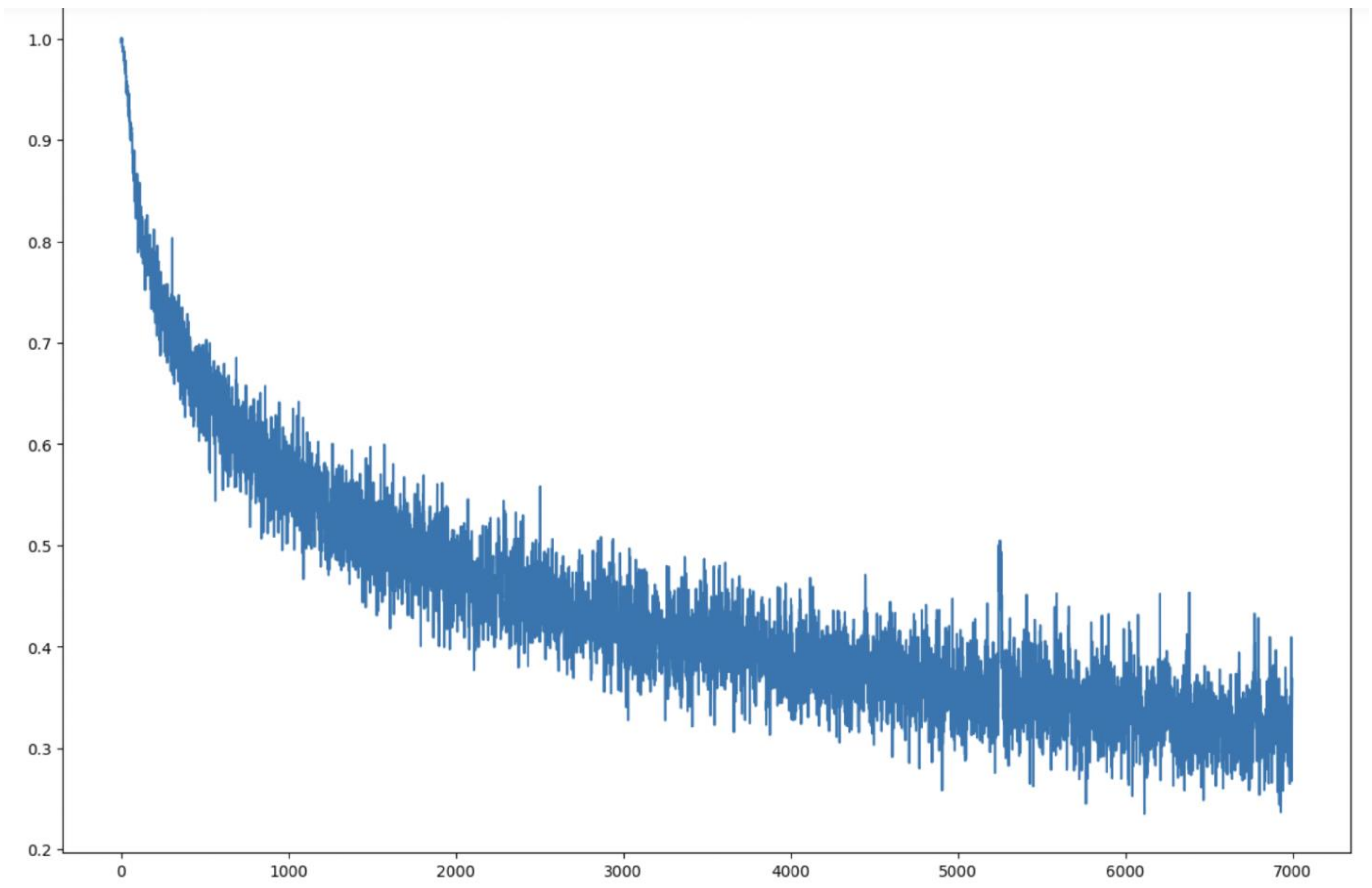
Implementation of Denoise Diffusion as per paper:

The implementation consists of a Denoise Diffusion class that has the following methods:

- **__init__(self, eps_model: nn.Module, n_steps: int, device: torch.device)**: Initializes the class with an epsilon model, the number of diffusion steps, and the device to use for computation. It computes the alpha and beta values based on the number of diffusion steps and computes the sigma2 and sigma$_t$ values.
- **q_xt_x0(self, x0: torch.Tensor, t: torch.Tensor) -> Tuple [torch.Tensor, torch.Tensor]**: Calculates the mean and variance of the distribution at time t for a given initial value x0.
- **q_sample(self, x0: torch.Tensor, t: torch.Tensor, eps: Optional[torch.Tensor] = None)**: Samples from the distribution at time t for a given initial value x0 and noise eps.
- **p_sample(self, xt: torch.Tensor, t: torch.Tensor)**: Samples from the prior distribution at time t for a given noise value xt.
- **loss (self, x0: torch.Tensor, noise: Optional[torch.Tensor] = None)**: Computes the loss between the original and the synthesized image using the diffusion model.

The **train** function trains the model using the given training data, optimizer, and number of epochs. It iterates through the dataset, computes the loss, computes the gradients, and updates the weights. The losses are tracked and plotted at the end of the training process.

Loss curve per epoch:



Note (gifs are in data folder)