Assignment 5

Generative Models

CMPUT 328 - Fall 2024

1 Assignment Description

The main objective in this assignment is to implement and evaluate some of the most popular generative models, including **Variational Auto-Encoders (VAE)**, **Denoising Diffusion Probabilistic Models (DDPM)**, **Denoising Diffusion Implicit Models (DDIM)**, and **Latent Diffusion Models (LDDPM)**. Our goal is to implement each of these models on the FashionMNIST dataset and see how such models can generate new images. However, instead of simply training the models on the whole dataset, we would like to be able to tell the model from which class it should generate samples. Hence, we are going to implement *class-conditional* generative models.

In this assignment the FashionMnist dataset will be padded by 2 in each side so that the image dimensions are 32×32, in order to have nicer behavior during down samplings (divisions by 2).

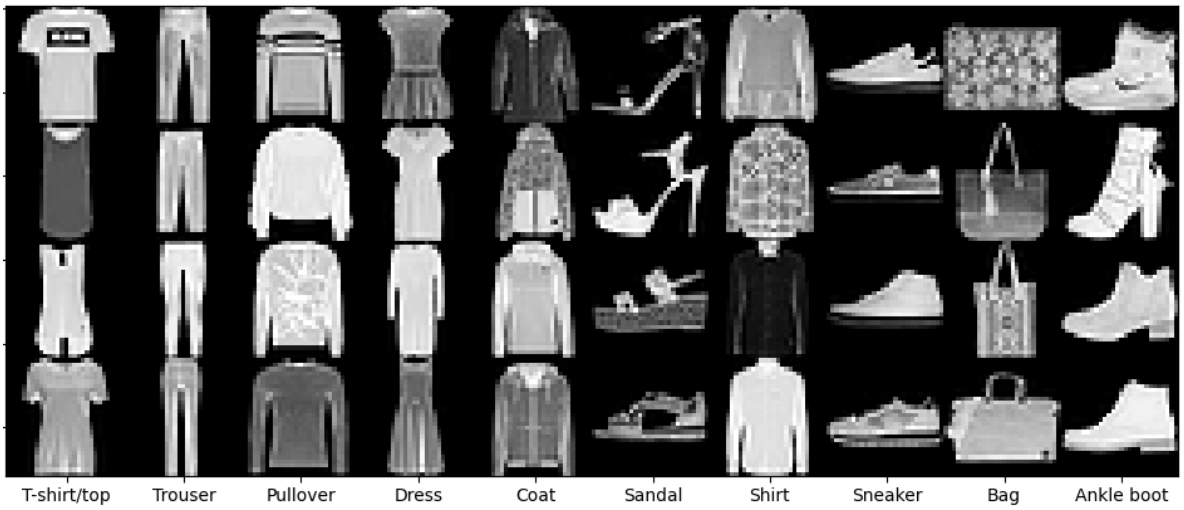

Figure 1: Sample images from the FashionMNIST dataset

2 What You Need to Do

For this assignment, the following files are given to you:

- models.py

- assignment.py
- notebook.ipynb
- trainers.py
- main.py
- classifier.py
- classifier.pt

However, all you need to concern about are **models.py** and **assignment.py**. Do not apply any changes to any other .py files. **notebook.ipynb** file is a notebook which you can use to train and evaluate different models.

During the training, a **checkpoints** directory will be created inside your project directory, which will contain the saved weights of your models. You will need to submit this directory as part of your assignment.

NOTE: In models.py some parts are already implemented. Do not change those parts or namings since it will lead to conflicts in other parts of the code base.

NOTE: In assignment.py, functions are filled in to demonstrate an example of how this functions should be filled. You can change the content of the functions (not the return value) according to your own model definitions.

NOTE: Inside the **main.py** file, lines 76-80 defines some general training configs including number of epochs, batch size, learning rate, the evaluation interval during training, and the number of samples to be generated in each evaluation interval. Feel free to change these configs for each part as you need.

You must submit a zip file containing the following: models.py, assignment.py, checkpoints.

Make sure you are uploading the correct weight files. If the weight files cannot be loaded into the models, the mark for the corresponding section will be deducted.

Make sure your submission size does not exceed the allowed size on eClass.

2.1 Task 1: Conditional DDPM (40%)

Inside the models.py file there are four classes that you need to complete for the DDPM:

1. **SinusoidalPositionEmbeddings**: This class receives the embedding dimension in its **__init__** and in the **forward** method receives a time step, and returns its sinusoidal positional encoding.

2. **UNet**: This class implements a UNet which is used in diffusion models to estimate the noise level. You are free to implement the UNet architecture as you please.
   NOTE: the **forward** method of the UNet must receive three components: 1- x, which is the noisy image, 2- time, which is the time step at which the noisy image is obtained, and 3- labels, which are the true labels of the input images.
   NOTE: The forward method should only return the estimated noise level.

3. **VarianceScheduler**: This class receives four values in its **__init__** method: 1- beta_start (β_1) (usually 0.0001), 2- beta_end (β_2) (usually 0.02), 3- num_steps (usually 1000), which determines the number of steps for diffusion forward and backward processes, and 4- interpolation, which specifies how beta values should be spread out in the interval (β_1, β_2).

   - interpolation argument should be either 'linear' or 'quadratic' for which you need to implement its corresponding interpolation strategy:
     1. linear interpolation from β_1 to β_2,
     2. quadratic interpolation (linearly from $\sqrt{\beta_1}$ to $\sqrt{\beta_2}$ and raise to the power of 2).
   - you might need to store other statistics here as well such as αs, $\bar{\alpha}$s, etc, inside **__init__**.
   - Implement **add_noise** method which receives the image $\mathbf{x}$ and the time step $\mathbf{t}$, and returns 1- the noisy image, and 2- the noise injected to the image.

4. **DDPM**:

   - **__init__**: Receives an instance of UNet and an instance of VarianceScheduler. This part is already done in models.py. Do not change this part.
   - **forward**: You need to complete this method. This method receives the input image and the labels, 1-randomly samples time steps, 2-adds noise to the image based on the variance scheduler, 3- estimates the noise using the UNet, 4- computes the loss (L1 or L2), and 5- returns the loss (take a look at the comments inside the models.py file.)
   - **recover_sample**: You need to complete this part. This method receives the noisy sample (x_t), the estimated noise, and the time step at which the noisy sample is obtained, and returns the sample at the previous step (x_{t-1}) based on the sample recovery strategy of DDPM.
   - **generate_sample**: You need to complete this method. This method receives the number of samples to be generated, the computational device (default is cuda), the labels for which the samples should be generated (could be None). The method then iteratively generates a sample from pure noise.

NOTE: In the generation phase DO NOT generate each sample individually. i.e., do not do a for loop over the number of samples. All samples must be generated in one batch.

NOTE: You can check the functionality of your variance scheduler using the first block of the **notebook.ipynb**.

NOTE: Once you are done with your implementation, you need to initialize your model inside the **prepare_ddpm()** function which returns your instance of the DDPM class. Then you can run the following command to train your model (the second block in the **notebook.ipynb**):

```
python main.py --model ddpm --mode train
```

Once the training is finished. You can generate images using your model by running the following command (the third block in **notebook.ipynb**):

```
python main.py --model ddpm --mode generate
```

2.2 Task 2: Conditional DDIM (30%)

Once you are done with the DDPM part, you only need complete the DDIM class for this part. The VarianceScheduler and the UNet from the previous part could be reused for this part.

For the DDIM class, everything is the same as DDPM class, except for the **recover_sample** method which receives the same arguments as that of DDPM, but runs the DDIM strategy to recover the previous sample.

NOTE: Once you are done with your implementation, you need to initialize your model inside the **prepare_ddim()** function which returns your instance of the DDIM class. Then you can run the following command to train your DDIM model (the fourth block in **notebook.ipynb**):

```
python main.py --model ddim --mode train
```

Once the training is finished. You can generate images using your model by running the following command (the fifth block in **notebook.ipynb**):

```
python main.py --model ddim --mode generate
```

2.3 Task 3: Conditional VAE (30%)

Here you are going to implement a Conditional Variational Autoencoder, which is going to be used in the next part as well. For this part you are supposed to complete the **VAE** class inside models.py.

NOTE: If you are going to implement the next part as well, your VAE must have at most 3 down sampling layers (i.e., the size of image at the bottleneck should be at least 4×4). Otherwise, you are allowed to have more downsampling layers.

You need to complete the following parts inside the VAE class:

- **__init__** Define the architecture of you network here including the μ-net and logvar-net, according to the comments.
- **forward** The forward method should receive the image, and its label, 1-encode the image, 2-sample from the latent distribution, 3-mix the label information, and 4- decode the sample to generate the image.
- **reparameterize**: You need to complete this part. This method implements the reparameterization trick for the VAE, and return the reparameterized sample.
- **reconstruction_loss**: This part is completed for you, by defining the similarity loss between the reconstructed image and the ground truth image in the form of binary cross entropy loss.

- **kl_loss**: This part is completed for you, by implementing the Kullback-Leibler divergence between the estimated μ and log-variance of your model and those of the standard Normal distribution as follows:

$$KL(\mathcal{N}(\mu, \sigma^2) \| \mathcal{N}(0, I)) = \frac{1}{2} \sum_{i=1}^{n} (\sigma_i^2 + \mu_i^2 - 1 - \ln(\sigma_i^2))$$

- **generate_sample**: You need to complete this part. This method receives the number of samples to be generated, the computational device (default is cuda), the labels from which the samples should be generated.

- **decode**: You must complete this part. This should encapsulate the decoder part of your network, which receives the noisy sample from the latent distribution and their corresponding labels and decodes the samples.

NOTE: Once you are done with your implementation, you need to initialize your model inside the **prepare_vae()** function which returns your instance of the VAE class. Then you can run the following command to train your VAE model (sixth block of **notebook.ipynb**):

```
python main.py --model vae --mode train
```

Once the training is finished. You can generate images using your model by running the following command (seventh block of the **notebook.ipynb**):

```
python main.py --model vae --mode generate
```

2.4 (Bonus) Task 4: Latent Diffusion Model (30%)

Once you have a working VAE and DDPM, implementing this part should be trivial. LDDPM, applies the diffusion model on the latent space of the VAE. For this part you need to complete the **LDDPM** class of the models.py file.

NOTE: Your UNet for the LDDPM class must have at most 2 down sampling layers. Since, at the bottleneck of the VAE, the image size reduces to 4×4, UNet further reduces this to 1×1. (If your vae reduces the image size to 8×8 you can have at most 3 down samplings in the UNet)

NOTE: The input channels of the UNet of the LDDPM is the same as the latent dimension of the VAE.

There are two differences between the LDDPM class and the DDPM class:

1. **__init__** method receives the VAE as well.

2. **generate_sample** method generates a sample in the latent space of the VAE, and hence the VAE's decoder must be employed to decode the sample from the latent space into a sample in the pixel space.

NOTE: Once you are done with your implementation, you need to initialize your model inside the **prepare_lddpm()** function which returns your instance of the LDDPM class. Then you can run the following command to train your LDDPM model (eighth block of **notebook.ipynb**):

```
python main.py --model lddpm --mode train
```

Once the training is finished. You can generate images using your model by running the following command (last block of **notebook.ipynb**):

```
python main.py --model lddpm --mode generate
```

2.5 Grading

:

1. After evaluation of each method aside from the generated samples which are saved into your project directory, am accuracy is also printed, which is the accuracy of a classifier provided to you when fed with the generated images by your model.
   For each part, an accuracy of 75% and lower will get no mark, and an accuracy of the 85% and above will get full mark.

2. For each part, 80% of the mark is designated to the accuracy you get and the other 20% is designated for the soundness of your codes.

3. We will evaluate you model 5 times, and consider the maximum score obtained as your final score for each part.