DLCV hw2

ID: r12922075

Name: 吳韋論

DDPM

1. Model architecture and implementation details

I follow this DDPM article and github Conditional UNet to implement DDPM.

Conditional Unet: (Same as github)

- Encoder → Downsampling with self-attention layer
- Decoder → Upsampling with self-attention layer
- Embedding → Embed label
- Position Encoding: sin and cos functions of varying frequencies

Training: Add noise + Predict noises with sample timesteps

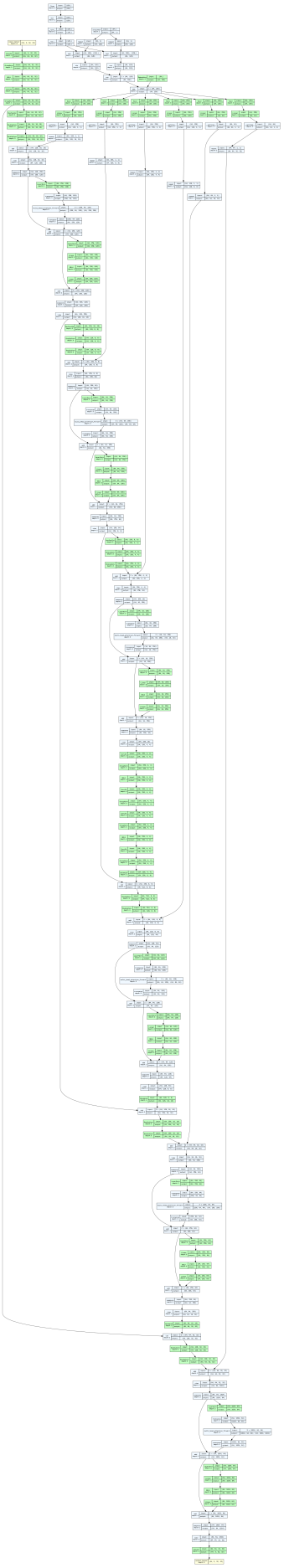
For each batch of images and labels, specifying the timesteps through 'sample_timesteps'. Then, it apply noise to the images based on the given 't'. Finally, it use the U-Net to predict how much noise was originally added, and calculate the loss accordingly.

Sampling: Sampling code (Same as github)

Starting from timestep T with randomly noise, denoising is performed, continuing to iterate down to 1, and the final result x_0 represents the denoised image.

Settings:

- Optimizer: AdamW
- Learning rate: 5e-5
- Epochs: 1000
- Loss: MSELoss
- Batchsize: 64

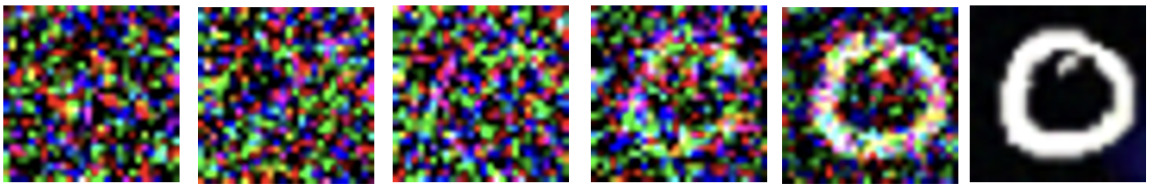


2. 10 generated images for each digit (0-9)

It can be seen that almost all classes can be clearly distinguished, and the colors also exhibit diversity, is similar to the training data of MNIST-M.

3. Reverse process of the first “0” in different time steps

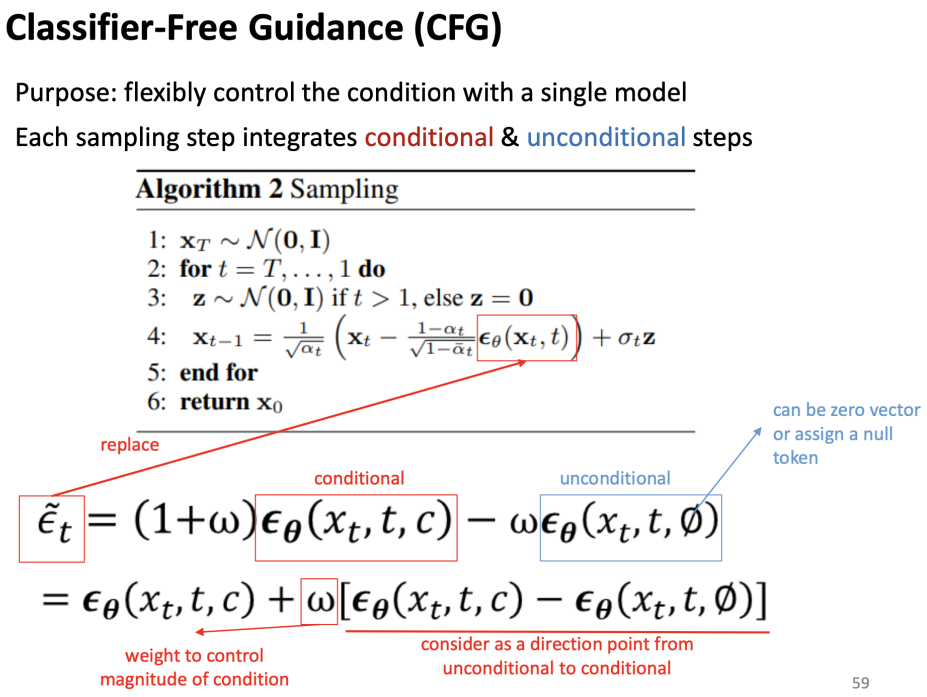
It can be observed that the gradual transition of simple noises to '0' over timesteps signifies that the denoising process for each timestep is meaningful and gradually eliminates the noise.



4. Observation and learn from DDPM

Observation:

Inference time is quite time-consuming. With 1000 images, it takes 20 minutes to run initially. It is discovered that CFG (Classifier-Free Guidance) leads to each image predicted twice, and after skipping that step, it reduces to 10 minutes. However, it still takes a significant amount of time.



With CFG enabled, the images become significantly clearer, allowing the accuracy to reach 99.5%. However, due to time constraints, CFG has to be turned off, resulting in an accuracy of around 96%.

Learn:

Most of the DDPM code is readily available. However, it's essential to understand the physical meaning of each component to make modifications and adapt them to the specific problem.

In this case, an attempt to reduce inference time is a challenge. Besides turning off CFG, I also enable 'cuda.amp.autocast()' to speed up the inference time.

DDIM

DDIM architecture is same as DDPM, only change sampling formulas: DDPM architecture

Sampling formulas → DDIM generalized_steps (Same as github)

1. Face images of noise 00.pt ~ 03.pt with different eta & Explain and report observation in this experiment

Explain & Report observation

- According to DDIM's sampling formula, when eta is set to 0 (the first row), the random noise 't' becomes 0, resulting in denoised images that are the same as the original images.
- When eta is set to 0.25 (the second row), randomness increases, and I observe that the denoised images roughly maintain the same contours as the original images. However, there are slight changes in details, such as the shape of the face and chin.
- From eta=0.5 to 1.0 (the third to fifth rows), randomness accounts for more than half of the changes. After running 1000 steps, the denoised images differ significantly from the original images

In order to check my guess, I perform images of noise 00.pt ~ 03.pt several times and observe that when eta is ≥ 0.5, the diversity of denoised images increases. This effect is particularly pronounced with eta = 1.0, which corresponds to DDPM.

$$\boldsymbol{x}_{t-1} = \sqrt{\alpha_{t-1}} \underbrace{\left(\frac{\boldsymbol{x}_t - \sqrt{1-\alpha_t}\epsilon_\theta^{(t)}(\boldsymbol{x}_t)}{\sqrt{\alpha_t}} \right)}_{\text{“ predicted } \boldsymbol{x}_0\text{”}} + \underbrace{\sqrt{1-\alpha_{t-1}-\sigma_t^2} \cdot \epsilon_\theta^{(t)}(\boldsymbol{x}_t)}_{\text{“direction pointing to } \boldsymbol{x}_t\text{”}} + \underbrace{\sigma_t \epsilon_t}_{\text{random noise}}$$

$$\sigma_{\tau_i}(\eta) = \eta \sqrt{(1-\alpha_{\tau_{i-1}})/(1-\alpha_{\tau_i})} \sqrt{1-\alpha_{\tau_i}/\alpha_{\tau_{i-1}}},$$

2. Face images of the interpolation of noise 00.pt ~01.pt & Explain and report observation in this experiment

Report observation:

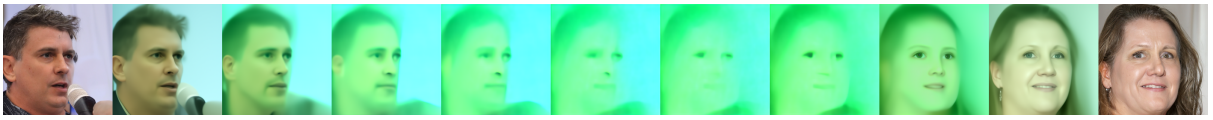
From the images, I observe that spherical linear interpolation (Slerp) applied to noise 00.pt to 01.pt smoothly transforms the facial features, transitioning from a male appearance to a neutral one, and finally to a female appearance.

However, with Simple Linear Interpolation applied to 'noise 00.pt' to '01.pt,' the intermediate images appear blurry, displaying only basic facial and eye contours and appearing in green.

Explain observation:

Slerp's key idea is to find a path on the sphere's surface connecting two points, like two 3D orientations. Unlike linear interpolation's shortest path in Euclidean space, Slerp always takes the shortest path on the sphere's curved surface.

Spherical Linear Interpolation



Simple Linear Interpolation

DANN

1. Conduct the experiments

According to the table, adaptation falls between the lower and upper bounds, and USPS showing better accuracy, reaching 80.44% at the lower bound as well as 93.28% at the DANN.

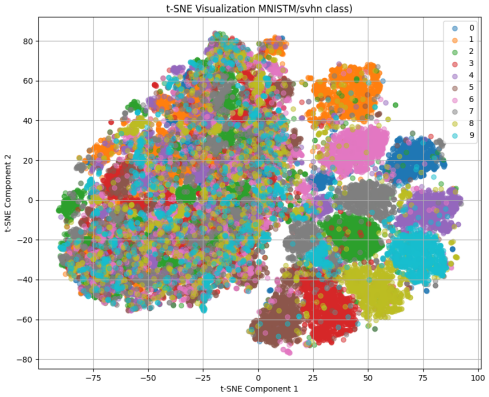
	MNIST-M → SVHN	MNIST-M → USPS
Trained on source	40.03% (6359/15887)	80.44% (1197/1488)
Adaptation (DANN)	52.51% (8343/15887)	93.28% (1388/1488)
Trained on target	93.64% (14877/15887)	98.86% (1471/1488)

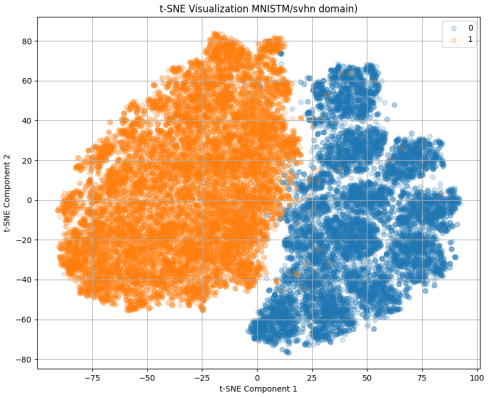
2. Visualize the latent space of DANN by mapping the *validation* images to 2D space with t-SNE by digit class and by domain

Based on the images below, it can observe that SVHN fails to achieve a strong domain-invariant feature in the latent space. However, it is evident that there is a trend of class separation.

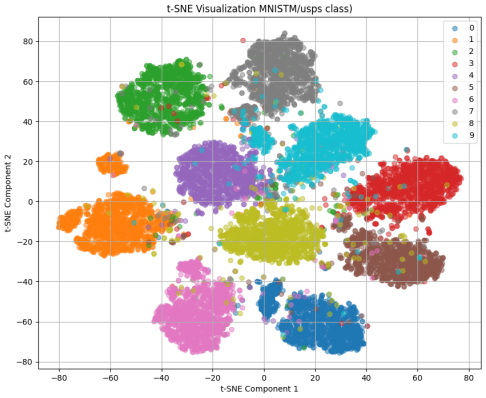
In contrast, it is clear that the USPS latent space has domain-invariant features, and the domains are mixed together. This means that the domain classifier indeed struggles to distinguish between the target and source domains.

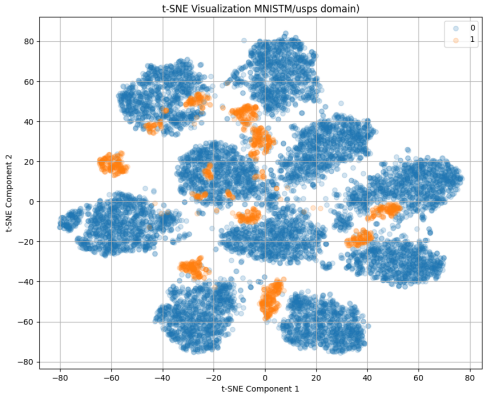
MNIST-M → SVHN (Class, Domain)





MNIST-M → USPS (Class, Domain)





3. Implementation details of model, observed and learned from DANN

Model architecture → DANN architecture, Model layer → Layers, Loss → label Loss&domain Loss

Alpha → Alpha weight (Same as github)

Model architecture & Layer:

- Encapsulate the feature extractor, label classifier, and domain classifier into a single class, named “DANNModel”.
- Change some layers in the original feature extractor, label classifier, and domain classifier. The reason for this is to increase the complexity of the model by adding layers.
- In the backward pass, using `grad_output.neg()` allows the domain classifier to propagate loss in a negative direction.

Model Training:

- itertools.cycle to handle source and target dataloader with different sizes.
- Use Alpha to increase domain classifier negative gradient weight from 0 to 1.
- Label loss → CrossEntropyLoss, Domain loss → BCEWithLogitsLoss

Observation:

- Adaption can maximize domain loss and minimize label loss, which achieves domain-invariant feature

- Not all target datasets result in a significant increase in domain loss. This means that the domain classifier can still differentiate between the target and source.

Learn: There are some details, like handling alpha values and changing the sign of gradients. After implementing it, I print the accuracy of both the label classifier and domain classifier for each epoch. As expected, the label classifier's accuracy improved over time, while the domain classifier's accuracy drop to around 50-60%, showing it is domain-invariant feature.