DLCV HW2 Report

NTUEE
B10901187
Hung Kai Chung

Oct. 2023

Problem1: DDPM

1. Draw model architecture & explain the implementation details

In problem1, I use the classifier-free-guidance + DDPM to implement conditional DDPM.
Classifier free guidance

$$\nabla_x \log p(x|y) = \nabla_x \log \frac{p(y)p(x|y)}{p(x)} (\nabla_x \log p(y) = 0) \tag{1}$$
$$= \nabla \log p(x) + \nabla \log p(y|x) \tag{2}$$

$$\nabla \log p(y|x) = \nabla \log p(x|y) - \nabla \log p(x) \tag{3}$$

Reformulate the formula we get 3 and replace the $\nabla \log p(y|x)$

$$\nabla \log p(x|y) = \nabla \log p(x) + \gamma \nabla \log p(y|x) \tag{4}$$
$$= \nabla \log p(x) + \gamma(\nabla \log p(x|y) - \nabla \log p(x)) \tag{5}$$
$$= (1-\gamma)\nabla \log p(x) + \gamma \log p(x|y) \tag{6}$$

Finally, we can get the classifier free guidance formula, I use a Unet(adapted from this website) to be my denoised model, when training I will give label and no label(using mask to block the label) paired data. The **training settings** is below.

- Transform = ToTensor()

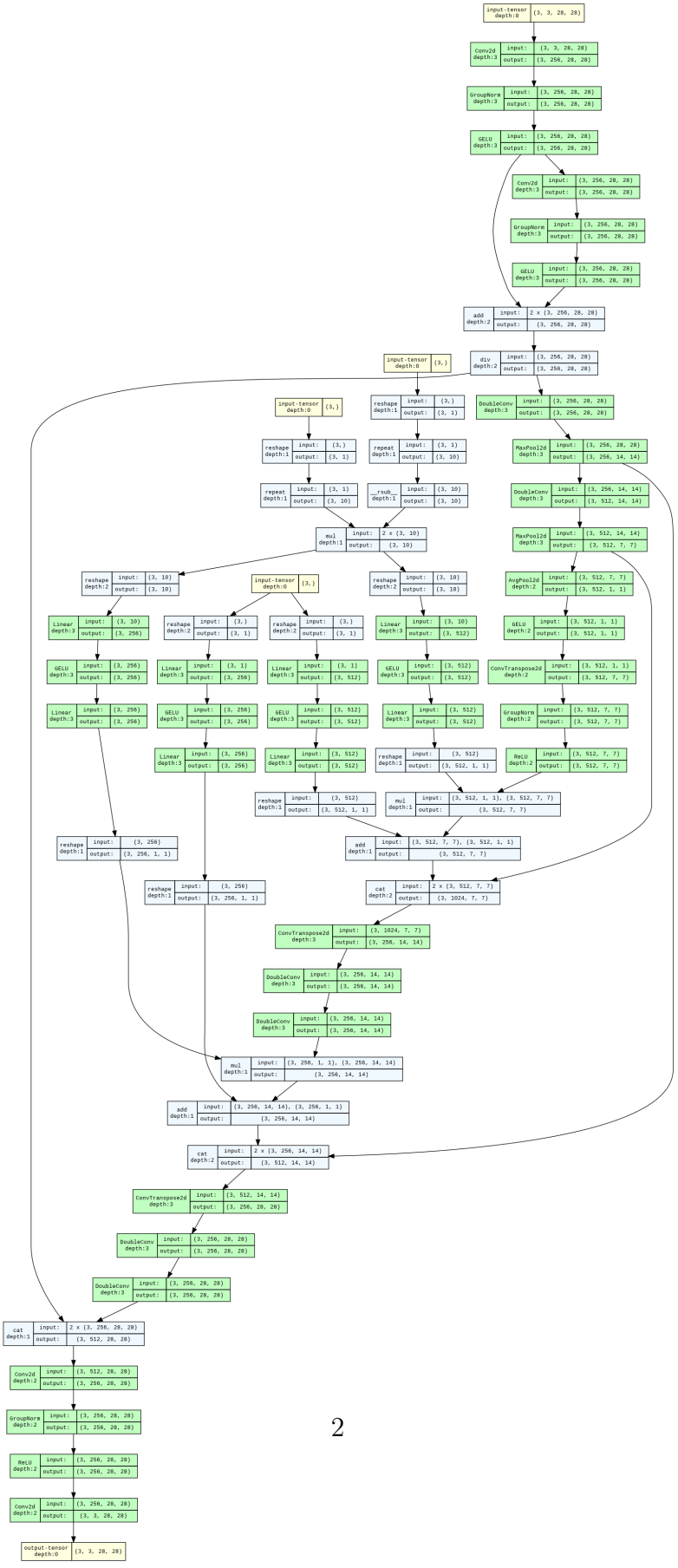


Figure 1: DDPM model architecture

- Batch size = 256
- Number of epoch = 75
- Time steps = 400
- lr = 1e-4
- Noise scheduler = cosine scheduler(due to linear scheduler causes the img to fast to be noise.)
- Classifier free guidance weight = 2.0
- Loss function = nn.MSELoss
- Optimizer = Adam

Another differnce between DDPM Unet Model and normal Unet Model is that the DDPM Unet Model will add **Time & Label Embedding**, I add embedding in up-conv part in Figure 2

```
x4 = self.up[1](label_emb1 * x3 + time_emb1, x2)
x5 = self.up[2](label_emb2 * x4 + time_emb2, x1)
```

Figure 2: Time & Label Embedding

2. Please show 10 generated images for each digit (0-9) in your report.

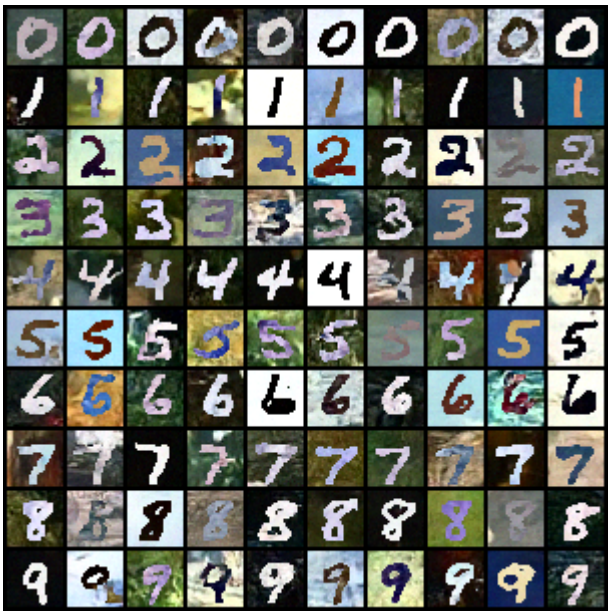


Figure 3: 100 generated images with each digit (0-9)

3. Visualize total six images in the reverse process of the first "0" in your grid in Figure 3 with different time steps.

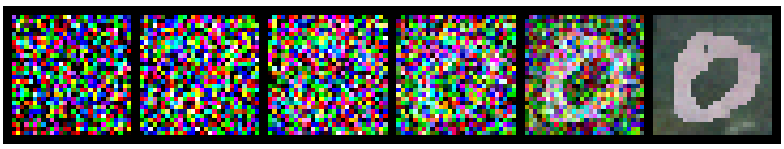


Figure 4: six images in the reverse process of the first "0" in Figure 3

4. Discuss what you've observed and learned from implementing conditional diffusion model

In this homework, I read the Understanding Diffusion Models: A Unified Perspective and Classifier-Free Diffusion Guidance. In this two paper, they used **Score-based Generative Models** to illustrate the diffusion model. In the denoised stage, the ddpm model will do is taking the gradient of its

log likelihood, which is the direction of increase its likelihood. The most important part of DDPM is that we want to generate different img, so given the same initialization point, we also want to get different img(output diversity), this has done by adding the stochastic noise term in the Langevin dynamics sampling procedure. In classifier free guidance, it used one diffusion model and without a classifier by using two different kinds of data (with label & without label), I implement it by using a mask to mask the y(label). By using the γ in 6 to intepolate the unconditional & conditional Denoised Model, I can control the gradient direction so that the img will more depend on the conditional Denoise Model direction.

```
mask = mask.reshape(-1, 1).repeat(1, self.n_cls)
mask = (1 - mask)     # due to the 1 is I want to mask so we have to flip 0->1 1->0
y = y * mask
```

Figure 5: mask label

Problem2: DDIM

1.Please generate face images of noise 00.pt - 03.pt with different eta in one grid. Report and explain your observation in this experiment.

Formula 7 is the DDIM sampling process, the η is a parameter that interpolates between the deterministic DDIM and the stochastic DDPM. As depicted in Figure 6, the resulting images demonstrate an increased level of diversity, when the η is gradually increasing. The η is small like 0.0, 0.2, this four images are still deterministic, but when the η is 0.4 to 1.0, it becomes another person and don't get any same feature in image of $\eta = 0.0$.

Figure 6: η = [0.0, 0.2, 0.4, 0.6, 0.8, 1.0]

$$x_{\tau-1} = \sqrt{\overline{\alpha_{\tau_{i-1}}}}\left(\frac{x_{\tau_i} - \sqrt{1-\overline{\alpha_{\tau_i}}}\epsilon_\theta^{(\tau_i)}(x_{\tau_i})}{\sqrt{\overline{\alpha_{\tau_i}}}}\right) + \sqrt{1-\overline{\alpha_{\tau_{i-1}}} - \sigma_{\tau_i}(1)^2} \cdot \epsilon_\theta^{(\tau_i)}(x_{\tau_i}) + \widehat{\sigma}_{\tau_i}\epsilon \tag{7}$$

$$\sigma_{\tau_i}(\eta) = \eta\sqrt{(1-\overline{\alpha_{\eta_{i-1}}})/(1-\overline{\alpha_{\eta_i}})}\sqrt{1-\overline{\alpha_{\tau_i}}/\overline{\alpha_{\eta_{i-1}}}} \tag{8}$$

2. Please generate the face images of the interpolation of noise 00.pt - 01.pt. What will happen if we simply use linear interpolation? Explain and report your observation.

In Sampling Generative Networks, the generative model are sampled from a set of latent space, so using the linear interpolation would be much more error, due to these can result in sampling the latent space from locations very far outside the manifold of probable locations. As illustrated by Figure 7, linearly interpolating between two latent vectors will usually result the magnitude of the vector decreases. In contrast, slerp treats the interpolation as a great circle path on an n-dimensional hypersphere, promising results on diffusion models with uniform and gaussian priors. The images of spherical linear interpolation & linear interpolation show in Figure 8 & 9

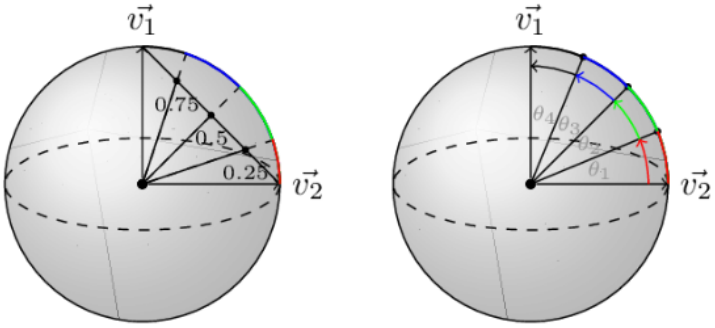


Figure 7: Slerp & Linear interpolation path

Figure 8: spherical linear interpolation

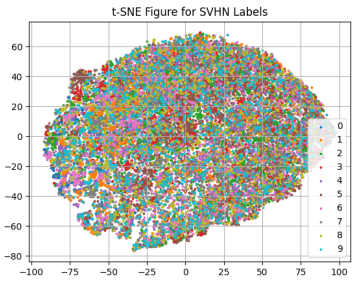
Figure 9: linear interpolation

Problem3: DANN

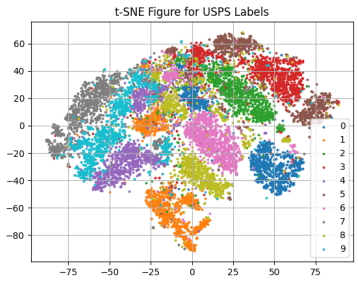
1. Please create and fill the table with the following format in your report

	MNIST-M → SVHN	MNIST-M → USPS
Trained on source	0.28353	0.6708
Adaptation (DANN)	0.48817	0.8692
Trained on target	0.91405	0.9558

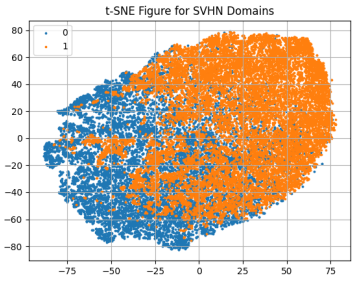
2. Please visualize the latent space (output of CNN layers) of DANN by mapping the validation images to 2D space with t-SNE.



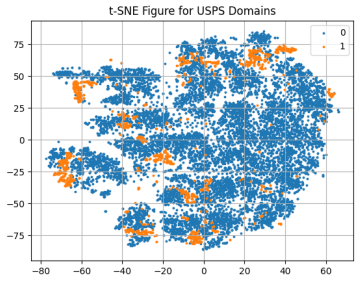
(a) SVHN Labels t-SNE



(b) USPS Labels t-SNE



(c) SVHN Domains t-SNE



(d) USPS Domains t-SNE

Figure 10: DANN validation t-SNE

3. Please describe the implementation details of your model and discuss what you've observed and learned from implementing DANN.

Domain-Adversarial Training of Neural Networks(DANN) represents a method employed to address the transfer of a source domain to different domain task. The fundamental tenets of this approach encompass (i) discriminativeness and (ii) domain-invariance. In the former, the assertion is that the features yielded by the Feature Extractor excel in classification tasks, while the latter posits that the Domain Classifier is incapable of discerning between source and target domain data. The DANN model consists of Feature Extractor, Label Predictor, Domain Classifier and a Gradient Reversal Layer(GRL). In Problem 3, we are given a labeld source domain - **mnistm** dataset and two unlabeled target domain - **usps** & **svhn** dataset, the first thing I notice is that the usps dataset is have one channel(only white & black), so I just copy this channel to three channel(rgb). In Figure 11, I adopted the architecture of this model from the DANN paper. It is noteworthy that the model architectures for SVHN and USPS are identical.

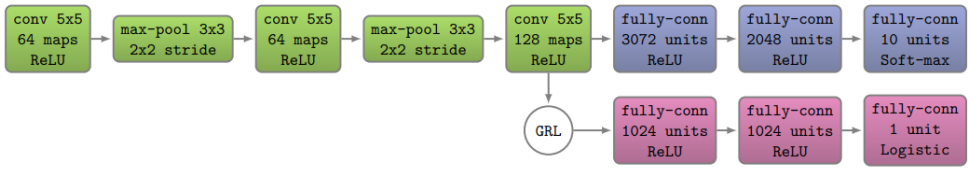


Figure 11: SVHN & USPS model architecture

The **training settings** are below.

- Transform = [ToTensor(), Normalize()]
- Batch size = 128
- Number of epoch = 100
- lr = 0.002 for SVHN 0.001 for USPS
- Loss function = (i) nn.CrossEntropyLoss() for Label Predictor (ii) nn.BCELoss() for Domain Classifier
- Optimizer = SGD
- scheduler = ConstantLR(factor=0.7, totaliters=10)

The **training process** is below, but first we have to write down the Loss.

$$L = L_{CE} + \lambda L_{BCE} \tag{9}$$

$$E(\theta_f, \theta_y, \theta_d) = \frac{1}{n}\sum_{i=1}^{n} L_y^i(\theta_f, \theta_y) - \lambda\left(\frac{1}{n}\sum_{i=1}^{n} L_d^i(\theta_f, \theta_d) + \frac{1}{n'}\sum_{i=n+1}^{N} L_d^i(\theta_f, \theta_d)\right) \tag{10}$$

Using pytorch SGD, the parameter will gradient descent like formula 11 to 14. Notice that in formula 13, a minus notation exists in chain rule of backpropagation, this is due to θ_d is argmax the E_{BCE}(we want the domain classifier can't classify well on source & target data), so θ_d will do gradient ascent. However, we use the SGD, which is argmin, gradient descent the parameter. Hence, we add a **Gradient Reversal Layer** (GRL) between the Feature Extractor & Domain Classifier, the implementation is in Figure 12

$$\theta_y \longleftarrow \theta_y - \mu\left(\frac{\partial L_y^i}{\partial \theta_f} - \lambda\frac{\partial L_d^i}{\partial \theta_f}\right) \tag{11}$$

$$\theta_d \longleftarrow \theta_d - \mu\lambda\frac{\partial L_d^i}{\partial \theta_d} \tag{12}$$

$$\theta_f \longleftarrow \theta_f - \mu\left(\left(\frac{\partial L_y^i}{\partial \theta_y}\right)\left(\frac{\partial \theta_y}{\partial \theta_f}\right) + \left(\lambda\frac{\partial L_d^i}{\partial \theta_d}\right)\left(-\frac{\partial \theta_d}{\partial \theta_f}\right)\right) \tag{13}$$

$$\longleftarrow \theta_f - \mu\left(\frac{\partial L_y^i}{\partial \theta_f} - \lambda\frac{\partial L_d^i}{\partial \theta_f}\right) \tag{14}$$

```python
class GradientReversalLayerFunction(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, lambda_):
        ctx.lambda_ = lambda_
        return x.view_as(x)

    @staticmethod
    def backward(ctx, grad_output):
        output = grad_output.neg() * ctx.lambda_
        return output, None
```

Figure 12: Gradient Reversal Layer