HW1: Autoencoder

Written by — 312581020 許瀚丰

Q1: Model

- Describe your Autoencoder model, including details about the Encoder and Decoder architecture, the activation functions used, any regularization techniques applied, and other relevant components.

AE

```
self.encoder = nn.Sequential(
    nn.Conv2d(3, 64, 4, 2, 1),   # 64 x 128 x 128
    nn.BatchNorm2d(64),
    nn.Mish(inplace=True),

    nn.Conv2d(64, 128, 4, 2, 1),  # 128 x 64 x 64
    nn.BatchNorm2d(128),
    nn.Mish(inplace=True),

    nn.Conv2d(128, 256, 4, 2, 1),  # 256 x 32 x 32
    nn.BatchNorm2d(256),
    nn.Mish(inplace=True),

    nn.Conv2d(256, 512, 4, 2, 1),  # 512 x 16 x 16
    nn.BatchNorm2d(512),
    nn.Mish(inplace=True),

    nn.Conv2d(512, 1024, 4, 2, 1),  # 1024 x 8 x 8
    nn.BatchNorm2d(1024),
    nn.Mish(inplace=True),
)

self.decoder = nn.Sequential(
    nn.ConvTranspose2d(1024 1024, 4, 2, 1),  # 1024 x 16 x 16
    nn.BatchNorm2d(1024),
    nn.Mish(inplace=True),

    nn.ConvTranspose2d(1024, 512, 4, 2, 1),  # 512 x 32 x 32
    nn.BatchNorm2d(512),
    nn.Mish(inplace=True),

    nn.ConvTranspose2d(512, 256, 4, 2, 1),  # 256 x 64 x 64
    nn.BatchNorm2d(256),
    nn.Mish(inplace=True),

    nn.ConvTranspose2d(256, 128, 4, 2, 1),  # 128 x 128 x 128
    nn.BatchNorm2d(128),
    nn.Mish(inplace=True),

    nn.ConvTranspose2d(128, 64, 4, 2, 1),   # 64 x 256 x 256
    nn.BatchNorm2d(64),
    nn.Mish(inplace=True),

    nn.Conv2d(64, 3, kernel_size=3, stride=1, padding=1),  # 3 x 256
x 256
    nn.Tanh()
)
```

VAE

```
# Encoder
self.encoder = nn.Sequential(
    nn.Conv2d(3, 64, 4, 2, 1),   # 64 x 128 x 128
    nn.BatchNorm2d(64),
    nn.Mish(inplace=True),

    nn.Conv2d(64, 128, 4, 2, 1),  # 128 x 64 x 64
    nn.BatchNorm2d(128),
    nn.Mish(inplace=True),

    nn.Conv2d(128, 256, 4, 2, 1),  # 256 x 32 x 32
    nn.BatchNorm2d(256),
    nn.Mish(inplace=True),

    nn.Conv2d(256, 512, 4, 2, 1),  # 512 x 16 x 16
    nn.BatchNorm2d(512),
    nn.Mish(inplace=True),

    nn.Conv2d(512, 1024, 4, 2, 1),  # 1024 x 8 x 8
    nn.BatchNorm2d(1024),
    nn.Mish(inplace=True),

    nn.Conv2d(1024, latent_dim * 2, kernel_size=3, stride=1,
padding=1) # (2 x latent_dim) x 8 x 8
)
# Decoder
self.decoder = nn.Sequential(
    nn.ConvTranspose2d(latent_dim, 1024, 4, 2, 1),  # 1024 x 16 x 16
    nn.BatchNorm2d(1024),
    nn.Mish(inplace=True),

    nn.ConvTranspose2d(1024, 512, 4, 2, 1),  # 512 x 32 x 32
    nn.BatchNorm2d(512),
    nn.Mish(inplace=True),

    nn.ConvTranspose2d(512, 256, 4, 2, 1),  # 256 x 64 x 64
    nn.BatchNorm2d(256),
    nn.Mish(inplace=True),

    nn.ConvTranspose2d(256, 128, 4, 2, 1),  # 128 x 128 x 128
    nn.BatchNorm2d(128),
    nn.Mish(inplace=True),

    nn.ConvTranspose2d(128, 64, 4, 2, 1),   # 64 x 256 x 256
    nn.BatchNorm2d(64),
    nn.Mish(inplace=True),

    nn.Conv2d(64, 3, kernel_size=3, stride=1, padding=1),  # 3 x 256
x 256
x 256
    nn.Tanh()
)
```

Initialization

```
def _initialize_weights(self):
    for m in self.modules():
        if isinstance(m, (nn.Conv2d, nn.ConvTranspose2d)):
            nn.init.kaiming_normal_(m.weight, nonlinearity='linear')
            if m.bias is not None:
                nn.init.zeros_(m.bias)
        elif isinstance(m, nn.BatchNorm2d):
            nn.init.ones_(m.weight)
            nn.init.zeros_(m.bias)
```

- First, I resize the input images to 3 × 256 × 256 and normalize the pixel values to the range (-1, 1).
- Next, the images are passed through five convolutional layers, transforming them into feature maps of size 1024 × 8 × 8. (In the VAE, an additional convolution layer is applied to facilitate the reparameterization trick.) The decoder mirrors the encoder, using five transposed convolutional layers to reconstruct the images back to 3 × 256 × 256. Finally, a Tanh activation function ensures that the output values are constrained within (-1, 1).
- During initialization, I apply Kaiming initialization to enhance convergence. Throughout the network, I utilize Mish as the activation function. Compared to ReLU, Mish provides smoother gradients and is differentiable everywhere. Its function is defined as $f(x) = x \cdot \tanh(\text{softplus}(x))$

Additionally, Batch Normalization is employed to stabilize the training process.

Q2: Training

- Explain how you trained the model, specifying the loss function, optimization method, learning rate, and any other training-related details.

AE

Foward

```python
def forward(self, x):
    x = self.encoder(x)
    x = self.decoder(x)
    return x
```

- The training process of an AE is very straightforward: the encoder compresses the image, and the decoder reconstructs it.

Training

```python
def loss_function(recon_x, x):
    recon_loss = F.mse_loss(recon_x, x)
    return recon_loss

def get_optimizer(model, lr=1e-4):
    return optim.Adam(model.parameters(), lr=lr, betas=(0.5, 0.999))

def get_scheduler(optimizer):
    return optim.lr_scheduler.ReduceLROnPlateau(optimizer,
mode='min', factor=0.5, patience=10)


model = AE().to(device)
optimizer = get_optimizer(model)
scheduler = get_scheduler(optimizer)

for epoch in range(1, num_epochs + 1):
    model.train()
    train_loss = 0
    progress = tqdm(dataloader, desc=f'Epoch {epoch}/{num_epochs}')

    for batch_idx, data in enumerate(progress):
        data = data.to(device)
        recon_batch = model(data)
        loss = loss_function(recon_batch, data)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        train_loss += loss.item()

    avg_loss = train_loss / len(dataloader)
    print(f'====> Epoch: {epoch} Average loss: {avg_loss:.4f}')

    scheduler.step(avg_loss)
```

- During the training of the AE, MSE is used to compare the original image with the reconstructed image. Our goal is to make them as similar as possible. Throughout the training process, I used Adam as the optimizer to update the network.

- In my experiments, the optimal hyperparameter settings I found are as follows:

Epoch	**Optimizer**	**LR**	**LR Decay Rate**	**LR Decay Epoch**
1000	Adam	1e-4	0.5	10

VAE

Foward

```python
# Forward
def encode(self, x):
    x = self.encoder(x)
    mu, logvar = torch.chunk(x, 2, dim=1)
    return mu, logvar

def reparameterize(self, mu, logvar):
    std = torch.exp(0.5 * logvar)
    eps = torch.randn_like(std)
    return mu + eps * std

def decode(self, z):
    x = self.decoder(z)
    return x

def forward(self, x):
    mu, logvar = self.encode(x)
    z = self.reparameterize(mu, logvar)
    x_recon = self.decode(z)
    return x_recon, mu, logvar
```

- During the training process of the VAE, in order to optimize both loss terms simultaneously, the reparameterization trick is employed. This involves using the mean (μ) and log variance ($\log \sigma^2$) generated by the encoder to sample from a Gaussian distribution, and then attempting to reconstruct the input through the decoder.

Training

```
# Training
def loss_function(recon_x, x, mu, logvar, kld_weight=0.000025):
    recon_loss = F.mse_loss(recon_x, x)
    kld_loss = -0.5 * torch.mean(1 + logvar - mu.pow(2) -
logvar.exp())
    return recon_loss + kld_weight * kld_loss, recon_loss, kld_loss

def get_optimizer(model, lr=1e-4):
    return optim.Adam(model.parameters(), lr=lr, betas=(0.5, 0.999))

def get_scheduler(optimizer):
    return optim.lr_scheduler.ReduceLROnPlateau(optimizer,
mode='min', factor=0.5, patience=10)

model = VAE(latent_dim=1024).to(device)
optimizer = get_optimizer(model)
scheduler = get_scheduler(optimizer)

for epoch in range(1, num_epochs + 1):
    model.train()
    train_loss = 0
    progress = tqdm(dataloader, desc=f'Epoch {epoch}/{num_epochs}')
    for batch_idx, data in enumerate(progress):
        data = data.to(device)
        recon_batch, mu, logvar = model(data)
        loss, recon_loss, kld_loss = loss_function(recon_batch, data,
mu, logvar, kld_weight=0.00025)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        train_loss += loss.item()

    avg_loss = train_loss / len(dataloader)
    print(f'====> Epoch: {epoch} Average loss: {avg_loss:.4f}')
    scheduler.step(avg_loss)
```

- The loss function of VAE consists of both MSE and KL Divergence, which are used simultaneously to optimize the model. These are expressed as $\mathcal{L}_{\text{recon}} = \frac{1}{n}\sum_{i=1}^{n} |\hat{x}_i - x_i|^2$ and $\mathcal{L}_{\text{KLD}} = -\frac{1}{2}\mathbb{E}\left[1 + \log(\sigma^2) - \mu^2 - \sigma^2\right]$.

- However, balancing MSE and KL Divergence is crucial for achieving good results. Relying too much on MSE may lead to samples that do not follow the distribution N(0, 1) , potentially making the latent space discontinuous and sparse. On the other hand, focusing too much on KL Divergence ensures that the output follows N(0, 1) , but this may degrade the quality of the generated images. Therefore, an additional hyperparameter β is introduced to control the weight of the KL Divergence term. As a result, the final loss function is given by $\mathcal{L}\text{VAE} = \mathcal{L}\text{recon} + \beta\mathcal{L}_{\text{KLD}}$, and Adam is used as the optimizer to update the model parameters.

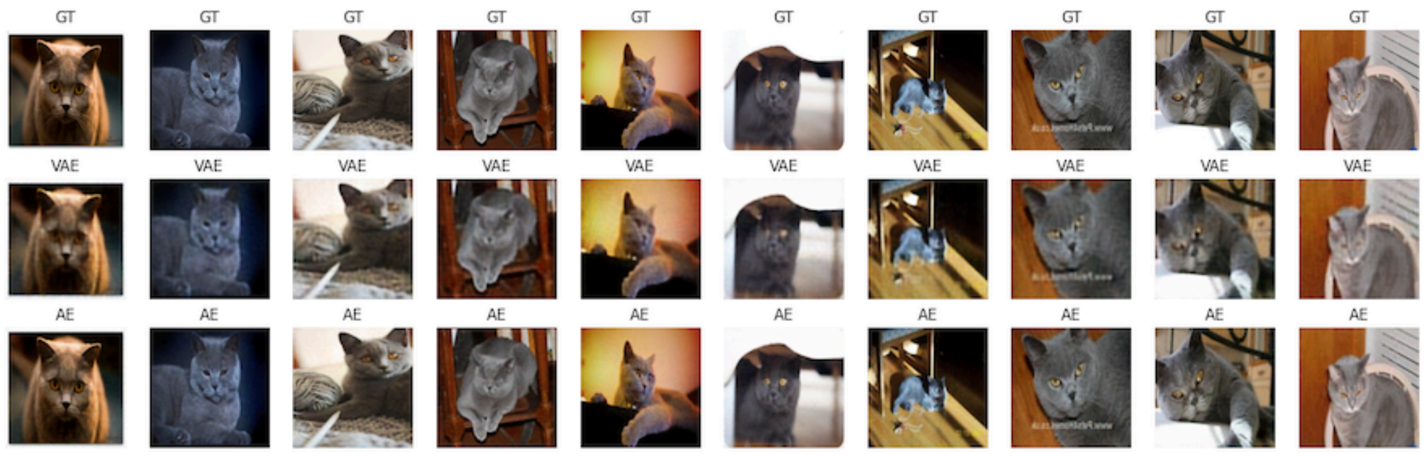
- In my experiments, the optimal hyperparameter settings I found are as follows:

Epoch	**Optimizer**	**LR**	**Beta**	**LR Decay Rate**	**LR Decay Epoch**
1000	Adam	1e-4	0.00025	0.5	10

Q3: Image Result

- Qualitative Analysis: Display the inference results for a random sample of 10 images, showing the Ground Truth, the predictions from the AE, and the predictions from the VAE.

Result



Q4: Evaluation Metrics

- Introduce the qualitative evaluation metrics: PSNR (Peak Signal-to-Noise Ratio), SSIM (Structural Similarity Index Measure), and LPIPS (Learned Perceptual Image Patch Similarity).

PSNR

- Peak Signal to Noise Ratio (PSNR) is used to evaluate the quality of compressed images. The closer the compressed image is to the original image, the higher the PSNR.
- Since MSE can vary based on the bit depth (for example, using pixel values in the range $0 \sim 255$ versus normalizing the image to $0 \sim 1$), the denominator uses the MSE. The numerator reflects the maximum possible pixel value squared to normalize the error relative to the bit depth, whole formula would be $\text{PSNR} = 10 \cdot \log_{10}\left(\frac{MAX^2}{\text{MSE}}\right)$

SSIM

- Structural Similarity Index (SSIM) is designed to solve the shortcomings of MSE (Mean Squared Error) by aligning with human subjective visual perception. MSE only measures the pixel-wise difference between two images, which may not reflect how similar they appear to human eyes. In contrast, SSIM focuses on comparing luminance, contrast, and structure to provide a more perceptually meaningful similarity score. The formula is $\text{SSIM}(x, y) = [I(x, y)]^\alpha \cdot [C(x, y)]^\beta \cdot [S(x, y)]^\gamma$

Where:

- I(x, y) is the luminance comparison between the two images.
- C(x, y) is the contrast comparison between the two images.
- S(x, y) is the structural comparison between the two images.
- α, β, γ are weights that control the relative importance of luminance, contrast, and structure, respectively.

The luminance, contrast, and structure components are defined as:

- Luminance comparison: $I(x, y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}$
    - Luminance measures the overall brightness of the image, represented by the mean pixel values. This comparison ensures that both images have consistent lighting conditions, minimizing discrepancies due to illumination differences.

- Contrast comparison: $C(x,y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2}$
  - Similar to Luminance, contrast reflects the difference between light and dark areas within the image, represented by the standard deviation of pixel values. This comparison ensures that the contrast in the reconstructed or compressed image remains consistent with the original.

- Structural comparison: $S(x,y) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3}$
  - Structure comparison evaluates the spatial relationship between pixels, focusing on patterns and edges in the image. This is captured through the covariance σ_{xy} between the two images, which reflects the similarity of their structure.

- A higher SSIM value, closer to 1, indicates a higher similarity between the two images, meaning that the compressed or reconstructed image retains most of the essential visual information from the original.

LPIPS

- LPIPS (Learned Perceptual Image Patch Similarity) measures perceptual similarity between two images by leveraging deep neural networks. It improves upon pixel-wise metrics by aligning with human visual perception. A lower LPIPS value indicates greater similarity in feature space and perceptual quality.

- The LPIPS metric measures the difference between two images x and x_0 using feature maps extracted from a pre-trained deep neural network, such as VGG or AlexNet. The formula is $d(x, x_0) = \sum_l \frac{1}{H_l W_l} \sum_{h,w} \| w_l \odot \left(\hat{y}^l_{hw} - \hat{y}^l_{0,hw} \right) \|_2^2$

Where:

- x and x_0 are the two images being compared.
- l denotes a layer in the pre-trained network.
- H_l and W_l are the height and width of the feature map at layer l.

- $\hat{y}_{hw}^{l}$ and $\hat{y}_{0,hw}^{l}$ are the normalized feature maps for images x and x_0 at layer l, at position (h, w) .
- w_l are learned weights for each layer, adjusting the importance of the feature map.

Q5: Evaluation Result

- Present the qualitative analysis results for both the AE and VAE models, including the PSNR, SSIM, and LPIPS values.

Result

Metrics	AE	VAE
PSNR	29.38	26.55
SSIM	0.78	0.69
LPIPS	0.23	0.35

Q6: Conclusion

- Conclusion: Based on the training and testing results of the AE and VAE models, provide a summary and final conclusion for this assignment.

Summary and Conclusion

- In this HW, I trained both Autoencoder and Variational Autoencoder. The experimental results show that AE outperforms VAE because AE only needs to learn a single mapping function, while VAE must model a distribution. As a result, AE cannot truly be considered a generative model since we cannot sample from its latent space to generate new images. In contrast, VAE focuses on learning the distribution of features, making its latent space more interpretable than AE's. With sufficient training data, VAE can generate images by sampling from latent noise.