Image & Video Processing: Assignment 5

Kyla Kaplan
June 2024

Contents

1 Image classification

1 Build the model

Note: I ran most of this code in Google Colab to allow for more epochs.

As you can find in the accompanying code in `exercise1.ipynb`, the definition of the model according to the assignment instructions are as follows:

```
class ClassificationModel(nn.Module):
    def __init__(self, num_classes):
        super(ClassificationModel, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2)
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.relu3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(kernel_size=2)
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(128 * 4 * 4, 128)
        self.relu4 = nn.ReLU()
        self.fc2 = nn.Linear(128, num_classes)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.relu2(x)
        x = self.pool2(x)
        x = self.conv3(x)
        x = self.relu3(x)
        x = self.pool3(x)
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.relu4(x)
        x = self.fc2(x)
        return x
```

2 Model training

From the Figure below we can see, the loss of the model steadily decreases with each epoch, indicating that the model is learning appropriately and improving its performance. Also, the validation accuracy (going from 40% to around 60%) also shows a consistent improvement with each epoch, suggesting that the model is generalizing well to unseen data. Overall, the model seems to be learning effectively, with both the loss and validation accuracy showing positive trends, but not near perfectly.

The fact that validation accuracy is increasing alongside the decreasing loss indicates that the model is not merely memorizing the training data but is learning meaningful patterns that can be applied to new data. The observed trends in the loss and validation accuracy over the epochs indeed reflect a model that is learning effectively, albeit not flawlessly. In summary, while the observed trends are positive indicators of the model's learning process, further analysis and fine-tuning may be necessary to optimize its performance further.

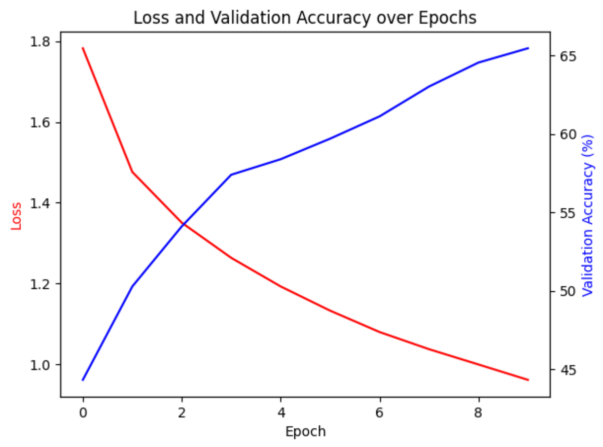

Figure 1: Model evaluation pre-data augmentation

3 Data Augmentation

Note: In this whole exercise, as well as the assignment, I am comparing against the validation set, not a test set, as I was unsure/unclear about this in the given instructions and code. If I were to use a test set, I would have split the validation set further into a validation and test set.

From the Figure below, we can see the training loss consistently decreases over 20 epochs (I chose 10 more epochs than prior to allow for the model to converge), which indicates that the model is learning and improving its ability to minimize the error. On the other hand, the validation accuracy generally increases with each epoch, although there are fluctuations. This indicates that the model is improving its performance on unseen data as training progresses (not just over-fitting). However, it's worth noting that there are some fluctuations in validation accuracy, which could be due to various factors such as model complexity, learning rate, or the presence of noise in the data.

Overall, the decreasing trend in training loss and the increasing trend in validation accuracy suggest that the model is effectively learning and generalizing well to unseen data.

Here is the snippet in the code where I apply the transformations to the `transforms.Compose` module.

```
import torchvision.transforms as transforms
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.RandomHorizontalFlip(),
    transforms.RandomVerticalFlip(),
    transforms.RandomRotation(90),
    transforms.ColorJitter()
    ])
```

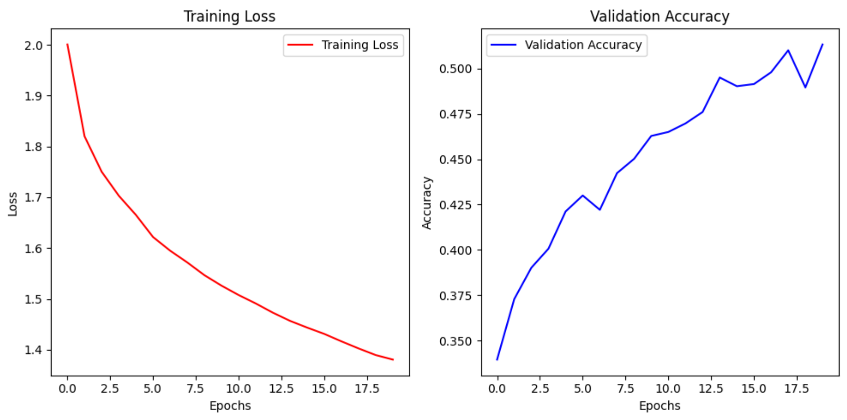

Figure 2: Model evaluation

2 Image denoising

All of the code for this exercise can be found in `exercise2.ipynb`.

1 Peak signal-to-noise ratio (PSNR) metrics

$$\mathrm{PSNR}(x,\hat{x}) = 10 \cdot \log_{10}\left(\frac{\mathrm{MAX}_I^2}{\mathrm{MSE}(x,\hat{x})}\right) \tag{1}$$

PSNR is a metric that measures the quality of a reconstructed image compared to its original version. It's expressed in decibels and higher values indicate better quality. A higher PSNR value indicates better reconstruction quality. However, PSNR is sensitive to changes in image intensity and doesn't necessarily correspond to perceived quality.

```python
import torch.nn.functional as F

def psnr(y_pred, y):
    mse = F.mse_loss(y_pred, y)
    if mse == 0:
        return float('inf')
    max_pixel = 1.0 # Since the images are normalized to [0, 1]
    psnr_value = 20 * torch.log10(max_pixel / torch.sqrt(mse))
    return psnr_value.item()
```

2 Structural similarity index measure (SSIM)

SSIM is a metric that quantifies the similarity between two images based on three factors: luminance, contrast, and structure. It ranges from -1 to 1, where 1 indicates perfect similarity between images. SSIM takes into account the perceived changes in structural information, whereas PSNR only considers pixel-wise differences. A higher SSIM value suggests a higher similarity between the reference and distorted images.

SSIM is often considered to be more aligned with human perception of image quality compared to PSNR. A higher SSIM value suggests a higher similarity between the reference and distorted images.

$$\mathrm{SSIM}(x,\hat{x}) = \frac{(2\mu_x\mu_{\hat{x}} + C_1)(2\sigma_{x\hat{x}} + C_2)}{(\mu_x^2 + \mu_{\hat{x}}^2 + C_1)(\sigma_x^2 + \sigma_{\hat{x}}^2 + C_2)} \tag{2}$$

$$\sigma_{x,y} = ((x \cdot y) * g) - (\mu_x \cdot \mu_y) \tag{3}$$

```python
import torch
import torch.nn.functional as F

def gaussian(window_size, sigma):
    gauss = torch.tensor([np.exp(-(x - window_size // 2) ** 2 / (2 * sigma ** 2)) for x in
        range(window_size)])
    return gauss / gauss.sum()

def create_window(window_size, channel):
    _1D_window = gaussian(window_size, 1.5).unsqueeze(1)
    _2D_window = _1D_window.mm(_1D_window.t()).float().unsqueeze(0).unsqueeze(0)
    window = _2D_window.expand(channel, 1, window_size, window_size).contiguous()
    return window

def ssim(y_pred, y, window_size=11, C1=np.square(0.01), C2=np.square(0.032)):
    channel = y.size(1)
    window = create_window(window_size, channel).to(y.device)

    mu1 = F.conv2d(y_pred, window, padding=window_size//2, groups=channel)
    mu2 = F.conv2d(y, window, padding=window_size//2, groups=channel)

    mu1_sq = mu1 ** 2
    mu2_sq = mu2 ** 2
    mu1_mu2 = mu1 * mu2

    sigma1_sq = F.conv2d(y_pred * y_pred, window, padding=window_size//2, groups=channel) -
        mu1_sq
    sigma2_sq = F.conv2d(y * y, window, padding=window_size//2, groups=channel) - mu2_sq
    sigma12 = F.conv2d(y_pred * y, window, padding=window_size//2, groups=channel) - mu1_mu2

    ssim_map = ((2 * mu1_mu2 + C1) * (2 * sigma12 + C2)) / ((mu1_sq + mu2_sq + C1) * (sigma1_sq
        + sigma2_sq + C2))
    return ssim_map.mean()
```

3 Model training and evaluation

Below is the snippet of code I used to visualize the original, noisy, and denoised versions of the images at the first epoch, halfway, and at the end. From the first Figure (epoch 0), the denoised version of the image does not seem to display that much correction, and in fact seems to display more artifacts than anything. However in the later epochs we can see from the denoised versions of the image that it is smoother, yet also far from perfect.

```
if epoch == 0 or epoch == epochs // 2 or epoch == epochs - 1:
    if i == 0:
        axes[0].imshow(y[0].squeeze().cpu().numpy(), cmap='gray')
        axes[0].set_title('Original')
        axes[1].imshow(x[0].squeeze().cpu().numpy(), cmap='gray')
        axes[1].set_title('Noisy')
        axes[2].imshow(y_pred[0].squeeze().cpu().detach().numpy(), cmap='gray')
        axes[2].set_title('Denoised')
```

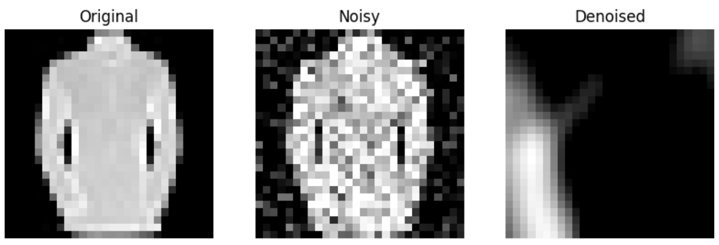

Figure 3: At epoch 0

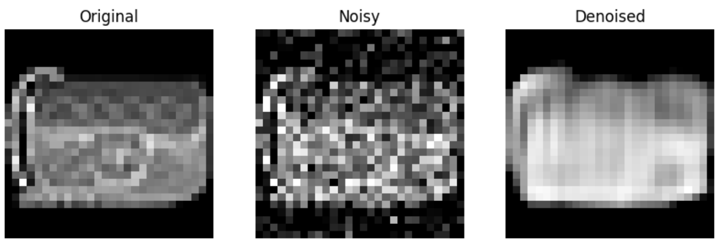

Figure 4: At epoch 25

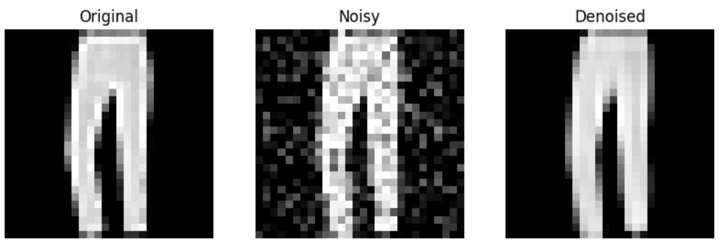

Figure 5: At epoch 50

From the results of the images we can holistically tell that the denoising process improves over the epochs. We can also see SSIM and PSNR rising and stabilizing towards the end of the epoch runs.

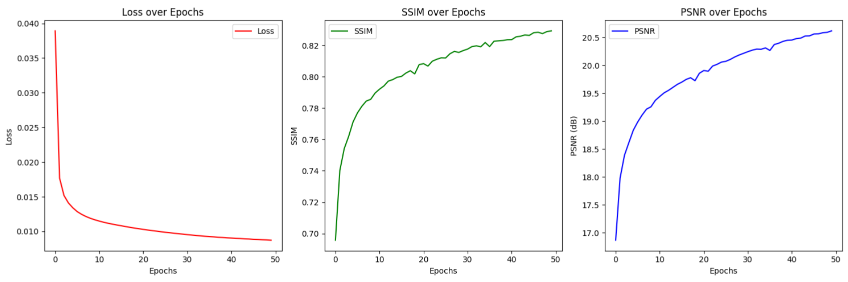

Figure 6: Model evaluation

4 Modify the predefined model

The model has been reconstructed according to the assignment requirements. From the outputted images below we can see that the denoised versions of the images improve over the epochs, without as much of the same artifacts present as in the previous model.

```
class ReconstructionModelConvTranspose(ReconstructionModel):
    def __init__(self, channels=1, height=28, width=28):
        super(ReconstructionModelConvTranspose, self).__init__(channels, height, width)
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(8, 16, kernel_size=3, stride=2, padding=1, output_padding=1),
            nn.SiLU(),
            nn.ConvTranspose2d(16, 32, kernel_size=3, stride=2, padding=1, output_padding=1),
            nn.SiLU(),
            nn.Conv2d(32, channels, kernel_size=3, padding=1)) #final convolution
        self.init_weights()
```

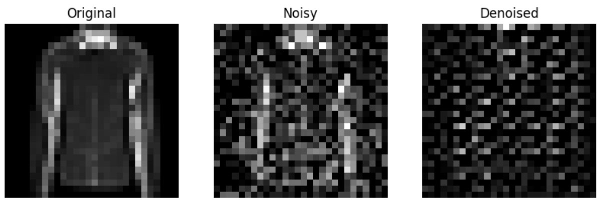

Figure 7: At epoch 0

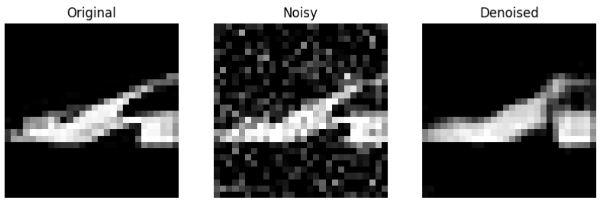

Figure 8: At epoch 25

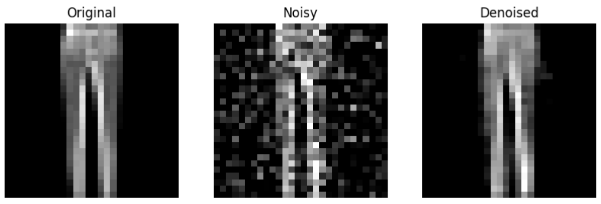

Figure 9: At epoch 50

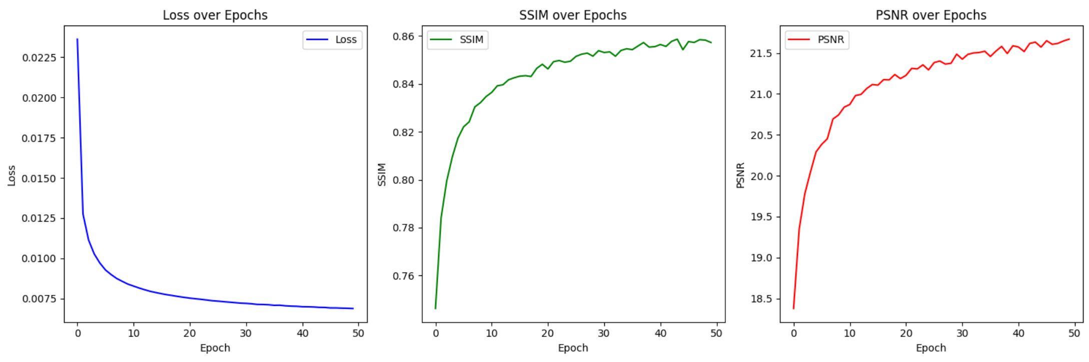

Figure 10: Evaluation for `ConvTranspose2d` model

Similarly, we can comment on the PSNR and SSIM levels over the epochs, as well as the loss; visually assessing the denoised images, the model seems to perform much better than the prior model, and the variations in contrast are certainly not as stark.

3 Image generation

All of the code for this exercise can be found in `exercise3.ipynb`.

1 Model training and evaluation

After plotting the results of the variational autoencoder (VAE) and the conditional variational autoencoder (CVAE) on the `MNIST` data set, the loss over epochs of both of the models seems much greater than of the models we had seen in this assignment prior.

The model architecture might not be expressive enough to capture the underlying distribution of the MNIST (seemingly simpler) dataset. Increasing the complexity of the model, such as adding more layers or increasing the hidden size, could potentially improve performance.

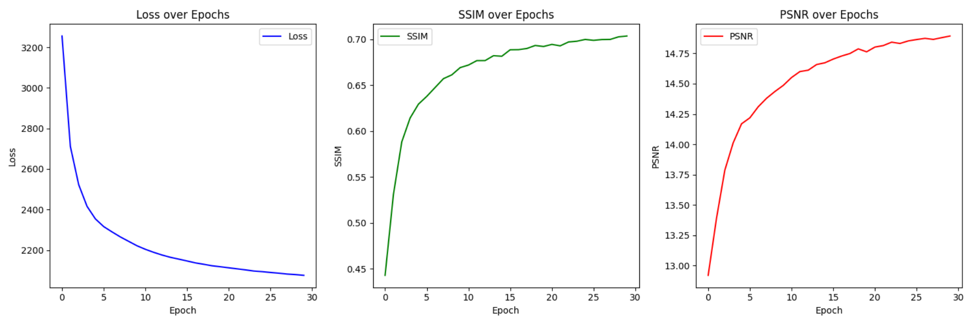

Figure 11: Evaluation for `GenerationModel`

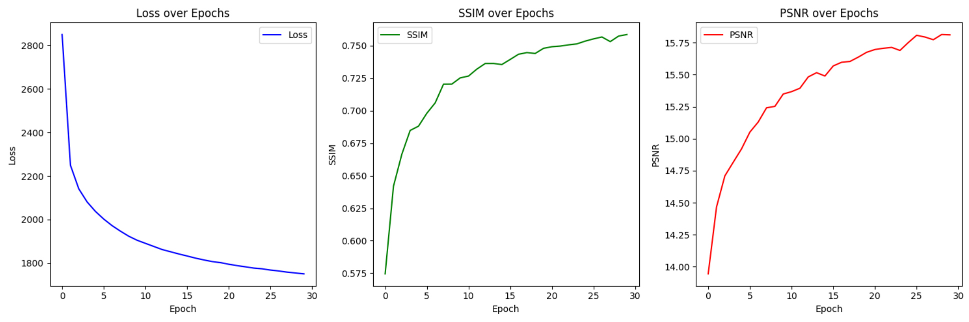

Figure 12: Evaluation for `ConditionalGenerationModel`

To compare both models: the CVAE model exhibits significantly lower loss (yet somehow still high) compared to the original VAE. This indicates that the CVAE is better able to reconstruct the input images while also capturing the underlying distribution of the data. The CVAE consistently shows higher SSIM and PSNR values across epochs compared to the original VAE which is positive in terms of the performance. Overall, the CVAE outperforms the original VAE in terms of loss, SSIM, and PSNR, indicating its effectiveness in generating high-quality images conditioned on specific class labels.

2 Conditional image generation

Below is the code snippet used to display in Figure 12 the digits `0, 1, 2`.

```
digits_to_generate = [0, 1, 2]
n_images_per_digit = 5

for digit in digits_to_generate:
    generated_images = generate_images(conditional_model, digit, n_images_per_digit)
    fig, axes = plt.subplots(1, n_images_per_digit, figsize=(15, 3))
    for i, img in enumerate(generated_images):
        axes[i].imshow(img.squeeze(0), cmap='gray')
        axes[i].axis('off')
    plt.show()
```

From a visual perspective, the images seem to be well representing from the dataset, and speaking from the high SSIM values from prior, we can be confident that the generated images are similar to the given data.

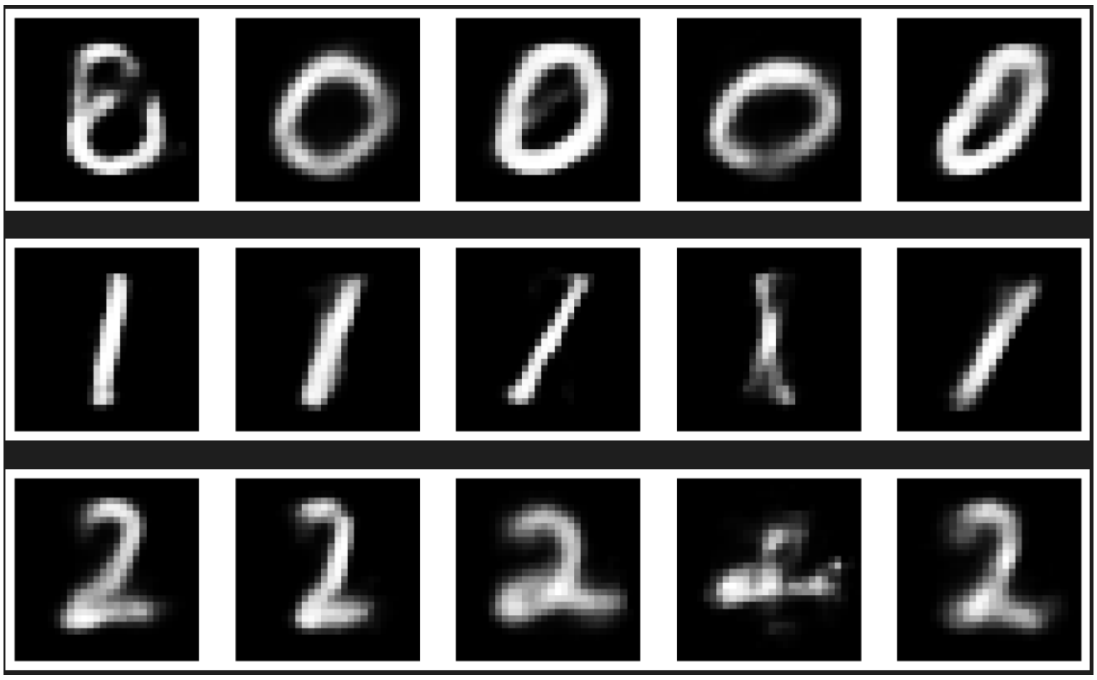

Figure 13: Visual Results of for `ConditionalGenerationModel` on `MNIST` set