Wilbert Aristo Guntoro
ISTD - 1003742

Deep Learning Homework 11 Report

A. Copy and paste the code for your Critic class. Briefly explain your choice of architecture.

```
# Critic
class Critic(nn.Module):

  def __init__(self, image_size):
    """
    Only forced parameter will be the image size, set to 28.
    """
    super().__init__()
    self.critic = nn.Sequential(
      # First Conv Block
      nn.Conv2d(1, 16, 6, stride = 3, padding = 1, bias=True),
      nn.LeakyReLU(0.2, inplace=True),
      # Second Conv Block
      nn.Conv2d(16, 32, 5, stride = 2, padding = 1, bias=True),
      nn.LeakyReLU(0.2, inplace=True),
      # Third Conv Block
      nn.Conv2d(32, 64, 4, stride = 2, padding = 0, bias=True),
      nn.LeakyReLU(0.2, inplace=True),
    )

  def forward(self, x):
    return self.critic(x)
```

I decided to go with a very simple architecture since we are working with a very simple dataset (MNIST). The convolution layers are used to extract features from the image as well as to downsample the input by setting the stride as > 1. I set the parameters of the convolution layers (e.g. filter size, stride, and padding) such that it will produce an output of shape [64, 1, 1] to accommodate our latent size which is fixed at 64. For the activation function, I decided to use LeakyReLU to allow some negative gradients to pass through, preventing the network from producing zeros for all the outputs as in the case for ReLU (dying state).

I also experimented with several architecture designs before settling on this one. For example, **on separate training instances**, I tried setting the bias of the convolution layers to False, adding batch normalization and dropout layers, and using Sigmoid as the last activation function. However, after several training instances, I found out that they performed worse than the simple architecture shown above.

B. Copy and paste the code for your Generator class. Briefly explain your choice of architecture.

```
# Generator
class Generator(nn.Module):
  def __init__(self, latent_size, image_size):
    """
    Only forced parameters will be the image size, set to 28,
    and the latent size set to 64.
    """
    super().__init__()
    self.generator = nn.Sequential(
      # First Conv Block
      nn.ConvTranspose2d(latent_size, 32, 4, stride = 2, padding = 0,\
                         bias=True),
      nn.LeakyReLU(0.2, inplace=True),
      # Second Conv Block
      nn.ConvTranspose2d(32, 16, 5, stride = 2, padding = 1, bias=True),
      nn.LeakyReLU(0.2, inplace=True),
      # Third Conv Block
      nn.ConvTranspose2d(16, 1, 6, stride = 3, padding = 1, bias=True),
      nn.LeakyReLU(0.2, inplace=True),
    )

  def forward(self, x):
    return self.generator(x);
```

My Generator model is the reverse/mirror of the Critic model shown in part A. Instead of using convolutions to downsample an input image, the generator model aims to upsample the bottleneck layer (output of the Critic model) to the original input image's shape (1 X 28 x 28). I did this by replacing Conv2d layers with ConvTranspose2d layers and making sure the parameters (filter size, stride, and padding) is a symmetric mirror to that of the Critic model's convolution layers. This symmetry would ensure a successful upsampling back to the original image's shape and a better consistency. The activation function used in the Generator model is still LeakyReLU (same as the Critic model) to ensure consistency.

C. For how many iterations did you have to train when using Wasserstein with Conv or TransposeConv layers to get plausible images from the generator? Is it training faster than the Fully Connected Wasserstein/Vanilla GAN?

I trained the network for 300 epochs with a batch size of 128. MNIST has a training dataset of 60,000 examples, meaning 1 epoch would have 60,000 ÷ 128 ≈ 468 iterations. Therefore, I trained the network through 468 * 300 = 140,400 iterations.

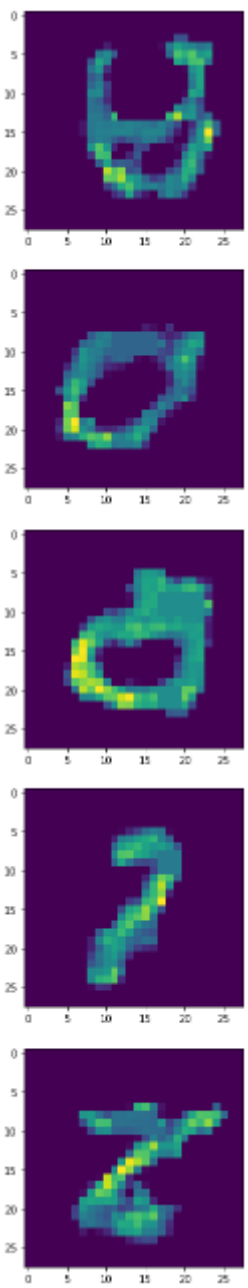
The training is much faster compared to Fully Connected Wasserstein/Vanilla GAN because we utilise convolution layers instead of fully connected layers. Convolution layers have much less parameters to use and train, thus it is faster to train. Even though I utilised 3 convolution layers as compared to 2 fully connected layers in Vanilla GAN, my training speed is still faster than that of Vanilla Gan's.

E. Let us assume we use Conv2d layers in the Critic. We do NOT use Transposed Conv2d layers, but only Fully Connected layers in the Generator. Would the GAN still be able to train both models or would it encounter difficulties? Discuss.

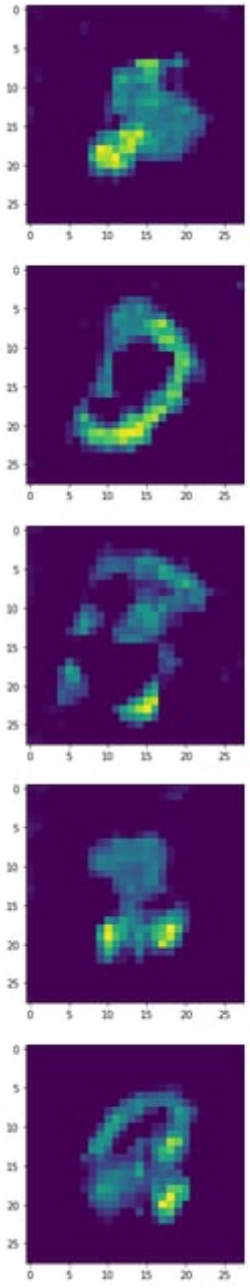
No. Since the output of the generator must be fed into the Critic model, there will be some trouble converting the output of FC layer which is a 1 dimensional tensor to a 3 dimensional tensor (as required by Conv2d layers) without sacrificing the integrity of information on features, etc. This is a difficulty that GAN will face if its Generator model is made up of Fully Connected layers.

Also, assuming we did downsampling in the Conv2d layers in the Critic model, the Fully Connected layers in the Generator model would not be able to upsample the input back to the original image size (28 x 28). This is a major problem as it does not follow the principles of autoencoding.

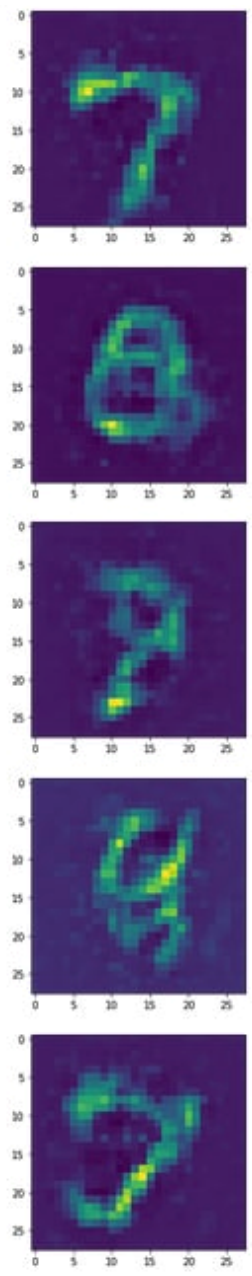
D. Display some samples generated by your trained generator. Do they look plausible?



1st training instance



2nd training instance



3rd training instance

Above are screenshots of samples generated by my trained generator from three separate training instances. Most of the samples look quite plausible, with the exception of some (for example the first image of the first training instance, last two images of the third training instance, etc.).