Assignment 04: Generative Models
CSI5340 – Introduction to deep learning and reinforcement learning
Cristopher McIntyre Garcia - 300025114

Designs
In this assignment three generative models were designed with several configurations. The first model is a Variational AutoEncoder (VAE), the second is a Generative Adversarial Network (GAN), and the third is a Wasserstein Generative Adversarial Network (WGAN). Each model was implemented in such a way as to allow the user to use different network depths and latent space dimensions. In this particular implementation, changing the model depth will also change the size of the image. Notation for a model with y [minus] 2 hidden layers (and y total layers) will be represented as MODEL x 2^y. This was done to study the network performance on different image sizes when limited changes are done to the complexity of the model. Each network module is implemented as a configuration of deep Convolutional Neural Network (CNN) and Linear Networks. To keep training code consistent, the loss of each network is calculated in the global module.

VAE
The VAE consists of an encoder and a decoder network. The encoder network is a simple architecture of convolution layers followed by linear layers. Every convolution layer is followed by a batch normalization layer and an activation function. The final two linear layers are branched from a former linear layer to produce the mean and standard deviation of the Gaussian distribution particular to the class of the input image, in the latent space.

The decoder network can be thought of as the reverse of the encoder network. Firstly, a latent representation is fed into a linear layer which maps it to a dimension compatible with the first hidden layer. The result is passed through a set of transpose convolution layers to upscale the representation. The transpose convolution layers are all followed by a batch normalization layer and an activation function. The final layer's activation function is tanh. With the encoder and decoder, the VAE is able to compute the reconstruction and KL divergence loss of an image and a provided reconstruction. The steps can be seen in the following modules.

```
def Encoder(x, depth, h_dim, z_dim):
        for _ in range(depth):
                x = conv(x)
                x = batch_norm(x)
                x = relu(x)
        z = linear(x, h_dim)
        μ = linear(z, z_dim)
        σ = linear(z, z_dim)
        return  μ, σ
```

Module 1 (a) Encoder

```
def Decoder(z, depth, h_dim):
        x' = linear(z, h_dim)
        for _ in range(depth-1):
                x' = transpose(x')
                x' = batch_norm(x')
                x' = leakyRelu(x')
        x' = conv(x')
        x' = batch_norm(x')
        x' = tanh(x')
        return  x'
```

Module 1 (b) Decoder

```
def VAE(x, encoder, decoder):
        μ, σ = encoder(x)
        z = N(σ) * σ +  μ
        x' = decoder(z)
        rec = mse(x, x')
        KL = -0.5 * mean(sum(1 + log(σ^2) - μ^2 - e^log(σ^2)))
        return  rec + KL * KL_factor
```

Module 1 (c) VAE

It is up to the encoder to produce proper latent space distribution parameters, and it is up to the decoder to generate an image given a latent representation. For generating data, the generator is solely used as long as we have the parameters for the distribution of the data we wish to generate. If not, we either chose a random sample or generated the parameters by auto-encoding an image from the same class.

Examples of a VAE models are shown below:

VAE x 32 – z_dim 100

```
VAE(
  (encoder): Encoder(
    (conv): Sequential(
      (0): Sequential(
        (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2))
      (1): Sequential(
        (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2))
      (2): Sequential(
        (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2))
      (3): Sequential(
        (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2)))
    (h_layer): Linear(in_features=1024, out_features=256, bias=True)
    (mu_layer): Linear(in_features=256, out_features=100, bias=True)
    (std_layer): Linear(in_features=256, out_features=100, bias=True)
)
  (decoder): Decoder(
    (z_linear): Linear(in_features=100, out_features=2048, bias=True)
    (convTranspose): Sequential(
      (0): Sequential(
        (0): ConvTranspose2d(512, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
                                                                    output_padding=(1, 1))
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
      (2): ReLU())
    (1): Sequential(
      (0): ConvTranspose2d(256, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
                                                                    output_padding=(1, 1))
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU())
    (2): Sequential(
      (0): ConvTranspose2d(128, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
                                                                    output_padding=(1, 1))
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU())
    (3): Sequential(
      (0): ConvTranspose2d(64, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
                                                                   output_padding=(1, 1))
      (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU()))
  (final_layer): Sequential(
    (0): ConvTranspose2d(32, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
                                                                   output_padding=(1, 1))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): Conv2d(32, 1, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (4): Tanh()))
)
```

VAE x 16 – z_dim 20

```
VAE(
  (encoder): Encoder(
    (conv): Sequential(
      (0): Sequential(
        (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2))
      (1): Sequential(
        (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2))
      (2): Sequential(
        (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2)))
    (h_layer): Linear(in_features=512, out_features=128, bias=True)
    (mu_layer): Linear(in_features=128, out_features=20, bias=True)
    (std_layer): Linear(in_features=128, out_features=20, bias=True)
  )
  (decoder): Decoder(
    (z_linear): Linear(in_features=20, out_features=1024, bias=True)
    (convTranspose): Sequential(
      (0): Sequential(
        (0): ConvTranspose2d(256, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
                                                                    output_padding=(1, 1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU())
      (1): Sequential(
        (0): ConvTranspose2d(128, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
                                                                    output_padding=(1, 1))
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU())
      (2): Sequential(
        (0): ConvTranspose2d(64, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
                                                                    output_padding=(1, 1))
        (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU()))
    (final_layer): Sequential(
      (0): ConvTranspose2d(32, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
                                                                    output_padding=(1, 1))
      (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU()
      (3): Conv2d(32, 1, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
      (4): Tanh()))
)
```

GAN

The GAN consists of a generator and a discriminator network. The generator network is an architecture of transpose convolution layers, which are all followed by a batch normalization layer and an activation function. The finally transpose convolution layer omits the batch normalization layer, and uses tanh as the activation function. The purpose of the generator is to realize an image from a latent representation.

The discriminator is an architecture of convolution layers, which are mostly all followed by a batch normalization layer and an activation function. The first convolution layer is not followed by a batch normalization layer, and the final convolution layer is followed by a sigmoid activation function. After the convolution layers, a linear function maps the representation to 1 dimension. The purpose of the discriminator is to identify whether a given image comes from the dataset distribution or not. With the generator and discriminator, the GAN is able to compute the generative and discriminative losses of a real and fake image. The steps can be seen in the sudo code found in the following modules.

```
def Generator(z, depth):
        for _ in range(depth):
                z = conv_transpose(z)
                z = batch_norm(z)
                z = relu(z)
        x' = conv_transpose(z)
        x' = tanh(x')
        return x'
```

Module 2 (a) Generator

```
def Discriminator(x, depth):
        x = conv(x)
        x = leakyRelu(x)
        for _ in range(depth-1):
                x = conv(x)
                x = batch_norm(x)
                x = leakyRelu(x)
        x = conv(x)
        x = batch_norm(x)
        x = sigmoid(x)
        i = linear(x, 1)
        return  i
```

Module 2 (b) Discriminator

```
def GAN(x, generator, discriminator):
        z = N(0, 1)
        x' = generator(z)
        real = discriminator(x)
        fake = discriminator(x')
        gen_loss = mse(fake, 1)
        disc_loss = (mse(fake, 0) + mse(real, 1)) / 2
        return gen_loss, disc_loss
```

Module 2 (c) GAN

It is up to the generator to generate images from a latent representation that can fool the discriminator into classifying them as real images. It is up to the discriminator to correctly classify whether an image was generated or came from the dataset distribution. The loss computed will be used to minimize the JSD between the distribution of generated images and that of the dataset.

Examples of a GAN models are shown below:

GAN x 32 – z_dim 100

```
GAN(
  (generator): Generator(
    (conv): Sequential(
      (0): Sequential(
        (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1))
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU())
      (1): Sequential(
        (0): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
        (1): Sequential(
          (0): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (1): ReLU()))
      (2): Sequential(
        (0): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
        (1): Sequential(
          (0): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (1): ReLU()))
      (3): Sequential(
        (0): ConvTranspose2d(128, 1, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
        (1): Tanh())))
  )
  (discriminator): Discriminator(
    (conv): Sequential(
      (0): Sequential(
        (0): Conv2d(1, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
        (1): LeakyReLU(negative_slope=0.2))
      (1): Sequential(
        (0): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2))
      (2): Sequential(
        (0): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): Sigmoid()))
    (final_layer): Sequential(
      (0): Linear(in_features=4096, out_features=1, bias=True)))
)
```

GAN x 16 – z_dim 20

```
GAN(
  (generator): Generator(
    (conv): Sequential(
      (0): Sequential(
        (0): ConvTranspose2d(20, 256, kernel_size=(4, 4), stride=(1, 1))
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU())
      (1): Sequential(
        (0): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
        (1): Sequential(
          (0): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (1): ReLU()))
      (2): Sequential(
        (0): ConvTranspose2d(128, 1, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
        (1): Tanh())))
  )
  (discriminator): Discriminator(
    (conv): Sequential(
      (0): Sequential(
        (0): Conv2d(1, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
        (1): LeakyReLU(negative_slope=0.2))
      (1): Sequential(
        (0): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): Sigmoid()))
    (final_layer): Sequential(
      (0): Linear(in_features=2048, out_features=1, bias=True)))
)
```

WGAN

The WGAN is very similar to the GAN and mostly differs in its means of calculating the loss. The generator is identical to that found in the GAN. The critic is identical to the discriminator except for the final activation function. The discriminator uses the sigmoid activation function, while the critic does not. The steps can be seen in the sudo code found in the following modules.

```
def Generator(z, depth):
        for _ in range(depth):
                z = conv_transpose(z)
                z = batch_norm(z)
                z = relu(z)
        x' = conv_transpose(z)
        x' = tanh(x')
        return x'
```

Module 3 (a) Generator

```
def Critic(x, depth):
        x = conv(x)
        x = leakyRelu(x)
        for _ in range(depth-1):
                x = conv(x)
                x = batch_norm(x)
                x = leakyRelu(x)
        x = conv(x)
        x = batch_norm(x)
        x = leakyRelu(x)
        i = linear(x, 1)
        return  i
```

Module 3 (b) Critic

```
def GAN(x, generator, critic):
        z = N(0, 1)
        x' = generator(z)
        real = critic(x)
        fake = critic(x')
        gen_loss = - mean(fake)
        critic_loss = mean(fake) - mean(real)
        return gen_loss, disc_loss
```

Module 3 (c) WGAN

The generator plays the same role as it did in the GAN network. The critic plays a similar role as the discriminator from the GAN network. The difference here is that the critic is not determining whether an image is fake or real; instead, it evaluates how likely it is to be real. The loss is used to minimize the Wasserstein distance between the distribution of generated images and that of the dataset.

Examples of WGAN models are shown below:

WGAN x 32 – z_dim 100

```
WGAN(
  (generator): Generator(
    (conv): Sequential(
      (0): Sequential(
        (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1))
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU())
      (1): Sequential(
        (0): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
        (1): Sequential(
          (0): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (1): ReLU()))
      (2): Sequential(
        (0): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
        (1): Sequential(
          (0): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (1): ReLU()))
      (3): Sequential(
        (0): ConvTranspose2d(128, 1, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
        (1): Tanh())))
  )
  (critic): Discriminator(
    (conv): Sequential(
      (0): Sequential(
        (0): Conv2d(1, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
        (1): LeakyReLU(negative_slope=0.2))
      (1): Sequential(
        (0): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2))
      (2): Sequential(
        (0): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2)))
    (final_layer): Sequential(
      (0): Linear(in_features=4096, out_features=1, bias=True)))
)
```

WGAN x 16 – z_dim 20

```
WGAN(
  (generator): Generator(
    (conv): Sequential(
      (0): Sequential(
        (0): ConvTranspose2d(20, 256, kernel_size=(4, 4), stride=(1, 1))
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU())
      (1): Sequential(
        (0): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
        (1): Sequential(
          (0): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (1): ReLU()))
      (2): Sequential(
        (0): ConvTranspose2d(128, 1, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
        (1): Tanh())))
  )
  (critic): Discriminator(
    (conv): Sequential(
      (0): Sequential(
        (0): Conv2d(1, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
        (1): LeakyReLU(negative_slope=0.2))
      (1): Sequential(
        (0): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2)))
    (final_layer): Sequential(
      (0): Linear(in_features=2048, out_features=1, bias=True)))
)
```

Training

Because the loss is computed in the network modules, training is simple and only involves running the model and performing backward propagation. The VAE is the simplest because the entire model is trained with a single optimizer. Below, we see the algorithm for training the VAE.

```
def train_VAE(model, x):
        for _ in range(epochs):
                for _ in range(iterations)
                        loss = model(x)
                        loss.backward()
                        optimizer.step()
        return model
```

Algorithm 1 Train VAE

The loss returned by the model is a combination of the KL divergence and the reconstruction loss. Below we see the algorithm for training the VAE and the loss curve.

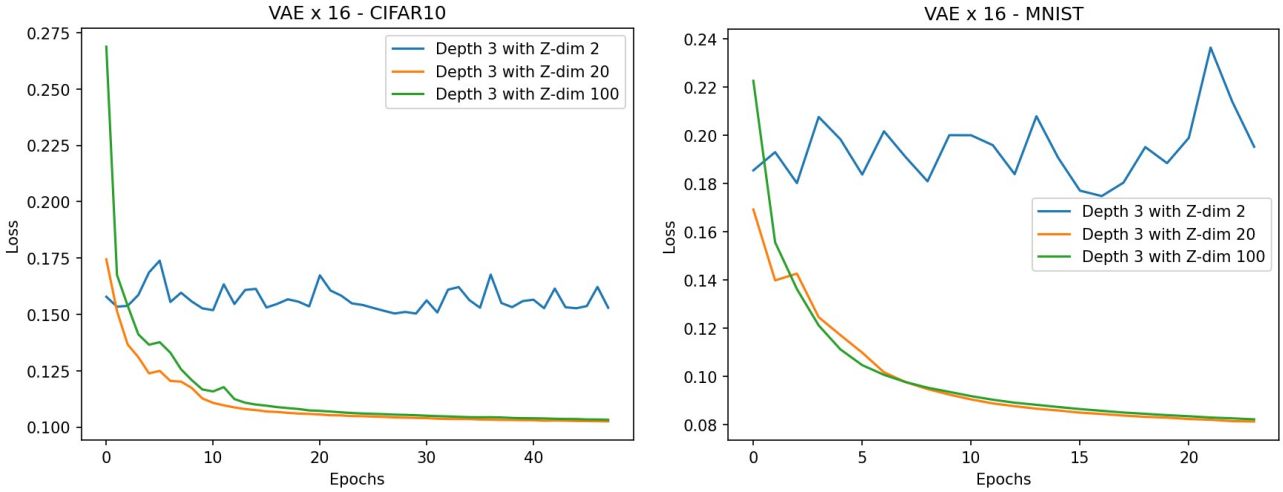


Figure. 1 VAE x 16 generated images – (a) CIFAR10 (b) MNIST

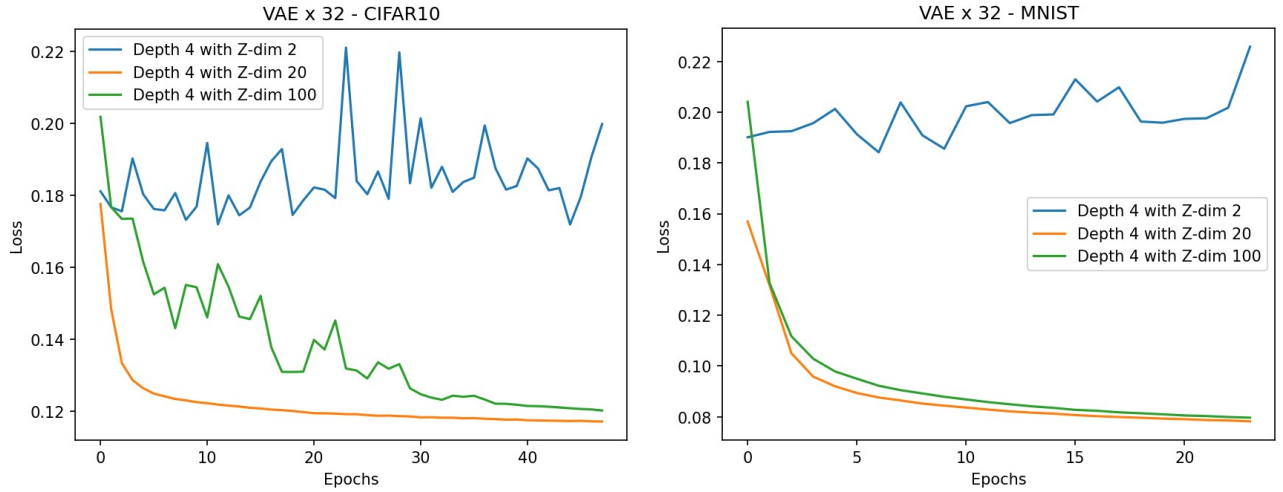


Figure. 2 VAE x 32 generated images – (a) CIFAR10 (b) MNIST

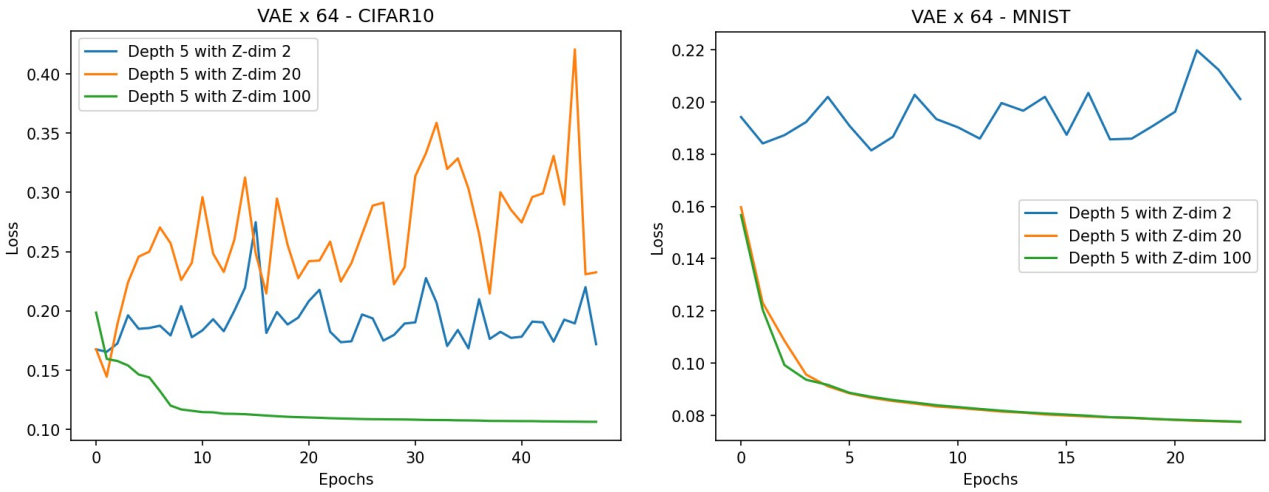


Figure. 3 VAE x 64 generated images – (a) CIFAR10 (b) MNIST

As shown in the above graphs, when the image sizes are augmented, the model needs a larger latent space dimension. It seems that without a larger dimension, the loss seems to no longer approximate zero and instead converges to a constant. This is more apparent when the images are complex. In every case, using a 2-dimensional latent space will make the loss converge to around 0.18. When using a 20-dimensional latent space, the loss converges to around 0.25 when the images are large and complex. Using a 100-dimensional latent space is a safe bet because the loss always converges to 0. That said, sometimes using a smaller latent space will accelerate the loss, which is shown in figure 2.

The GAN and WGAN are trained similarly to each other. Because the modules of these networks are essentially competing with each other, they need to be trained separately. Before the generator, the discriminator and critic can be trained for several iterations. Below we see the algorithm for training the GAN and WGAN. The WGAN has an additional clamping method, not included below, that will fix the weights of the network after each gradient update.

```
def train_GAN(model, x):
        for _ in range(epochs):
                for _ in range(iterations)
                        for _ in range(K)
                                D_loss = model(x)
                                D_loss.backward()
                                D_optimizer.step()
                        G_loss = model(x)
                        G_loss.backward()
                        G_optimizer.step()
        return model
```

Algorithm 2 Train GAN/WGAN

The loss returned by the GAN's generator is the estimated JSD, or more specifically, the virtual training criterion.

$$C(G) = -\log(4) + 2 \cdot JSD\left(p_{\text{data}} \| p_g\right)$$

Eq. 1 Virtual training criterion

Here, JSD is a metric that dictates the distance between p_{data} (the probability distribution of the data) and p_g (the probability distribution of the generated data). When performing optimality, the loss should be around log(2), which is somewhat shown in the results below when the latent dimension is large enough. However, this makes it difficult to know whether the model is properly learning.

The loss curve of the JSD is shown below.

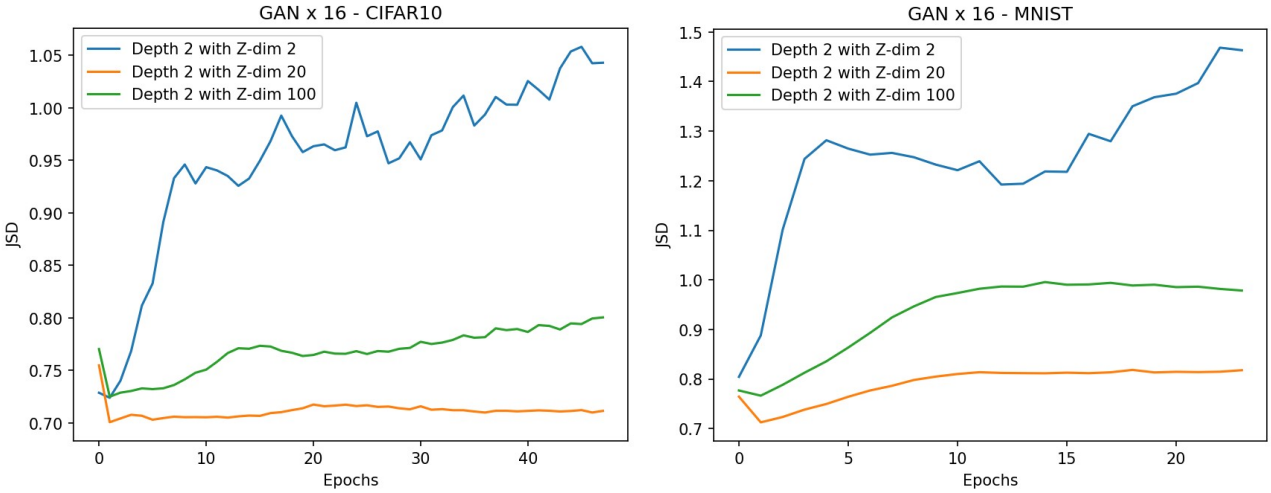


Figure. 4 GAN x 16 generated images – (a) CIFAR10 (b) MNIST

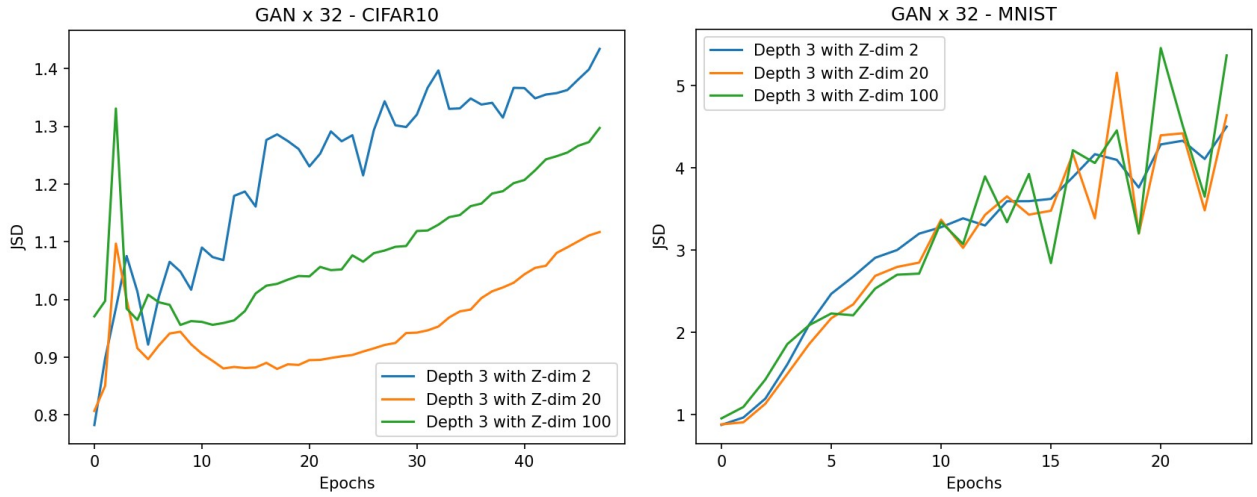


Figure. 5 GAN x 32 generated images – (a) CIFAR10 (b) MNIST

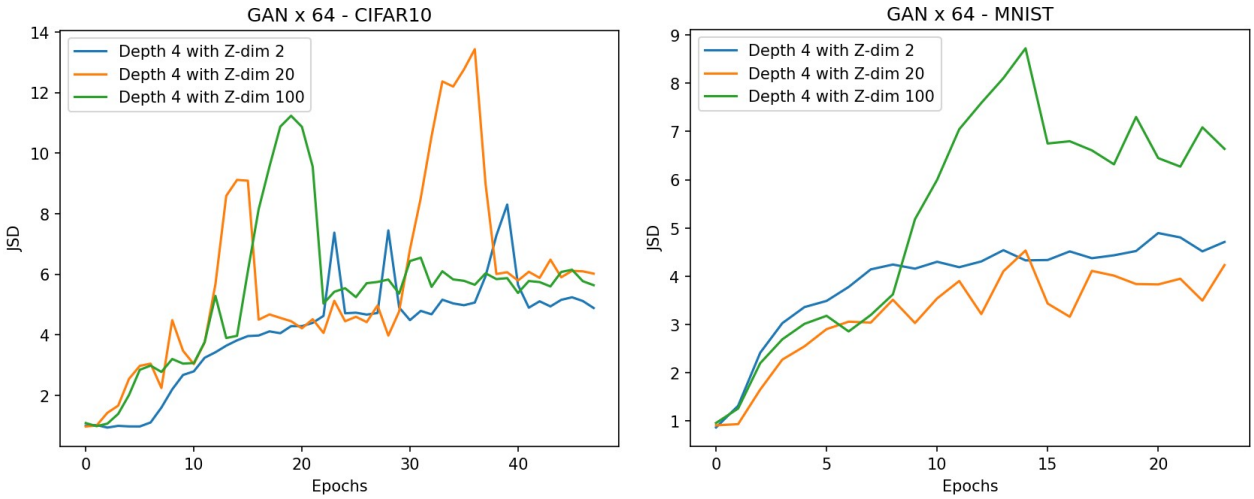


Figure. 6 GAN x 64 generated images – (a) CIFAR10 (b) MNIST

When using smaller images, the model seems to perform better. This is due to the ease of matching the distribution of a dataset with smaller, more simple images. In these tests, the discriminator was trained for an equal number of iterations with the generator. This is because over training the discriminator could lead to vanishing gradients, which is caused by the saturation of the JSD.

The loss returned by the WGAN's generator is the EM distance (Wasserstein-1) estimation, which is minimized with training.

$$W(\mathbb{P}_r, \mathbb{P}_g) = \inf_{\gamma \in \Pi(\mathbb{P}_r, \mathbb{P}_g)} \mathbb{E}_{(x,y)\sim\gamma} \left[\|x - y\| \right]$$

Eq. 2 EM distance

$$L_{gen}(w) = \min_{\theta} \mathbb{E}_{x\sim\mathbb{P}_r}[f_w(x)] - \mathbb{E}_{z\sim Z}[f_w(g_\theta(z))] = \min_{\theta} -\mathbb{E}_{z\sim Z}[f_w(g_\theta(z))]$$

Eq. 3 Generator Objective Function

where P_r denotes the distribution being transformed into P_g by transporting the "mass" of x to y. x and y are pulled from the joint distribution between P_r and P_g. This loss is minimized by optimally training the critic before training the generator. The closer the EM distance is to zero, the better the generator performs. This, in contrast to the JSD, makes the learning curve converge to zero and is a good representation of how good the generated image will be. It is also less likely to suffer from vanishing gradients, and thus should be more stable and robust to parameter differences.

The loss curve of the EM distance is shown below.

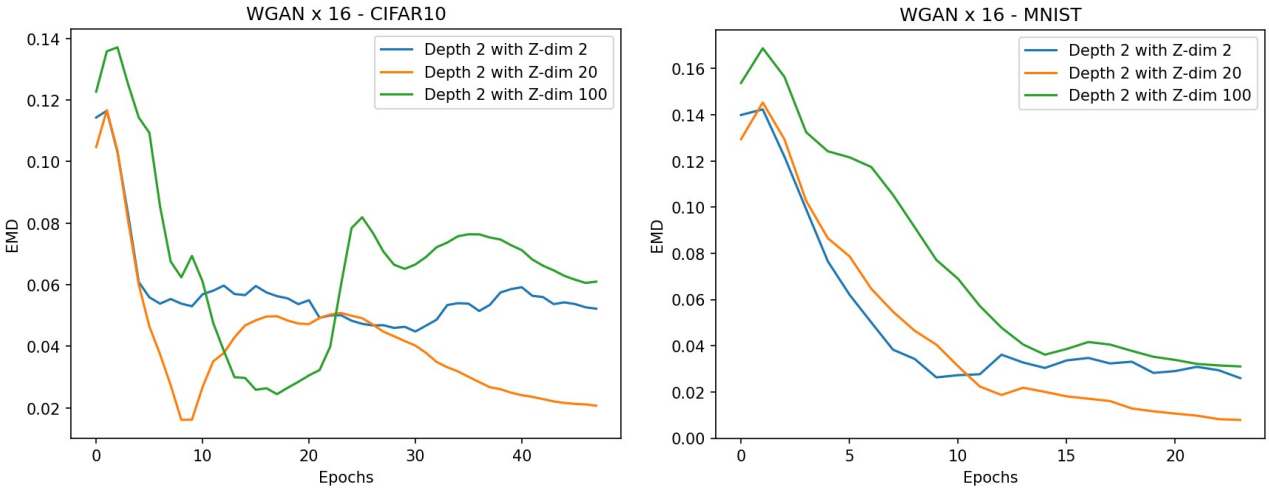


Figure. 7 WGAN x 16 generated images – (a) CIFAR10 (b) MNIST

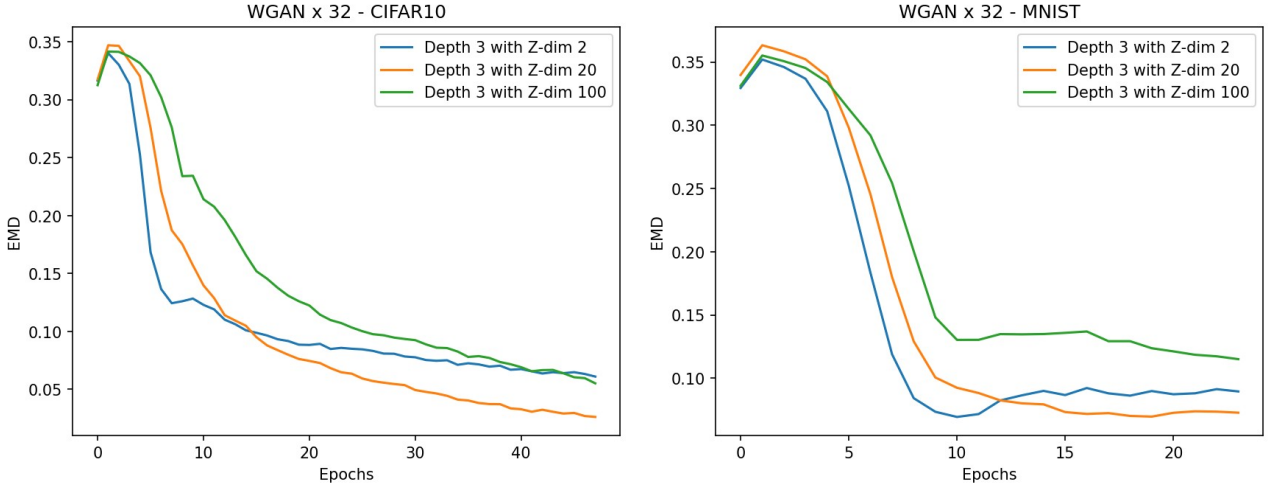


Figure. 8 WGAN x 32 generated images – (a) CIFAR10 (b) MNIST

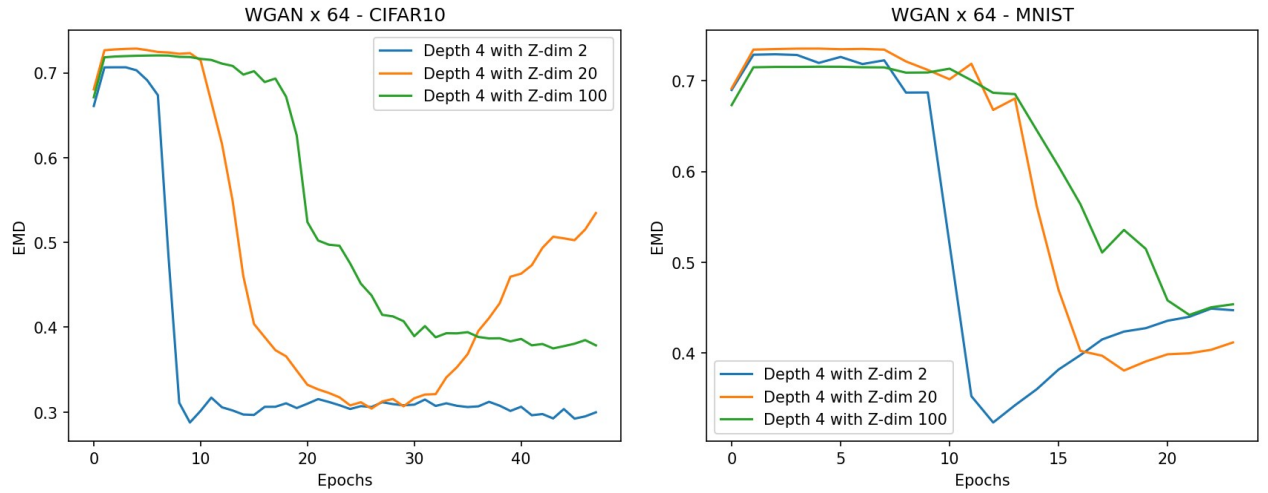


Figure. 9 WGAN x 64 generated images – (a) CIFAR10 (b) MNIST

As shown in the above figures, the loss function of the WGAN tends to converge towards zero. We see this in every case, regardless of the parameters (although some do perform better than others). Using a large latent dimension is a safe bet for producing a learning curve that smoothly converges to zero. If the images are not too large, a latent dimension of 20 will usually outperform the other options.

VAE Results

First, we look at the results obtained when using the VAE model. Depending on the KL factor chosen, the generated images will look more or less similar to the original image. During training, the KL factor is unchanged regardless of latent dimension and image size. Epochs 20, 30, 40, 50:

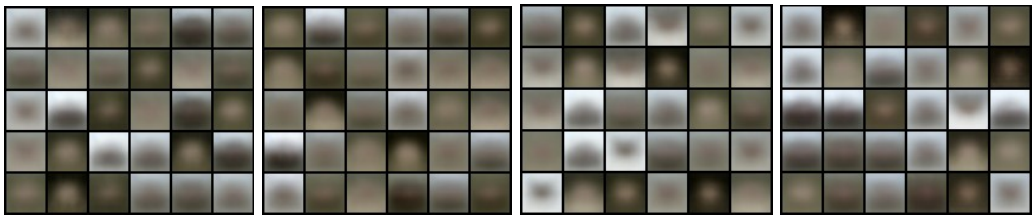


Figure. 10 VAE x 16 generated images – z_dim 2 – CIFAR10

Figure. 11 VAE x 16 generated images – z_dim 20 – CIFAR10

Figure. 12 VAE x 16 generated images – z_dim 100 – CIFAR10

Figure. 13 VAE x 32 generated images – z_dim 2 – CIFAR10

Figure. 14 VAE x 32 generated images – z_dim 20 – CIFAR10

Figure. 15 VAE x 32 generated images – z_dim 100 – CIFAR10

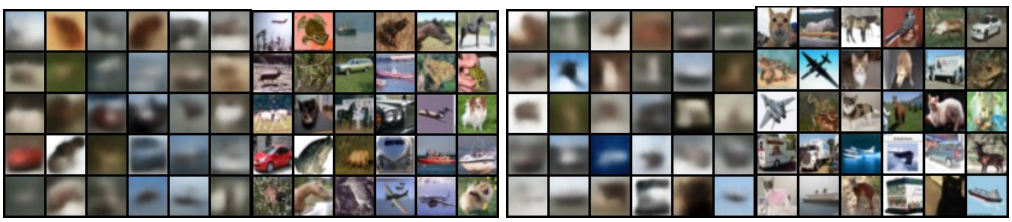
Figure. 16 VAE x 64 generated images – z_dim 2 – CIFAR10

Figure. 17 VAE x 64 generated images – z_dim 20 – CIFAR10

Figure. 18 VAE x 64 generated images – z_dim 100 – CIFAR10

The results look cloudy, but shapes can be made out when the model is performing well. The model performs poorly when the latent dimension is low. When the images are larger, the model also performs poorly when the latent dimension is 20. The best results are found in figure 14 and figure 18. The variety in images is a lot larger when the latent dimension is large, and there is also a lot more expressivity.

Some comparisons to the reference images are below:



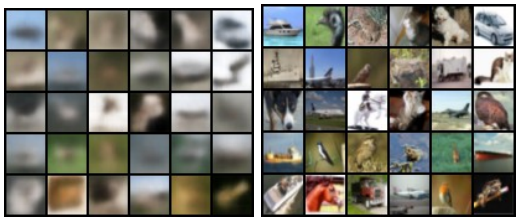


Figure. 19 VAE x 64 generated images – z_dim 100 – CIFAR10 – Generated vs Reference

Some of the images look like blurry versions of the original; however, there are a few cases where they differ from the original in a way that makes sense.

More Results are found in the respective training folder. The MNIST results are found at the end of the report.

GAN Results

Next we look at the results obtained when using the GAN model. Epochs 20, 30, 40, 50:

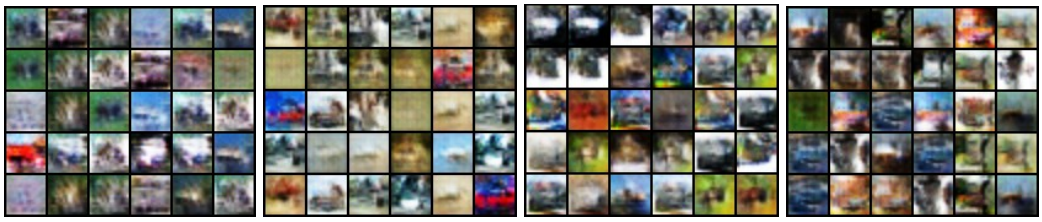


Figure. 20 GAN x 16 generated images – z_dim 2 – CIFAR10

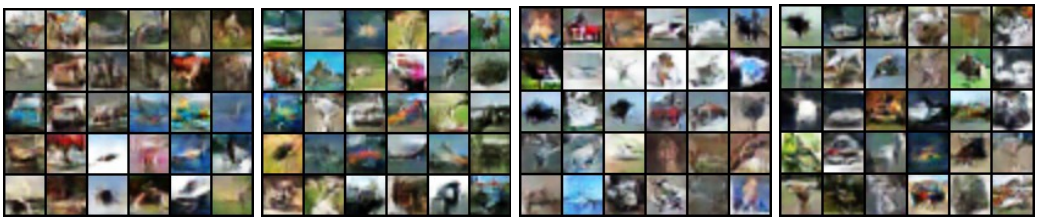


Figure. 21 GAN x 16 generated images – z_dim 20 – CIFAR10

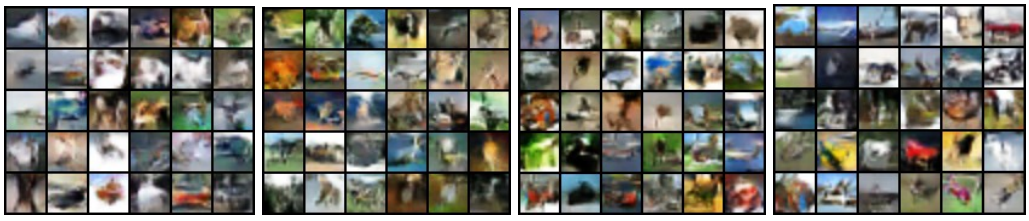


Figure. 22 GAN x 16 generated images – z_dim 100 – CIFAR10

Figure. 23 GAN x 32 generated images – z_dim 2 – CIFAR10

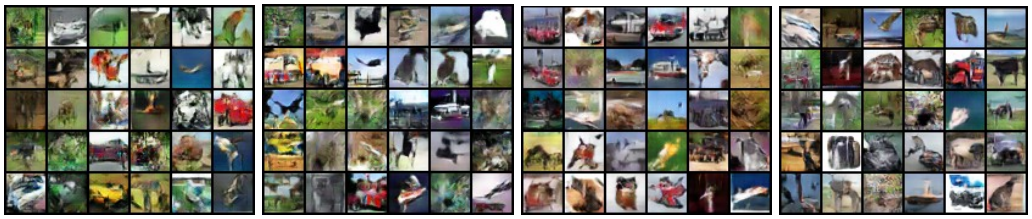


Figure. 24 GAN x 32 generated images – z_dim 20 – CIFAR10

Figure. 25 GAN x 32 generated images – z_dim 100 – CIFAR10

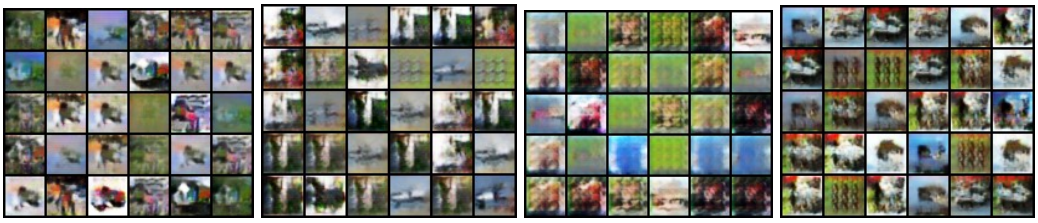


Figure. 26 GAN x 64 generated images – z_dim 2 – CIFAR10

Figure. 27 GAN x 64 generated images – z_dim 20 – CIFAR10

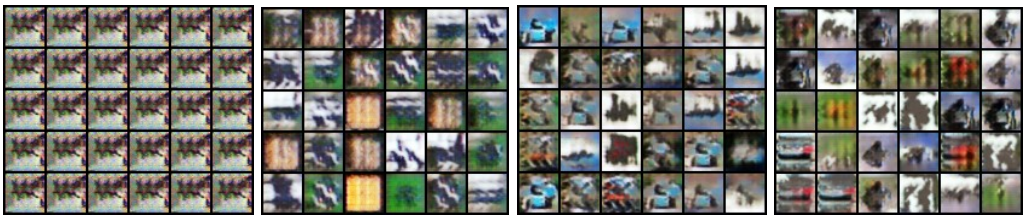


Figure. 28 GAN x 64 generated images – z_dim 100 – CIFAR10

These images look much sharper than those generated using the VAE network. There also seems to be more variation and ease when training with smaller latent dimensions. That said, there are still repeating images when a two-dimensional latent space is used. In the earlier epochs, the images looked like painted versions of images from the original dataset. As the training progresses, the shapes of the objects in the images become more apparent, but the pasty paint-like texture never fully fades away.

The model has difficulty learning when the image size is larger and when the latent dimension is smaller. These results correlate to the loss curve seen above in figures 4 through 6. When the loss is constantly around log(2), the model seems to be learning how to generate better images. As the curve strays away from log(2), the quality of the images lowers. A good example of this is seen in figure 26. The best results are found in figure 21 and figure 24.

The previous results were obtained by training the generator every time the discriminator was trained. In an attempt to improve the results shown in figure 28, I decided to increase the number of discriminator training. In the below figures, K is the number of training iterations that the discriminator undergoes before the generator is trained.

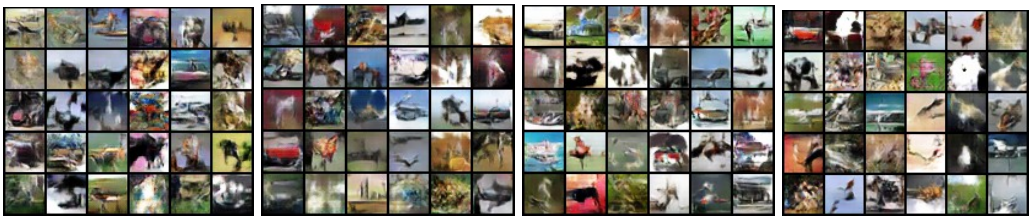


Figure. 29 GAN x 64 generated images – z_dim 100 – CIFAR10 – K = 3

There is a clear difference in the quality of the generated images between figure 28 and figure 29. This leads to believe that training the discriminator more than the generator is beneficial when larger images are being used to learn.

More Results are found in the respective training folder. The MNIST results are found at the end of the report.

WGAN Results

Next we look at the results obtained when using the WGAN model. Epochs 20, 30, 40, 50
(5 critic iterations for each generator iteration):

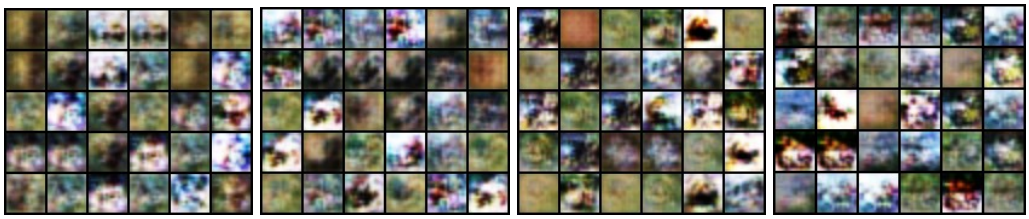


Figure. 30 WGAN x 16 generated images – z_dim 2 – CIFAR10

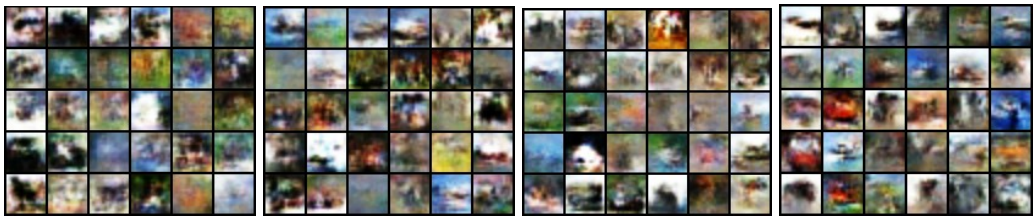


Figure. 31 WGAN x 16 generated images – z_dim 20 – CIFAR10

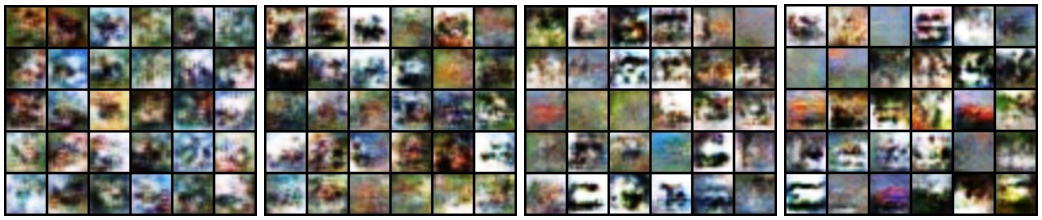


Figure. 32 WGAN x 16 generated images – z_dim 100 – CIFAR10

Figure. 33 WGAN x 32 generated images – z_dim 2 – CIFAR10

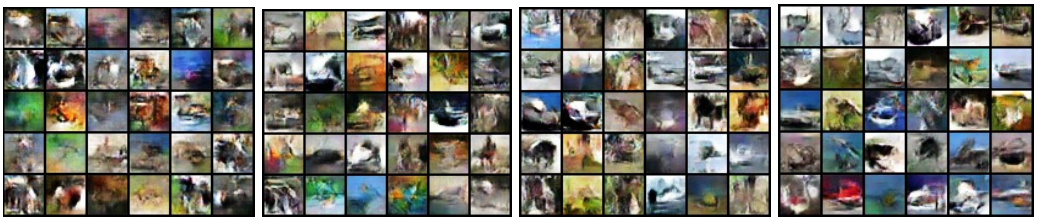


Figure. 34 WGAN x 32 generated images – z_dim 20 – CIFAR10

Figure. 35 WGAN x 32 generated images – z_dim 100 – CIFAR10

Figure. 36 WGAN x 64 generated images – z_dim 2 – CIFAR10

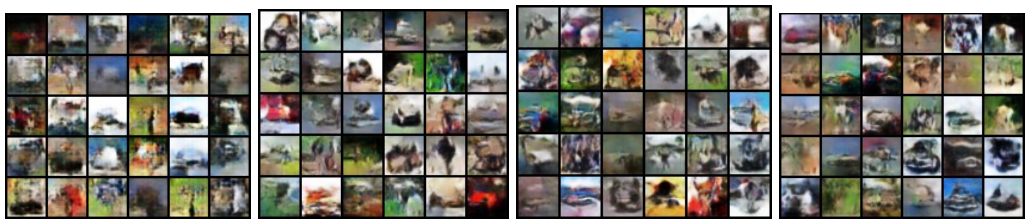


Figure. 37 WGAN x 64 generated images – z_dim 20 – CIFAR10

Figure. 38 WGAN x 64 generated images – z_dim 100 – CIFAR10

These images are similar to those generated by the GAN. That said, the quality of the images when using a WGAN is more consistent when using different hyperparameters. When using larger images, the GAN would not generate proper images. The WGAN does not suffer from this drawback. However, more training is required because the generator is only trained every 5 iterations, whereas in the implementation of the GAN, I chose to train the generator at every iteration. The images below were trained using a WGAN over double the number of epochs.

Epochs 40, 60, 80, 100 (5 critic iterations for each generator iteration):

Figure. 39 WGAN x 32 generated images – z_dim 100 – CIFAR10 – Epochs x 2

The quality of the images does seem to improve a little when training the model for longer periods of time. The results are very similar to the results found in figure 25. This leads to the conclusion that the WGAN generates comparable images to the GAN, however, with more stable training. It is also to generate better images with a wider range of image sizes and hyperparameters.

More Results are found in the respective training folder. The MNIST results are found at the end of the report.

End of report.

VAE Results (MNIST)

Next we look at the results obtained when using the VAE model. Epochs 1, 10, 15, 25:

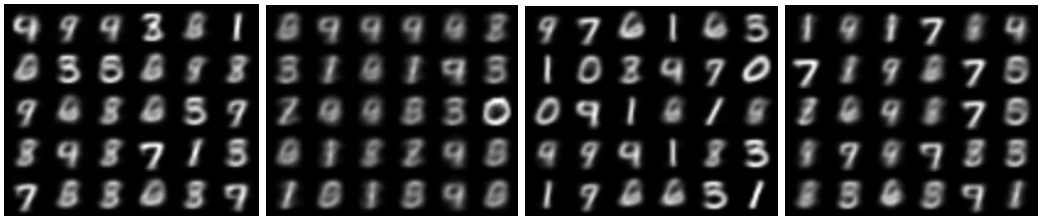


Figure. 40 VAE x 16 generated images – z_dim 2 – MNIST

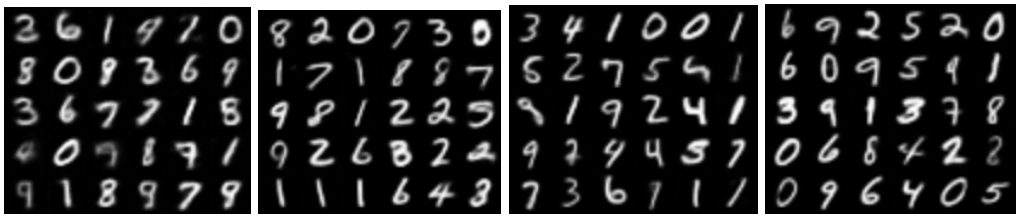


Figure. 41 VAE x 16 generated images – z_dim 20 – MNIST

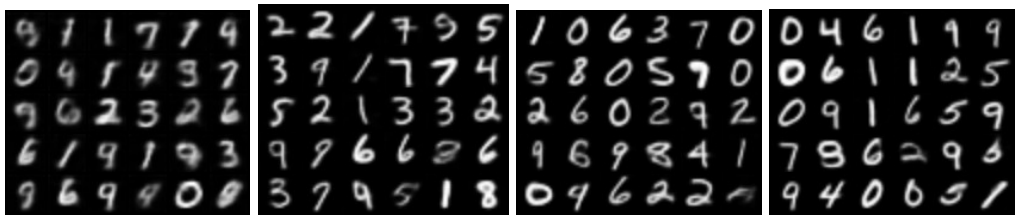


Figure. 42 VAE x 16 generated images – z_dim 100 – MNIST

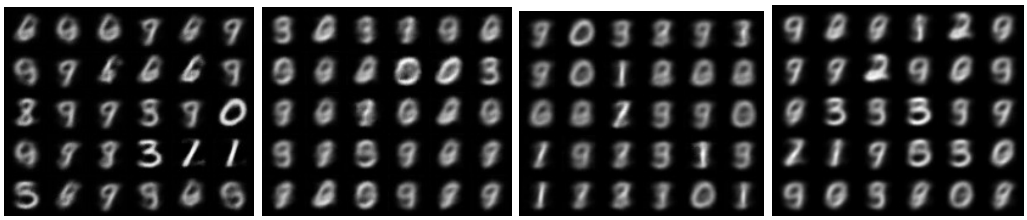


Figure. 43 VAE x 32 generated images – z_dim 2 – MNIST

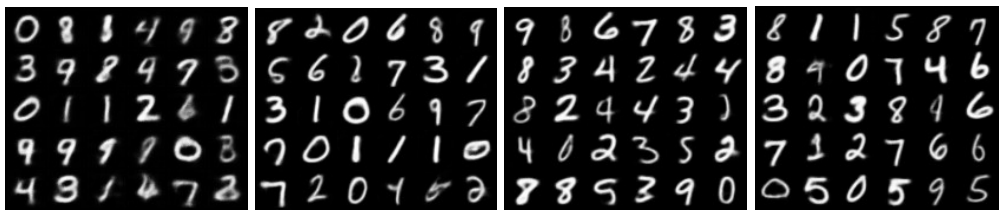


Figure. 44 VAE x 32 generated images – z_dim 20 – MNIST

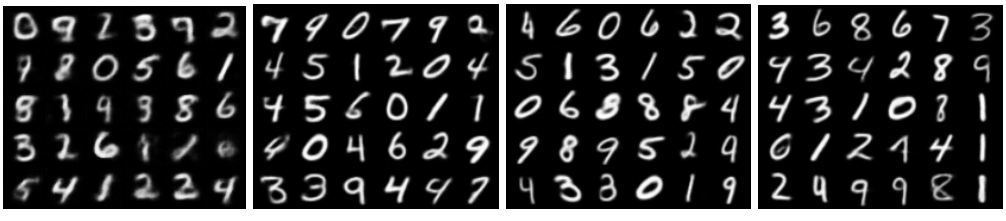


Figure. 45 VAE x 32 generated images – z_dim 100 – MNIST

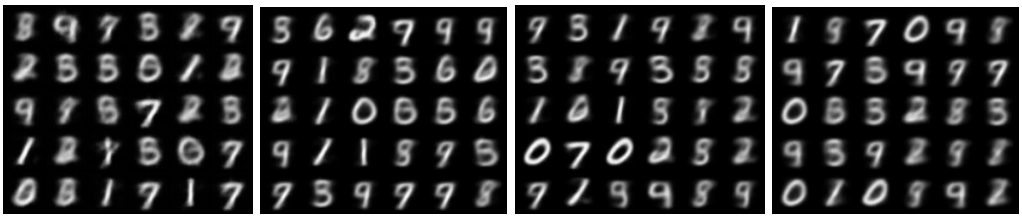


Figure. 46 VAE x 64 generated images – z_dim 2 – MNIST

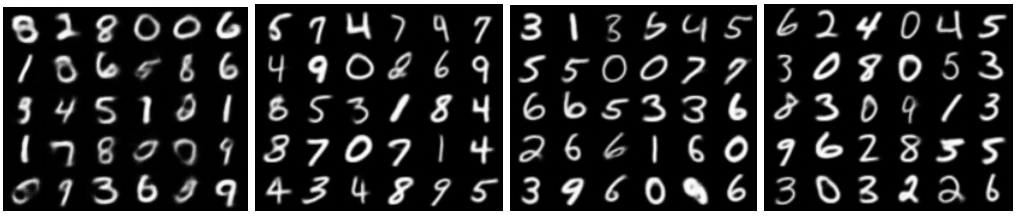


Figure. 47 VAE x 64 generated images – z_dim 20 – MNIST

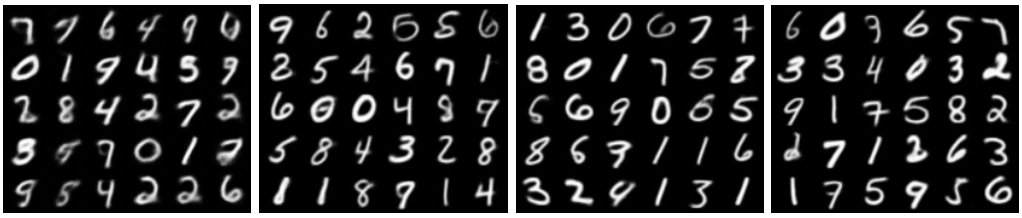


Figure. 48 VAE x 64 generated images – z_dim 100 – MNIST

More Results are found in the respective training folder.

GAN Results (MNIST)

Next we look at the results obtained when using the GAN model. Epochs 1, 10, 15, 25:

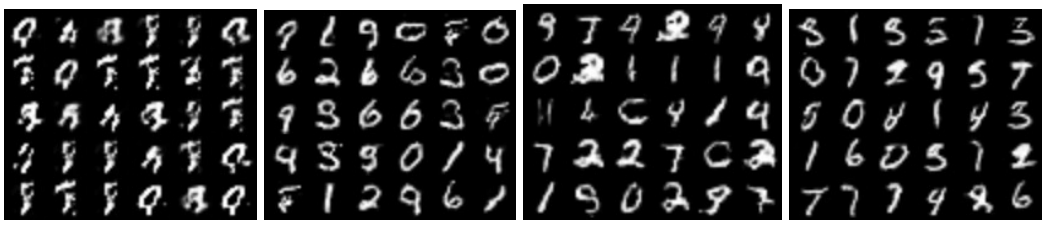


Figure. 49 GAN x 16 generated images – z_dim 2 – MNIST

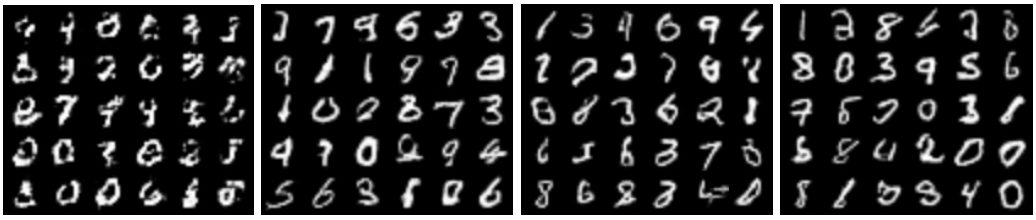


Figure. 50 GAN x 16 generated images – z_dim 20 – MNIST

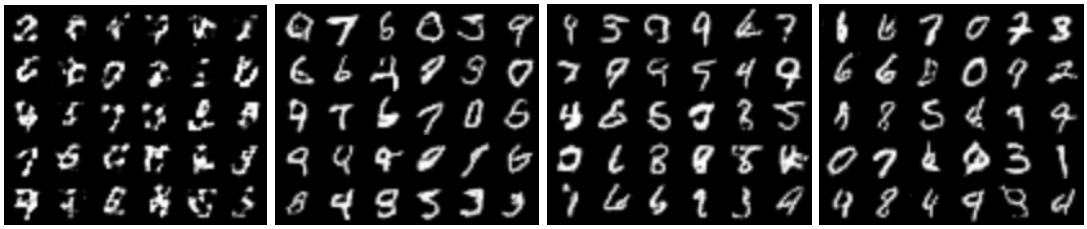


Figure. 51 GAN x 16 generated images – z_dim 100 – MNIST

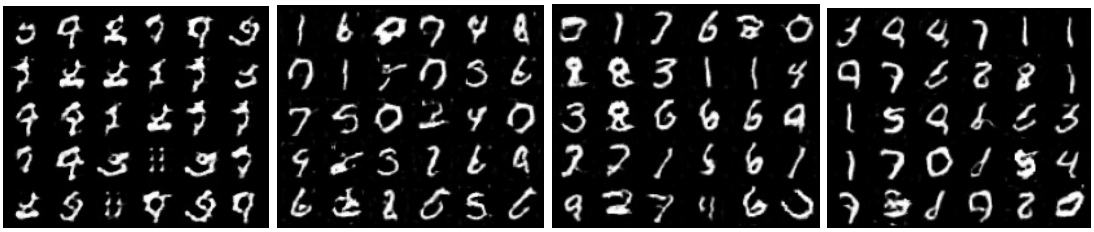


Figure. 52 GAN x 32 generated images – z_dim 2 – MNIST

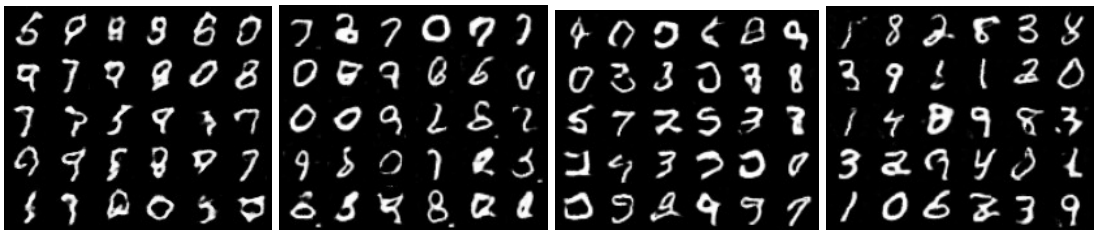


Figure. 53 GAN x 32 generated images – z_dim 20 – MNIST

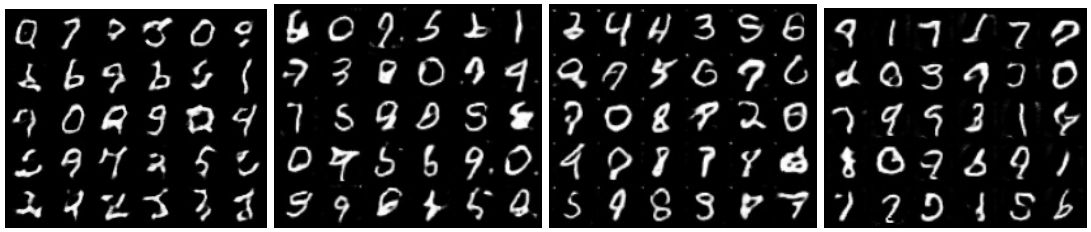


Figure. 54 GAN x 32 generated images – z_dim 100 – MNIST

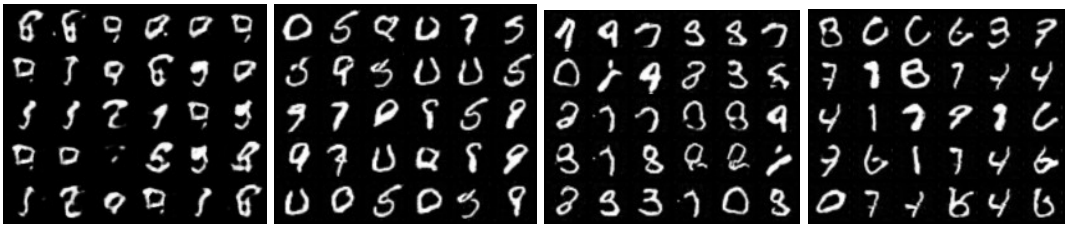


Figure. 55 GAN x 64 generated images – z_dim 2 – MNIST

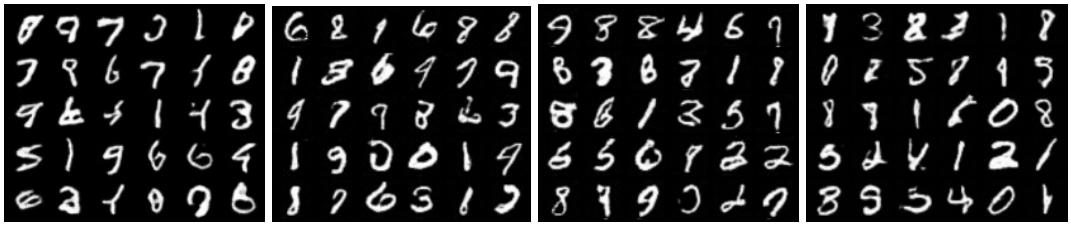


Figure. 56 GAN x 64 generated images – z_dim 20 – MNIST

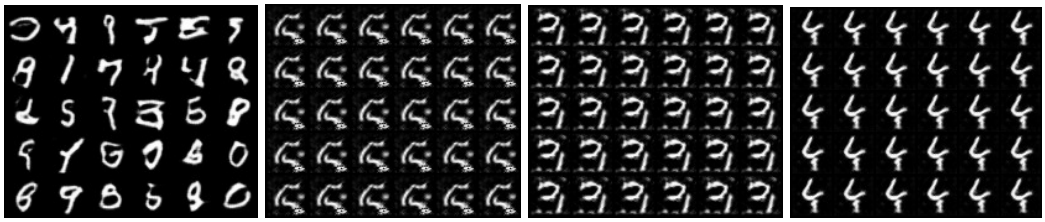


Figure. 57 GAN x 64 generated images – z_dim 100 – MNIST

More Results are found in the respective training folder.

WGAN Results (MNIST)

Next we look at the results obtained when using the WGAN model. Epochs 1, 10, 15, 25:

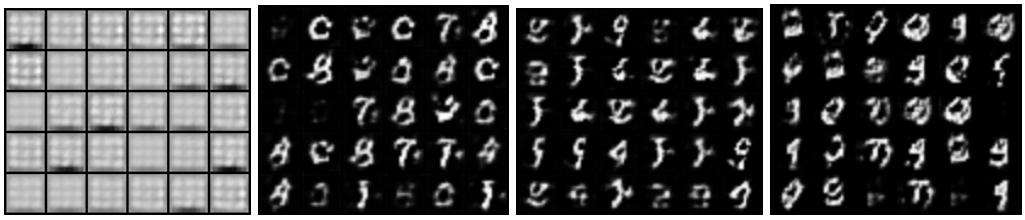


Figure. 58 WGAN x 16 generated images – z_dim 2 – MNIST

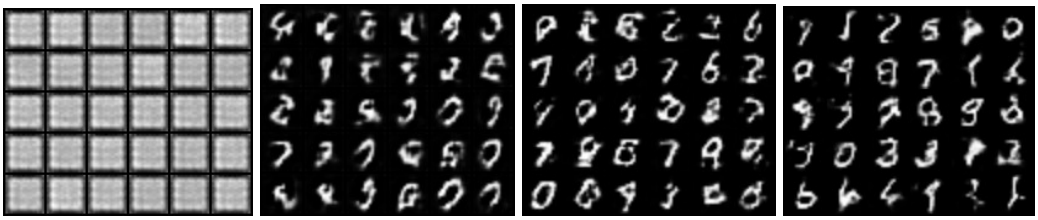


Figure. 59 WGAN x 16 generated images – z_dim 20 – MNIST

Figure. 60 WGAN x 16 generated images – z_dim 100 – MNIST

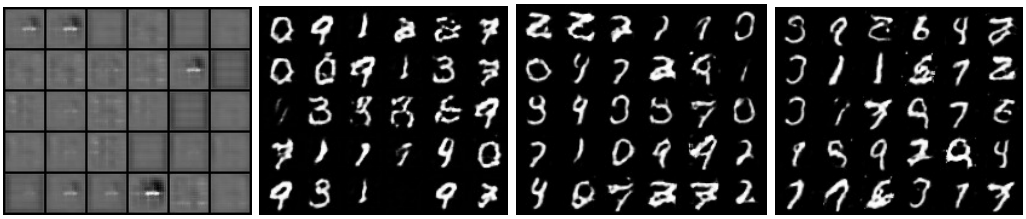


Figure. 61 WGAN x 32 generated images – z_dim 2 – MNIST

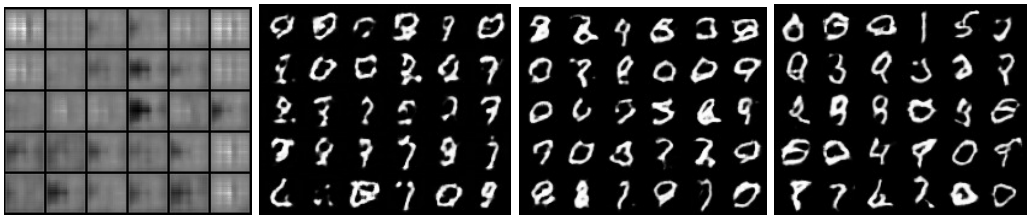


Figure. 62 WGAN x 32 generated images – z_dim 20 – MNIST

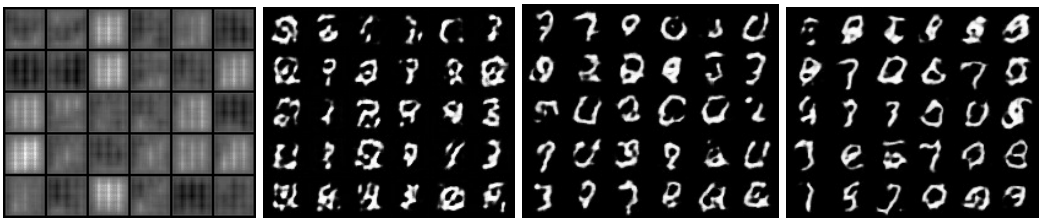


Figure. 63 WGAN x 32 generated images – z_dim 100 – MNIST

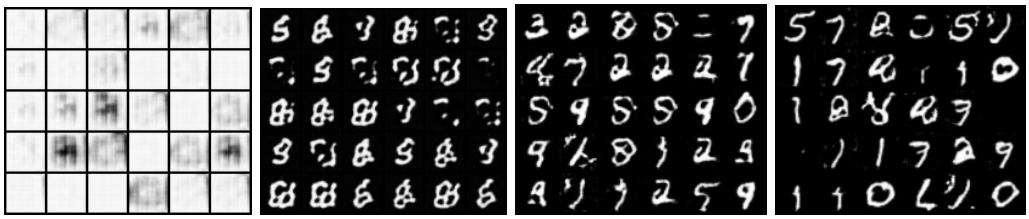


Figure. 64 WGAN x 64 generated images – z_dim 2 – MNIST

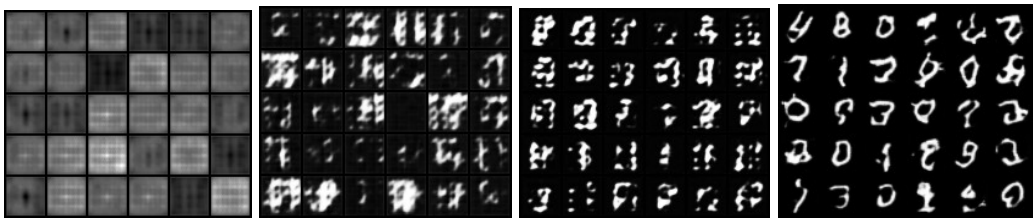


Figure. 65 WGAN x 64 generated images – z_dim 20 – MNIST

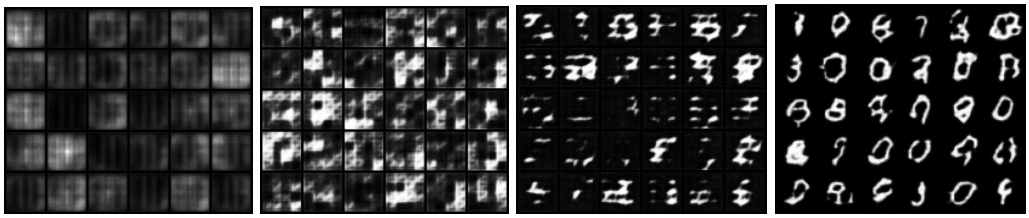


Figure. 66 WGAN x 64 generated images – z_dim 100 – MNIST

More Results are found in the respective training folder.