STATS M231A Homework 4

Laxman Dahal

Abstract

This homework aims at giving a hands-on experience with state-of-the-art algorithms on unsupervised learning such as Generative Adversarial Networks (GAN) and Variational AutoEncoders (VAE). For the purpose of training, MNIST data on hand-written digits of size 28×28 is used because of the limitation in computational resources. In total, three different networks were trained: 1) GAN with all fully connected linear layers, 2) GAN with Conv Network, and 3) VAE with fully connected linear layers. All three networks were able to generate reasonable results. It was found that GAN with Conv net required lower number of epochs to produced good results as compared to the one with linear layers.

1 Generative Adversarial Networks (GAN)

GAN is used in unsupervised learning where two networks (generator and discriminator) utilizes each other's loss to establish an equilibrium. The generative network generates candidates by learning to map from a latent space to a data distribution of interest, while the discriminative network evaluates them by distinguishing if the candidates produced by the generator comes from the true data distribution. The generative network's training objective is to increase the error rate of the discriminative network. Figure 1 illustrates a graphical overview of the GAN.

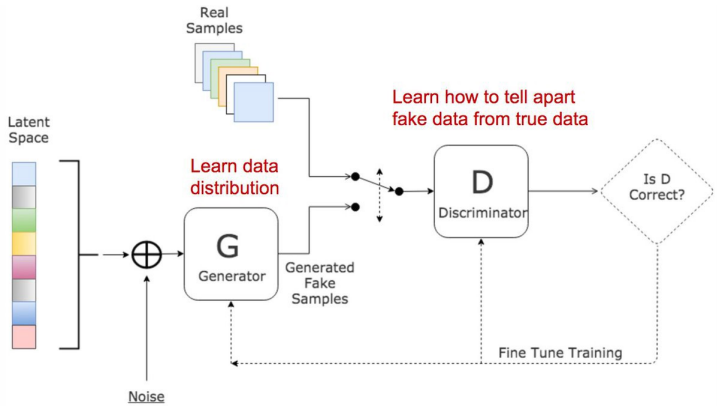


Figure 1: Graphical overview of the GAN. The generator samples from latent space with noise which is evaluated by the discriminator.

1.1 Algorithm

Figure 2 shows the algorithm implemented in GAN. Ideally, given enough training data and computational capacity, to get a good estimate of p_{data}, the algorithm should converge. The generator G implicitly defines a probability distribution p_g as the distribution of the samples $G(z)$ obtained when $z \sim p_z$.

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for
  • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  • Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Figure 2: Algorithm of the GAN. Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter.

1.2 Equations

We can train the pair of (G, D) by an adversarial, zero-sum game. Specifically, let $G(h) = g_\theta(h)$ be a generator. Let

$$V(D,G) = \mathbb{E}_{p_{\text{data}}}[\log D(X)] + \mathbb{E}_{h \sim p(h)}[\log(1 - D(G(h))],$$

where $\mathbb{E}_{p_{\text{data}}}$ can be approximated by averaging over the observed examples, and $\mathbb{E}_h$ can be approximated by Monte Carlo average over the faked examples generated by the model.

Thus, we learn D and G by $\min_G \max_D V(D,G)$. $V(D,G)$ is the log-likelihood for D, i.e., the log-probability of the real and faked examples. However, $V(D,G)$ is not a very convincing objective for G. In practice, the training of G is usually modified into maximizing $\mathbb{E}_{h \sim p(h)}[\log D(G(h))]$ to avoid the vanishing gradient problem.

For a given θ, let p_θ be the distribution of $g_\theta(h)$ with $h \sim p(h)$. Assuming a perfect discriminator according to the Bayes rule $D(x) = p_{\text{data}}(x)/(p_{\text{data}}(x) + p_\theta(x))$ (assuming equal numbers of real and fake examples). Then θ minimizes Jensen-Shannon

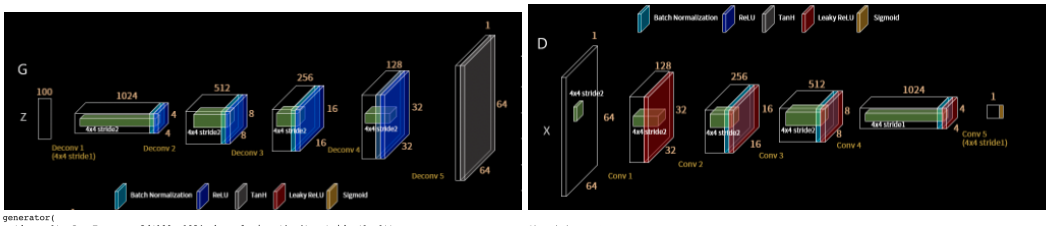
$$\text{JSD}(p_{\text{data}} | p_\theta) = \text{KL}(p_\theta | p_{\text{mix}}) + \text{KL}(p_{\text{data}} | p_{\text{mix}}),$$

where $p_{\text{mix}} = (p_{\text{data}} + p_\theta)/2$. As a result, GAN has mode collapsing behavior.

1.3 DCGAN

GAN with Conv net.

1.3.1 Network Structure




```
generator(
  (deconv1): ConvTranspose2d(100, 1024, kernel_size=(4, 4), stride=(1, 1))
  (deconv1_bn): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (deconv2): ConvTranspose2d(1024, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  (deconv2_bn): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (deconv3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  (deconv3_bn): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (deconv4): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  (deconv4_bn): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (deconv5): ConvTranspose2d(128, 1, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
)
```

```
discriminator(
  (conv1): Conv2d(1, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  (conv2): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  (conv2_bn): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  (conv3_bn): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv4): Conv2d(512, 1024, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  (conv4_bn): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv5): Conv2d(1024, 1, kernel_size=(4, 4), stride=(1, 1))
)
```

Figure 3: Network structure for GAN with Conv layers

1.3.2 Loss Curve

Figure 4 shows the training loss for the generator and discriminator and the sum of the two for the GAN with convolutional layers. As expected, the D loss increases with the number of epochs while the G los decreases. It implies that they if a large number of epochs is used, they the two losses will be at equilibrium.

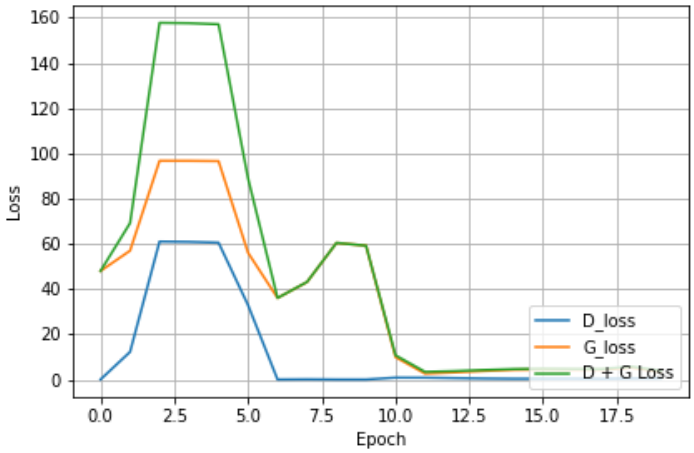


Figure 4: Training loss curve for GAN with Conv layers.

1.3.3 Synthesis

Figure 5 shows and example of synthesized images using a fixed noise. Please note that the Figure 5 is for the 19^{th} epoch. For comparison between each epoch, please refer to the attached notebook.

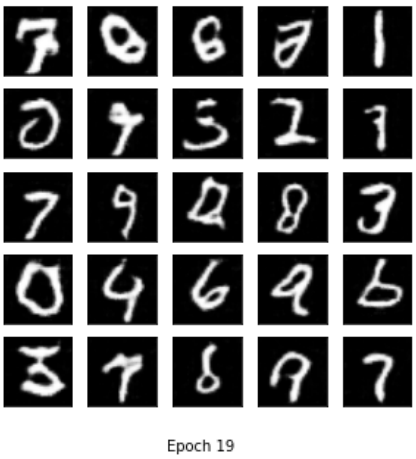
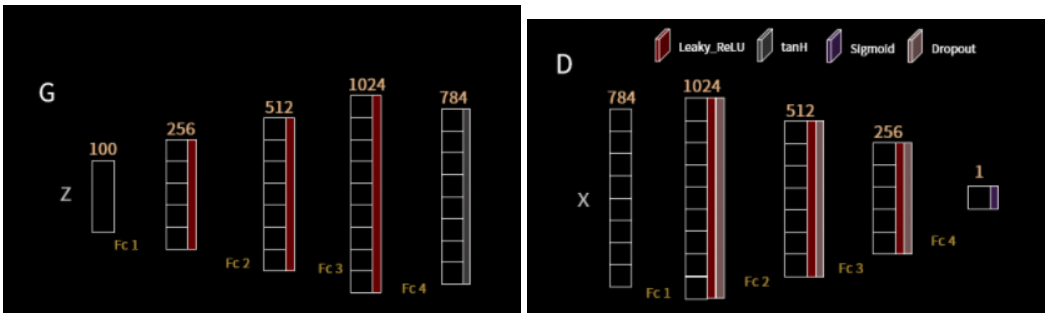


Figure 5: Synthesis result using a fixed sampled latent vector, z.

1.4 GAN-Linear

GAN with all fully connected linear layers.

1.4.1 Network Structure




```
Generator(
  (fc1): Linear(in_features=100, out_features=256, bias=True)
  (fc2): Linear(in_features=256, out_features=512, bias=True)
  (fc3): Linear(in_features=512, out_features=1024, bias=True)
  (fc4): Linear(in_features=1024, out_features=784, bias=True)
)
```

```
Discriminator(
  (fc1): Linear(in_features=784, out_features=1024, bias=True)
  (fc2): Linear(in_features=1024, out_features=512, bias=True)
  (fc3): Linear(in_features=512, out_features=256, bias=True)
  (fc4): Linear(in_features=256, out_features=1, bias=True)
)
```

Figure 6: Two fully connected layers of GAN

1.4.2 Loss Curve

Figure 7 shows the training loss for the generator, discriminator and the sum of two for the GAN with all fully connected linear layers. As expected, the D loss increases with the number of epochs while the G los decreases. It implies that they if a large number of epochs is used, they the two losses will be at equilibrium.

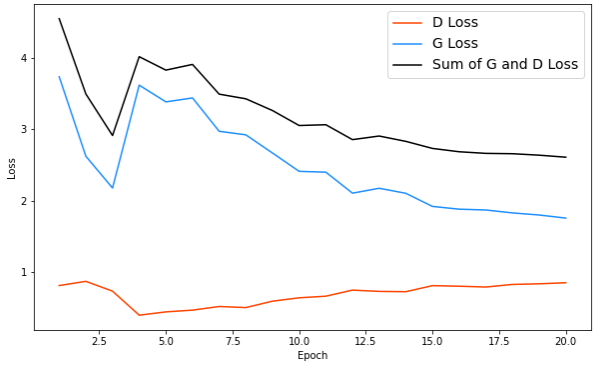


Figure 7: Training loss curve for GAN with only linear layers.

1.4.3 Synthesis

Figure 8 shows and example of synthesized images using a fixed noise. Please note that the Figure 8 is for the 10^{th} epoch. For comparison between each epoch, please refer to the attached notebook.

1.4.4 Grid-in-Latent

Figure 9 shows the 10×10 grid from the latent space.

Figure 8: Synthesis result using a fixed sampled latent vector, z.

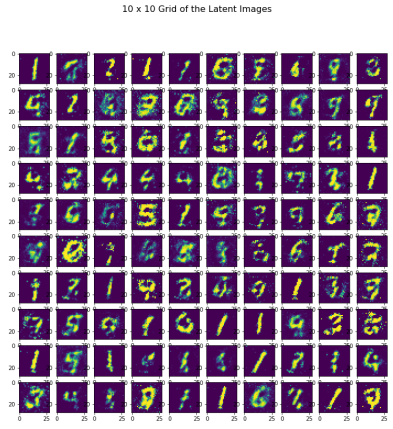


Figure 9: Grid-in-latent for the linear GAN

2 Variational AutoEncoders (VAE)

Variational autoencoder (VAE) was primarily designed for unsupervised learning where the idea of representation is utilized. It is a special type of neural network with a bottleneck layer (also referred to as latent representation) that is used for dimensionality reduction. In other words, the input information is compressed into a latent distribution (encoding) which is then used to reconstruct the input (decoding). **??** shows the architecture of the VAE. The input data is sampled from a parametrized distribution (the prior), and the encoder and decoder are trained jointly such that the output minimizes the reconstruction error in the sense of the Kullback-Leibler divergence (KL divergence) between the parametric posterior and the true posterior.

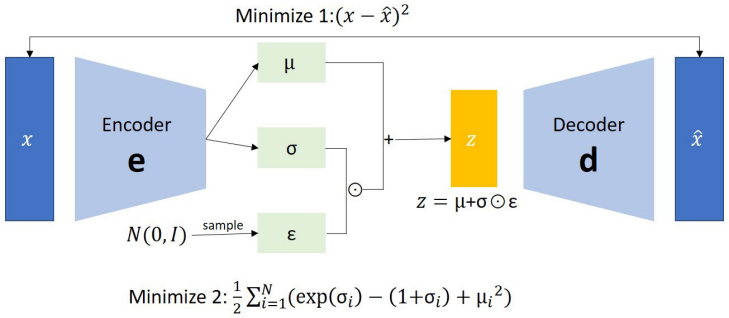


Figure 10: The architecture of a VAE. The model receives input x which is compressed by encoder into a latent space. The samples from the latent space is provided as an input to the decoder. The decoder aims to produce x' as close to x as possible.

2.1 The encoder

The encoder consists of two separated fully-connected layers. The layers take in the input data and output two vectors in the dimension of intended latent space. The two output vectors represent mean and the standard deviation respectively. The encoder model is $[h|x] \sim q_\phi(h|x) \sim \mathrm{N}(\mu_\phi(x), V_\phi(x))$, where V is a diagonal matrix. Together with $p_{\text{data}}(x)$, we have a joint distribution $q_\phi(h,x) = p_{\text{data}}(x)q_\phi(h|x)$.

2.2 Reparameterization

With mean and variance computed, the distribution is randomly sampled. This random sample is used as an input for the decoding.

2.3 The decoder

The decoder takes samples from the latent vector and utilized two fully connected linear layer and sigmoidal function to reconstruct the image based on the given latent representation. The decoder model is $h \sim p(h) \sim \mathrm{N}(0, I_d)$, and $[x|h] \sim p_\theta(x|h) \sim \mathrm{N}(g_\theta(h), \sigma^2 I)$. It defines a joint distribution $p_\theta(h,x) = p(h)p_\theta(x|h)$.

2.4 Loss Function

In VAE, there are two loss functions:

2.4.1 Binary Cross Entropy

BCE calculate the pixel-to-pixel difference of the reconstructed image with the original image to maximize the similarity of reconstruction. It is given by:

$$\sum_{i=1}^{n} x_i' log x_i + (1 - x_i) log(1 - x_i)$$

where x_i and x_i' denote the original and reconstructed image pixels for n total pixels, respectively.

2.4.2 KL-Divergence Loss

KL divergence measures the similarity of two distributions. In this case, we assume the distribution to be normal, and hence the loss is designed as the follows

$$\sum_{i=1}^{m} = \sigma_i^2 + \mu_i^2 + log(\sigma_i) - 1$$

which is calculated via our predicted mean and sigma of every value in the latent vector (size m).

2.5 Algorithm

Figure 11 shows the algorithm of VAE. More specifically, it shows the computation of unbiased estimate of single datapoint ELBO for example VAE with a full covariance Gaussian inference model. L_{mask} is a masking matrix with zeros on and above the diagonal, and ones below the diagonal.

Data:
$\mathbf{x}$: a datapoint, and optionally other conditioning information
$\boldsymbol{\epsilon}$: a random sample from $p(\boldsymbol{\epsilon}) = \mathcal{N}(0, \mathbf{I})$
$\boldsymbol{\theta}$: Generative model parameters
$\boldsymbol{\phi}$: Inference model parameters
$q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$: Inference model
$p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})$: Generative model
Result:
$\tilde{\mathcal{L}}$: unbiased estimate of the single-datapoint ELBO $\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x})$

$(\boldsymbol{\mu}, \log \boldsymbol{\sigma}, \mathbf{L}') \leftarrow \text{EncoderNeuralNet}_{\boldsymbol{\phi}}(\mathbf{x})$
$\mathbf{L} \leftarrow \mathbf{L}_{mask} \odot \mathbf{L}' + \text{diag}(\boldsymbol{\sigma})$
$\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$
$\mathbf{z} \leftarrow \mathbf{L}\boldsymbol{\epsilon} + \boldsymbol{\mu}$
$\tilde{\mathcal{L}}_{\text{logqz}} \leftarrow -\sum_i (\frac{1}{2}(\epsilon_i^2 + \log(2\pi) + \log \sigma_i))_i$ ▷ $= q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$
$\tilde{\mathcal{L}}_{\text{logpz}} \leftarrow -\sum_i (\frac{1}{2}(z_i^2 + \log(2\pi)))$ ▷ $= p_{\boldsymbol{\theta}}(\mathbf{z})$
$\mathbf{p} \leftarrow \text{DecoderNeuralNet}_{\boldsymbol{\theta}}(\mathbf{z})$
$\tilde{\mathcal{L}}_{\text{logpx}} \leftarrow \sum_i (x_i \log p_i + (1 - x_i) \log(1 - p_i))$ ▷ $= p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$
$\tilde{\mathcal{L}} = \tilde{\mathcal{L}}_{\text{logpx}} + \tilde{\mathcal{L}}_{\text{logpz}} - \tilde{\mathcal{L}}_{\text{logqz}}$

Figure 11: Algorithm of VAE

2.6 Network Structure

Figure 12 shows the network structure implemented in VAE for the MNIST dataset. As shown in the figure, all linear layers were used for both the encoder and decoder portion.

```
VAE(
  (fc1): Linear(in_features=784, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=256, bias=True)
  (fc31): Linear(in_features=256, out_features=2, bias=True)
  (fc32): Linear(in_features=256, out_features=2, bias=True)
  (fc4): Linear(in_features=2, out_features=256, bias=True)
  (fc5): Linear(in_features=256, out_features=512, bias=True)
  (fc6): Linear(in_features=512, out_features=784, bias=True)
)
```

Figure 12: Network structure implement in VAE for the MNIST dataset

2.7 Loss Curve

Figure 13 shows the training and testing loss of the VAE. As expected, both the training and testing loss decrease as the number of epochs increase.

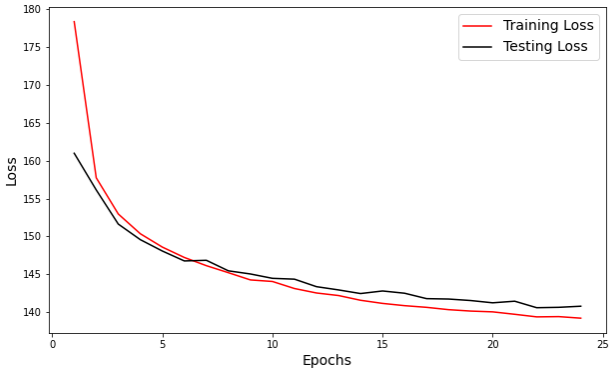


Figure 13: Training and testing loss curve VAE

2.8 Reconstruction

Figure 14 shows the 5×5 grid of randomly sampled test images and the reconstructed images.

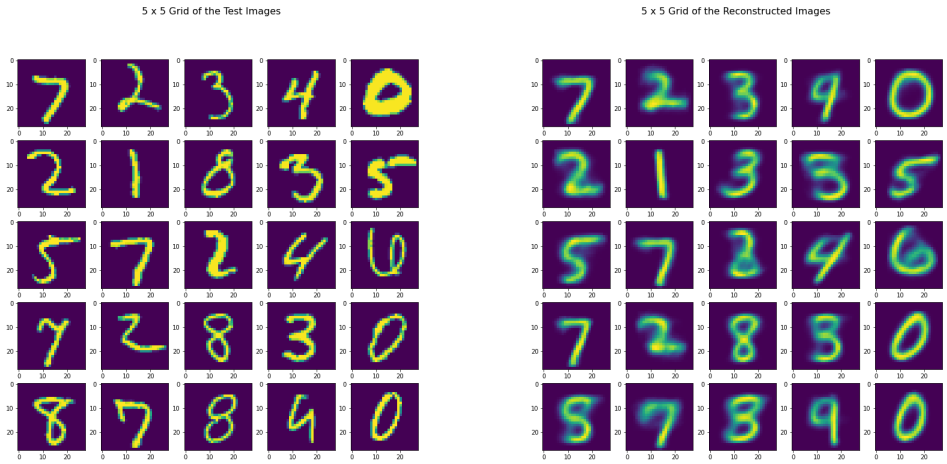


Figure 14: 5×5 grid of randomly sampled test images (left) and the reconstructed images using VAE (right)

2.9 Synthesis

Figure 15 shows and example of synthesized images using a fixed noise. Please note that the Figure 15 is for the 10^{th} epoch. For comparison between each epoch, please refer to the attached notebook

How to run There are three jupyter notebook attached with this written report. The first notebook titled "hw4_VAE.ipynb" is the notebook for VAE which also include the synthesis output on 5×5 grid for each epoch. The second notebook titled "hw4_GAN_linear.ipynb" is used for the GAN with all linear layers and includes the synthesis output for each epoch. The third notebook titled

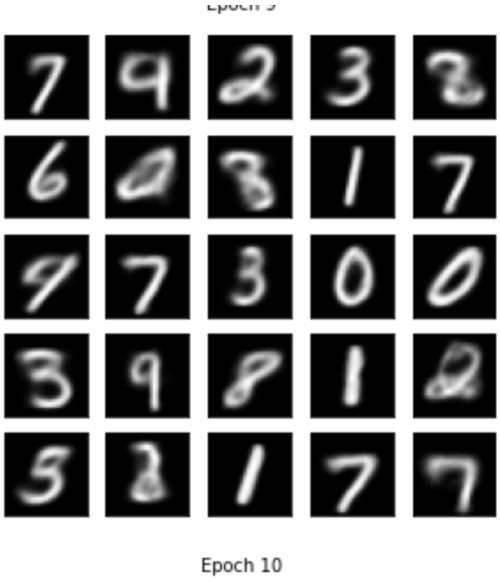


Figure 15: Synthesis result using a fixed sampled latent vector, z.

"hw4_DCGAN.ipynb" is used for the GAN with Conv net layers and includes the synthesis output for each epoch. All the notebooks were executed in the Google Colab Pro using their GPU capability.

2.10 Conclusion

In total, three different networks were trained: 1) GAN with all fully connected linear layers, 2) GAN with Conv Network, and 3) VAE with fully connected linear layers. All three networks were able to generate reasonable results. It was found that GAN with Conv net required lower number of epochs to produced good results as compared to the one with linear layers.