Deep Learning

Vrije Universiteit Amsterdam

Assignment 5

Shwetambari Tiwari *(2643671)*

Contents

Part 1: Descriptions of a VAE and a GAN

VAE

The Varational AutoEncoder has the purpose of storing generalizing information on data in a lower dimensional latent space, from which similar high-dimensional data could be generated. The basic architecture of a VAE is schematically visualized in figure 1.

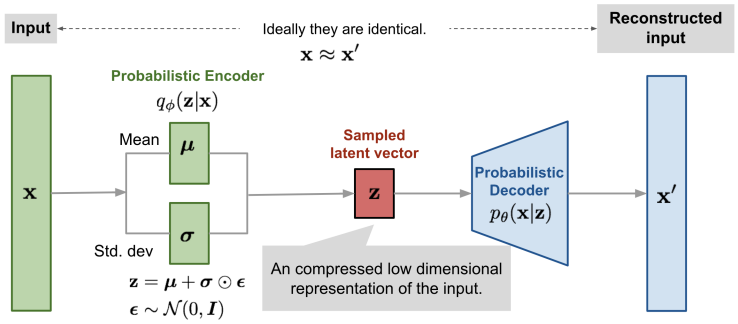


Figure 1: The basic architecture of a VAE. Source

The formal objective function that we want to minimize in learning is the log-likelihood. Within the framework of the VAE set-up, it is defined as provided in the lecture slides (Jakub M. Tomczak 2020):

$$\mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\log p_\theta(\mathbf{x}|\mathbf{z})\right] - \mathrm{KL}\left(q_\phi(\mathbf{z}|\mathbf{x})\|p_\lambda(\mathbf{z})\right)$$

To be more precise, this expression is a lower bound (ELBO) on the log likelihood, which can be derived using Jensen's inequality. The breakdown of this objective function becomes apparent if we analyze the separate components of the VAE in detail. The two essential elements in the VAE machinery are the **encoder** and the **decoder**. The encoder has to 'translate' the patterns behind the input data into Gaussian distribution parameters μ and σ that in their turn define the data in the latent space. The target of the encoder is to learn these parameters. To put it more formally, we want to learn the conditional distribution of our latent variable $\mathbf{z}$ given the data $\mathbf{x}$, or: $p(z|x)$. Since it is not possible to learn this distribution explicitly it is replaced with the *variational posterior* $q_\phi(\mathbf{z} \mid \mathbf{x})$. The encoder thus has to learn its defining parameters μ and σ, having the form of standard normal prior $p_\lambda(\mathbf{z})$. The degree of divergence between the approximate function $q_\phi(\mathbf{z}|\mathbf{x})$ with, in our case, the standard normal is forced down by application of the Kullback-Leibler divergence function KL (Wikipedia contributors 2020) in order to not over-complicate the model. The decoder has the reverse task: it wants to learn the parameters for the distribution of the data $\mathbf{x}$ given the latent representation $\mathbf{z}$: $p_\theta(\mathbf{x}|\mathbf{z})$. The goal of the decoder is thus to generate data similar to $\mathbf{x}$, getting samples from the just defined latent distribution as an input. Hence we can now dissect the objective function in its two essential parts:

- **Reconstruction Error:** $\mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})]$ where $\log p_\theta(\mathbf{x}|\mathbf{z})$ represents the **decoder**. It reads: the likelihood of the generated sample $\mathbf{x}$ given that the input for the decoder is the latent variable $\mathbf{z}$. This latent variable $\mathbf{z}$ was, in its turn, sampled from the variational posterior $q_\phi(\mathbf{z}|\mathbf{x})$, of which the parameters are learned by the **encoder**.
- **KL Regularization** $\mathrm{KL}\left(q_\phi(\mathbf{z}|\mathbf{x})\|p_\lambda(\mathbf{z})\right)$ where $q_\phi(\mathbf{z}|\mathbf{x})$ represents the **encoder** and $p_\lambda(\mathbf{z})$ the marginal prior which is the standard normal. The term can be interpreted as applying a penalty proportional to the dissimilarity of $q_\phi(\mathbf{z}|\mathbf{x})$ and $p_\lambda(\mathbf{z})$). With this term we balance out the fact that want to approximate $p(\mathbf{z}|\mathbf{x})$ but also want to keep $q_\phi(\mathbf{z}|\mathbf{x})$ simple enough and close to the standard normal.

A furhter important note on the VAE model should be made on the learning algorithm. Since the subcomponents of the VAE that define the parameters for the distributions are neural networks, we would like to train the model using backpropagation. The backpropation algorithm propagates the chain of gradients from the loss towards the parameters; this mechanism operates in a deterministic space. However, as can be observed in figure 1, we encounter stochastic nodes along the computation graph which would block the flow of gradients. This issue is resolved by applying the *reparametrization trick*, which provides a deterministic path on the distribution parameters and adds a random noise to it via a side path. This principle is clearly visualized in figure 2.

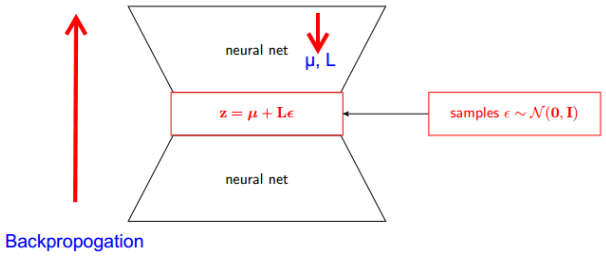


Figure 2: The reparametrization trick within the computation graph. Source

GAN

The common objective function for a GAN is the Adverserial loss, which is defined in the lecture slides as follows (Jakub M. Tomczak 2020):

$$\min_G \max_D \mathbb{E}_{\mathbf{x}\sim p_{\text{real}}}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))]$$

The breakdown and meaning of this objective funtion will become clear when the working principle of a GAN is dissected into its different components. The core architecture concists of the **generator** and the **discriminator**, which are the two main elements in any GAN. The two counteract eachother in terms of their individual learning objective and it is this

dynamic that drives the learning process towards a solid generative model. The set-up is visualized in figure 3.

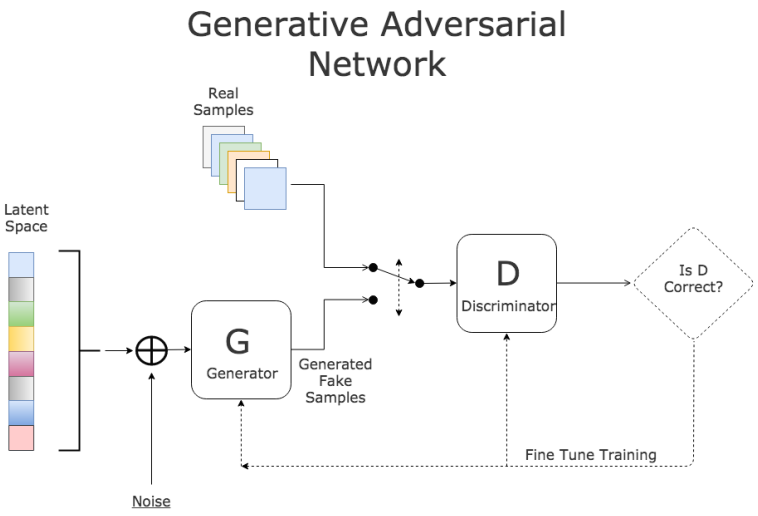


Figure 3: The basic architecture of a GAN. Source

The discriminator is being fed with data originating from either a sample on the real data or a 'fake' artificially generated from the generator. The discriminator than has to classify whether a given input is real or generated. Hence, its learning objective is to do this as good as possible, or formally to maximize $\log D(\ldots)$. The generator on the other hand samples from the learned latent representation of the data, contained in $\mathbf{z}$ with added noise (again, the reparametrization trick). The learning objective of the generator is to challenge the discriminator as much as possible. In the ideal situation, it would generate samples that are indistinguishable from the real data, for the discriminator. The formal objective function that was presented earlier, the adversarial loss, can now be interpreted in this context. The $\mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))]$ term represents the learning objective of the generator; we want to minimize this term with respect to it. The total objective function $\min_G \max_D \mathbb{E}_{\mathbf{x}\sim p_{\text{real}}}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))]$ than represents the learning objective of the discriminator; we want to perform as good as possible on the real and generated input, which in turn forces the generator to perform as good as possible, hence the $\min_G \max_D$ objective. This mechanism explains the 'adversarial' part in 'adversarial loss'; the loss on the discriminator is minimized but the counteracting adverse objective of the generator is embedded within it.

Part 2: Implementation

VAE

Two VAE models were trained; one on the `MNIST` dataset and one on the `Imaginette` dataset. For the first model, the encoder and decoder consisted of linear layers combined with ReLU activation functions. Furthermore, the Adam optimzer was used in combination with the Binary Cross Entropy (BCE) loss. The learning rate was set to 0.001 and a batchsize of 64 was used. See 1 in the appendix or the code attached to the submission for the exact set-up. The second model was designed to contain linear and convolutional layers, again with ReLU activation (see 2). Again the Adam optimizer and BCE loss were chosen; the learning rate was 0.001 and the batchsize 32.

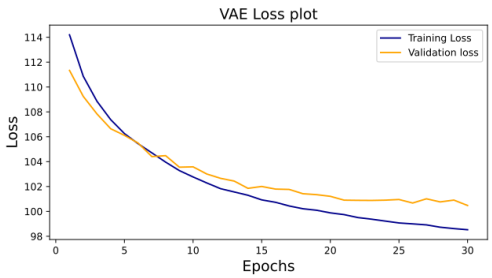
GAN

Two GAN models were trained on the datasets. The two models had similar base architectures. The discriminator used linear layers combined with leaky ReLU and Sigmoid aciti-vation. The generator concisted of linear layers together with leaky ReLU and Tanh. The complete architecture is provided in 3.

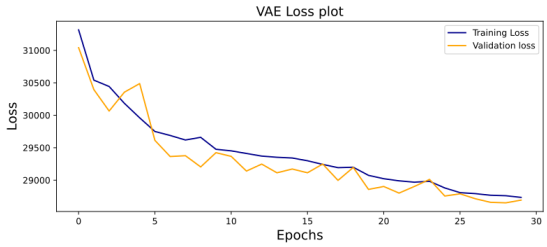
Part 3: Analysis

VAE

A VAE model with an architecture as was briefly described in the previous section was trained on the `MNIST` and `IMAGINETTE` datasets. Learning curves were recorded during training of the model and are provided in figure 8. As can be seen, learning converges in in both cases. As expected, loss on the training set keeps decreasing after every epoch meaning that the learning grows towards maximal memorization of the training set if we would go on longer. The learning curve on the validation set plateaus to a stable level after 30 epochs on both datasets. A notable difference can be seen in the stability of the validation learning curve, which seems to be less noisy on for `MNIST` case. The calculated FID score for the test set of the Imaginette dataset is 330.25, that is relatively high as thinking that the FID score of two identical dataset should be zero. The implementation of the inception model and fid score is taken from mseitzer 2020.



(a) Training on MNIST.



(b) Training on Imaginette.

Figure 4: Learning curves of the models during training on both the train and validation sets.

To qualitatively judge the performance of the model, some samples are generated and presented in figure 5. The model output seems to mimic the real data very well. Almost every reconstructed digit is perfectly identifiable as a digit. Only in a few instances some confounding between different underlying patterns produces somewhat fuzzy results (see reconstruction [row 4, col 3] and [row 1, col 8] for example). As can be seen in figure 6, the VAE generated images from the `Imaginette` model were of less quality. The generated images mimic the true data paterns much less accurate compared to the digit task.

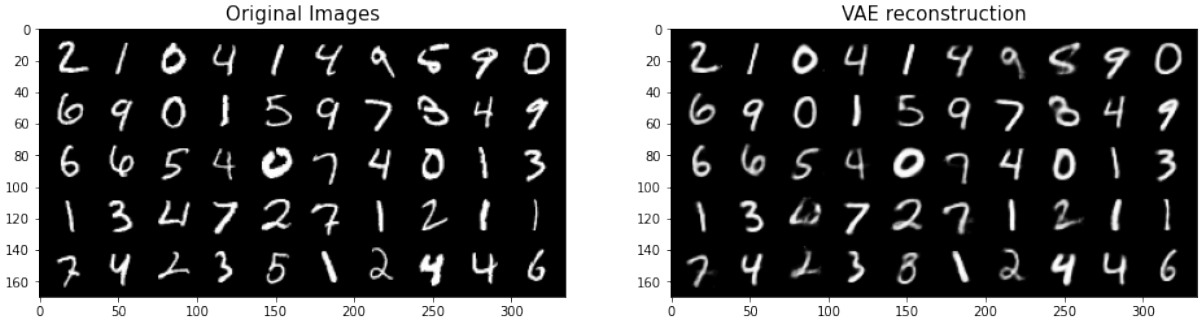


Figure 5: Samples of the VAE model trained in the MNIST data.

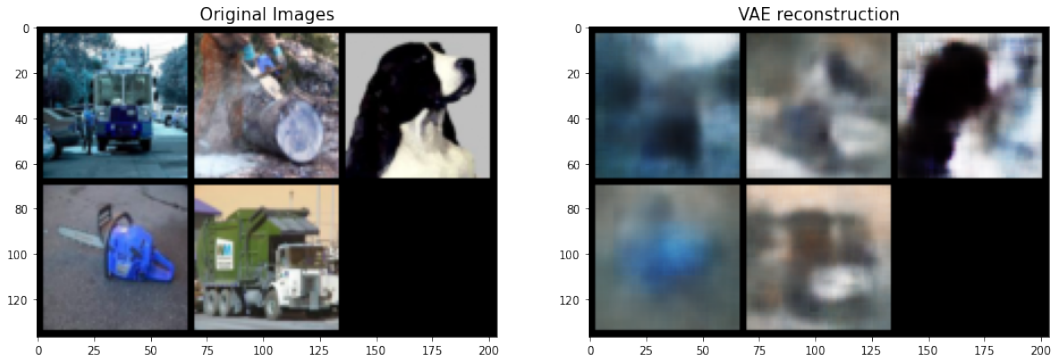


Figure 6: Samples of the VAE model trained in the Imaginette data.

To analyze the latent space, two point were sampled from it and linear interpolation was performed. From the linear interpolation, 10 points were sampled to generate the images as shown in figure 7.

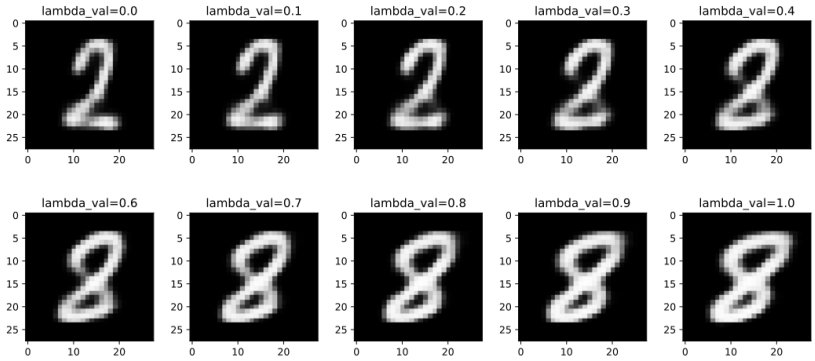


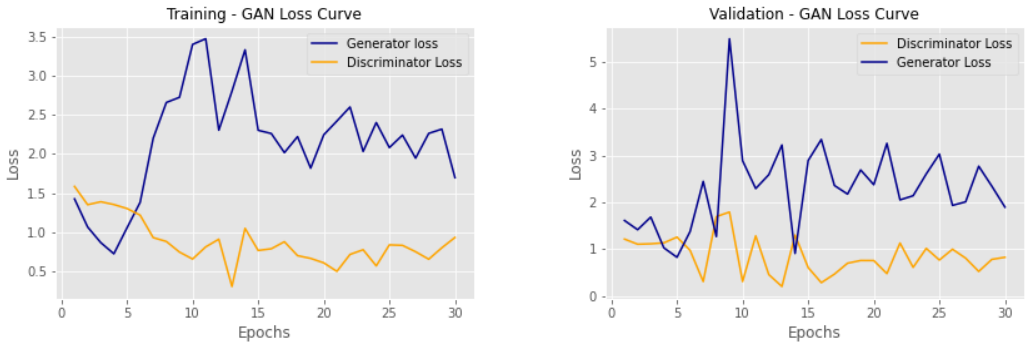
Figure 7: Samples generated from a linear interpolation between two points in the latent space (digits 2 and 8).

The figure clearly illustrates that we are making a transition between to points in the latent space that correspond to different digits. Generating images from the linear interpolation between the two points clearly shows the transition because we start at a point resembling a 2 and move towards an image resembling an 8. From the samples, we can conclude that the latent space is properly defined as the interpolation translates to somewhat sensible output in image space. Hence the topology of the latent space is not random or meaningless, based on this information.

GAN

The loss curve for GAN's on training and validation set for MNIST dataset is shown below in Figure 8. The training was done for 30 epochs with Adam optimizer and the number of

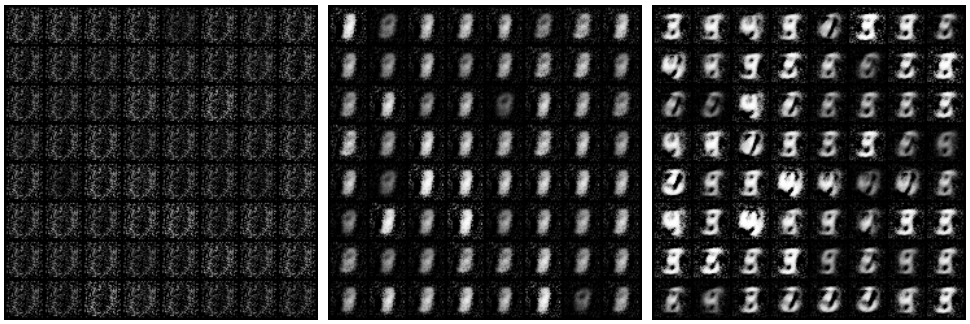
training parameters is comparable to the VAE architecture implemented. The performance of generator is discriminator is similar on training and validation set. The performance of GAN is expected to stabilize after 200 epochs.



(a) GAN loss curve - Training Set (b) GAN loss curve - Validation Set

Figure 8: Performance of GAN on MNIST dataset

Samples generated for MNIST during the training procedure is shown in Figure 9 and samples generated for Imagenette during training in Figure 11. During training on Imagenette we calculated also the FID scores that was around 230. The implementation of the inception model and fid score is taken from mseitzer 2020.



(a) Epoch 0 (b) Epoch 15 (c) Epoch 30

Figure 9: Performance of GAN on MNIST dataset

Latent space interpolation was implemented for GAN's, with the help of BKHMSI 2018. This was visualized with a gif image as well. Results obtained after training over 5 epochs is shown in the Figure10

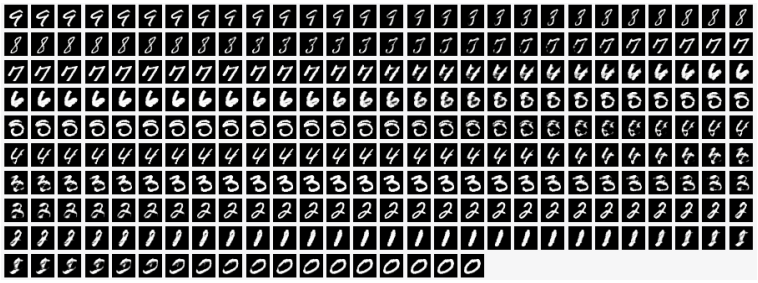
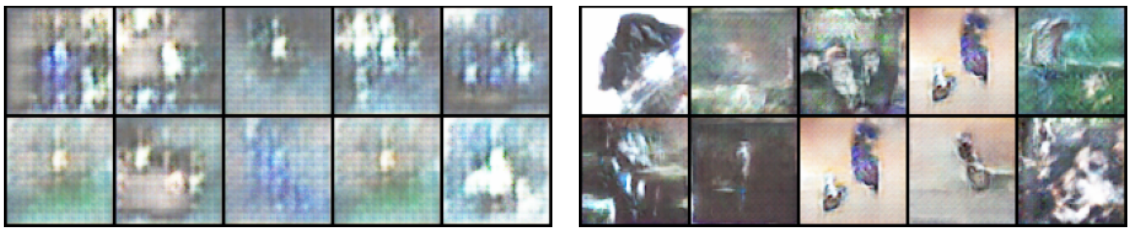


Figure 10: MNIST latent space interpolation



(a) Epoch 20

(b) Epoch 45

Figure 11: Performance of GAN on Imagenette dataset

Discussion and Conclusion

In this section we discuss about our current implementations and further directions that can be useful to improve our results and conclude about our work. As an overall impression of the setup we are convinced that the results look promising. Our models VAE and GAN with almost the same parameter setting (1.079.600 and 1.256.698 for MNIST and 4.348.547, 3.576.704 for Imagenette respectively), which are trained with different architectures, for MNIST dataset both our models use only linear layers, which gave us very decent results and for Imagenette we used also convolutional layers, as with only linear layers the results were bad due to complexity of the color images. We trained our models for 30 and 50 epochs respectively. We consider this as a small number and believe that a larger number of epochs will yield to even better results. Also we tried small architectures as taking into account the parameters of the models and that is something that can improve the performance even more, if more complex architectures are involved together with higher number of epochs.
In conclusion, we manage to get reasonable results for both dataset, specifically very close generated images for MNIST and some close images for Imagenette. VAEs and GANs are as expected very promising at generating new images.

References

[1] BKHMSI. *MNIST-Latent-Interpolation*. 2018. URL: https://github.com/BKHMSI/MNIST-Latent-Interpolation.git.

[2] Jakub M. Tomczak. *VU Deep Learning Lecture Slides, Lecture 6 and 7: Latent Variable Models, Implicit Models*. [Online; accessed 18-December-2020]. 2020. URL: hhttps://www.youtube.com/watch?v=2nqtz3GzybQ.

[3] mseitzer. *FID score for PyTorch*. 2020. URL: https://github.com/mseitzer/pytorch-fid.

[4] Wikipedia contributors. *Kullback–Leibler divergence — Wikipedia, The Free Encyclopedia*. [Online; accessed 20-December-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=Kullback%E2%80%93Leibler_divergence&oldid=994841348.

Appendix A: Model Architectures

Listing 1: Architecture of the VAE model trained on MNIST

```
VAE(
  (enc1): Linear(in_features=784, out_features=512, bias=True)
  (enc2): Linear(in_features=512, out_features=256, bias=True)
  (enc3): Linear(in_features=256, out_features=32, bias=True)
  (dec1): Linear(in_features=16, out_features=256, bias=True)
  (dec2): Linear(in_features=256, out_features=512, bias=True)
  (dec3): Linear(in_features=512, out_features=784, bias=True)
)
```

Listing 2: Architecture of the VAE model trained on Imagenette

```
VAE(
  (encoder): Sequential(
    (0): Conv2d(3, 32, kernel_size=(4, 4), stride=(2, 2))
    (1): ReLU()
    (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
    (3): ReLU()
    (4): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2))
    (5): ReLU()
    (6): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2))
    (7): ReLU()
    (8): Flatten()
  )
  (fc1): Linear(in_features=1024, out_features=32, bias=True)
  (fc2): Linear(in_features=1024, out_features=32, bias=True)
  (fc3): Linear(in_features=32, out_features=1024, bias=True)
  (decoder): Sequential(
    (0): UnFlatten()
    (1): ConvTranspose2d(1024, 128, kernel_size=(5, 5), stride=(2, 2))
    (2): ReLU()
    (3): ConvTranspose2d(128, 64, kernel_size=(5, 5), stride=(2, 2))
    (4): ReLU()
    (5): ConvTranspose2d(64, 32, kernel_size=(6, 6), stride=(2, 2))
    (6): ReLU()
    (7): ConvTranspose2d(32, 3, kernel_size=(6, 6), stride=(2, 2))
    (8): Sigmoid()
  )
)
```

Listing 3: Architecture of GAN

```
Generator(
  (gen): Sequential(
    (0): Linear(in_features=64, out_features=256, bias=True)
    (1): LeakyReLU(negative_slope=0.01)
    (2): Linear(in_features=256, out_features=784, bias=True)
    (3): Tanh()
  )
)
Discriminator(
  (disc): Sequential(
    (0): Linear(in_features=784, out_features=128, bias=True)
    (1): LeakyReLU(negative_slope=0.01)
    (2): Linear(in_features=128, out_features=1, bias=True)
    (3): Sigmoid()
  )
)
```