Deep Learning 2020 **(Due: 20 Dec 2020)**

Submission Assignment #5A

Instructor: Jakub Tomczak *Name:* Sebastian Keil, Asif Anwar, *Netid:* 2664850, 2660561

1 Theory

Variational auto-encoder (VAEs) and generative adversersial networks (GANs) both belong to the broader family of generative networks. As the name suggests, those types of networks are able to generate new data (e.g. images), which can generally be viewed as an unsupervised task (although some models utilize elements of (semi) supervision. VAEs and GANs function in fundamentally different ways. VAEs first encode an input instance into a latent space that maps to a given distribution, typically the standard normal, and then decode this latent vector back into a data that should roughly resemble the input. Through this de- and reconstruction process, the latent vector can actually capture semantic meanings (e.g. one number in the vector may actually represent to rotation of a face etc.), which makes VAEs an interesting model to capture deeper information about the data. GANs operate in an adversarial context, where two neural networks essentially play a min-max game against each other. The first network generates images from random noise and its goal is to have the second network classify these fake images as real. The second network, on the other hand, aims to classify fake images as fake and real images as real. Hence, the first network is called 'generator' and the second network is called 'discriminators'. While GANs do not capture semantic meaning like VAEs do, they generally create more convincing (and less blurry) fake data.

1.1 Objective functions

VAE The objective function of the VAE consists of two parts: reconstruction loss and KL divergence. The reconstruction loss can be chosen by the researcher, it could for example be mean-squared error, cross-entropy or Frechet distance. The purpose of the reconstruction loss is simply to quantify how different the generated instance is from the original input data. This contrains the network to not just create any random data, but aim to replicate the input earnestly. The second term of the objective function is the KL divergence, which measures how much the distribution mapped onto by the encoder differs from the standard normal. This aims to constrain the network in the choice of parameters (mu, variance), so that the distributions in the latent space are a nice continuum instead of scattered all over the place. We can essentially imagine the latent space as a sort of ball of distribution, in which each latent variable is its own sphere. The KL divergence term makes sure that these distributional spheres do not just float randomly throughout space, but that they cluster together. In that manner, the decoder can work more reliably and we can inspect the latent space meaningfully as there is a continuum of meaning encoded in it. This allows us to manipulate the latent space and inspect the generated results, which will be shown later in this paper. Mathematically, we can describe the different components of the VAE loss as follows:

$$CE = -\frac{1}{N}\sum_{i=1}^{N} x_i log\hat{x}_i + (1-x_i)log(1-\hat{x}_i)$$

$$KL = -0.5\sum_{i=1}^{N}(1+\sigma^2-\mu^2-e^{\sigma^2})$$

$$TotalLoss = CE + KL$$

In PyTorch, we can implement this loss function as follows:

```
def loss_term(x, x_hat, mu, var):
  CE = F.binary_cross_entropy(x_hat, x.view(-1, 784), reduction='sum')
```

```
    KLD = -0.5 * torch.sum(1 + var - mu.pow(2) - var.exp())
    return CE + KLD
```

GAN *There are two main components of generative modeling network, generator (G) and discriminator (D). The generator latent directed variable model that generates samples x from z, and the discriminator D is a classification model which classify real and sample generated by the generators.*

$$min_G max_D E_{xP_{real}}[log_D(x)] + E_{zp(z)}[log(1 - D(G(z)))]$$

1.2 Components

VAE There are essentially three components that characterize the VAE which are the encoder, decoder and latent space. Here are some remarks on each of these components:

- **Encoder:** The encoder is a neural network (for this paper we use a simple MLP, but it could in principle be other architectures), that transforms the (higher dimensional) input into a latent vector. Unlike regular auto-encoders, which compute this latent vector deterministically, in the VAE the encoder actually produces a two vectors, one representing means and the other variances (which are then converted to standard deviations). This can be interpreted as the parameters of a distribution, i.e. its offset (mean) and stretch (std). To get a concrete sample vector for z, we sample from these given distributions, using the reparameterization trick.

- **Decoder:** The decoder receives the sample latent vector z as its input and learns to reconstruct the original image as good as possible (which is achieved by penalizing a reconstruction loss). Like the encoder, this paper uses a simple MLP to decode the latent vector, but in principle more complex structures could be used.

- **Latent space:** The latent space is of particular interest in the VAE model, as this is essentially what makes VAEs so versatile. Unlike autoencoders, the latent space in the VAE does not just represent a particular point, but a continuous range (this can be visualized as a bubble or a ball). This enables us to later detach the decoder from the model and have it generate random images based on some new z-vector (which may be randomly generated or designed in a certain way). Note that in the autoencoder setting, each specific point that is mapped to in the latent space would be separated from other point by an infinite amount of continous values. Hence, if we just fa random z-vector to the decoder network, it would not know what to do since it in all likelihood would never have seen this continous value before. The VAE however can still generate output based on a random z-vector, since these random values likely lie within the range at least one of the continuous distributions mapped to in the latent space. To avoid the problem of distributions 'drifting apart', like the points in the regular autoencoder model, the VAE employs a loss that penalizes the divergence between the distribution found by the encoder and a standard normal distribution, which is the KL divergence.

GAN *The there two major components of the generative adversarial networks, generator and discriminator. Details of these components are given below:*

- **Generator:***The generator learns to create sample data by incorporating feedback from the discriminator. The generator network takes random input and transform the input into a sample instance. Based on discriminator output, generator finds the generator loss. Based on this loss the parameters of the generator are updated.*

- **Discriminator:***The discriminator is a simple classifier. The objective of the discriminator is to distinguish the real data and the sample data from the generator. The input for discriminator comes from two sources,* ***Real data*** *discriminator uses this data as positive examples during training* ***Fake data*** *The discriminator uses these instances as negative examples.*

1.3 Adversarial loss:

The learning objective of generative adversarial networks is Adversarial loss. GANs treis to replicate a latent probability distribution. The loss function of GAN reflect the distance between the sample generate by generator and the real data. Minimax loss and Wesserstein loss are the two common loss functions. For the implemented

*GAN in this paper, Minimax loss function is used. The objective of the generator is to minimize the function $E_{xP_{real}}[log_D(x)] + E_{zp(z)}[log(1 - D(G(z)))]$, On the other hand discriminator tries to maximize this function. The two models generator and discriminator has opposite objective, as a result during the training models show adversarial behavior and try to improve their self in respect to the other one. The **BCEWithLogitsLoss()** function of the pytorch has been used during the training the GANs.*

2 Implementation

VAE We choose the same architecture for both the MNIST and Imagenette datasets. In the encoder, we first pass the input through a linear layer, reducing the input dimensions (1x28x28 for the MNIST, 3x64x64 for Imagenette) to 400 neurons. We then use another two linear layer to map to two vectors, which represent the distribution variables for the means and variances. To generate the z vector, we use the reparameterization trick, in which we sample some noise ϵ from a standard gaussian, multiple by our trained variables σ and add μ to have each z variable represent a distribution. For the MNIST data, we choose a length of 2 for the z-vector, as this is a relatively simple dataset and a shorter latent vector makes it easier to manipulate the latent space and infer the meaning of the given components. For the Imagenette data, we choose a length of 6. After computing the latent vector, the decoder essentially inverts the structure of the encoder, first broadcasting the latent vector to 400 neurons and then back to the original dimension (flattened out). We choose the following hyperparameters: learning rate = 1e-3, epochs= 30.

GAN We have implemented two different GANs for the MNIST and Imagenette datasets. The details of each implementation are given below.

- **MNIST GAN:**

  For implementing MNIST GAN we have used linear models for both generator and discriminator. A generator block was designed with a Linear layer followed by batch normal layer and ReLU function. There are total four generator blocks followed by a linear layer and Sigmoid activation in a sequential model for the generator. The discriminator is a simple classifier network. A discriminator block with a linear layer and with LeakyReLU activation function was designed for this network. There are total three discriminator block followed by a Linear layers in a sequential model.

  The MNIST dataset was loaded with batch size 128 witouth any normalization. For training Adam optimizer with learning rate 0.00001 has been used. The BCEWithLogitsLoss function has been used finding log loss. The training was done for 200 epochs and intermediary steps were implemented for reviewing the output.

- **Imagenette GAN:**

  A DCGAN was implemented for the Imagenette GAN. We have implemented transpose2d convolution layers. Beside the last convolution layer a batch normalization and ReLU activation function were implemented after each convolution. For the last layer Tanh activation function is used. For the discriminator five convolution layers with batch normalization and LeakyReLU activation function was used. In the last layer a Sigmoid activation function was use to identify real or fake image.

  The imagenette data was loaded with a batch size of 64 for the training. While data loading the images were normalized and resized to 64x64. like the MNIST GAN Adam and BCEWithLogitsLoss functions were used for optimization and loss. For Adam optimization learning rate of 0.0002 and beta1 = 0.5 and beta2 = 0.999 are used.

3 Results

VAE For the MNIST data, after training for 30 epochs, we get the loss curve shown in Figure 1 below. We can observe that the loss keeps on decreasing with as the number of epochs decrease, indicating that a larger of amount of epochs is desired to get decent outcomes.

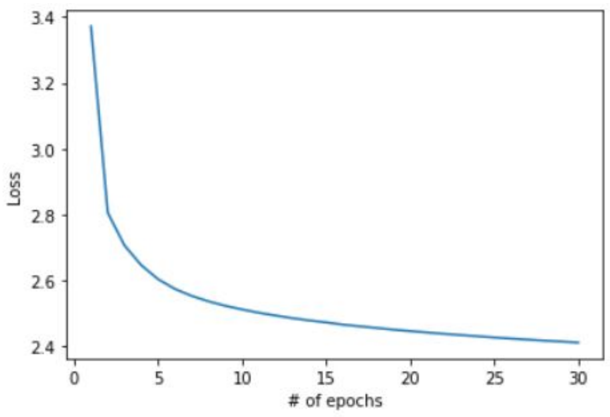


Figure 1: MNIST-VAE Training Loss Curve

In the validation loop, we can compare an original (input) batch of images to the reconstruction via visually inspection. Figure 2 shows a randomly picked batch and its reconstruction. It can be observed that while the reconstruction is generally more blurry, the VAE does manage to represent many number of the input accurately. However, it does apparently run into difficulties with some numbers, particularly the number 9 seems to be over represented in the reconstruction, as the model has difficulties distinguishing between 4 and 9, 5 and 9 and 8 and 9. Interestingly, the VAE mostly opts for representing this uncertain decision with a 9.

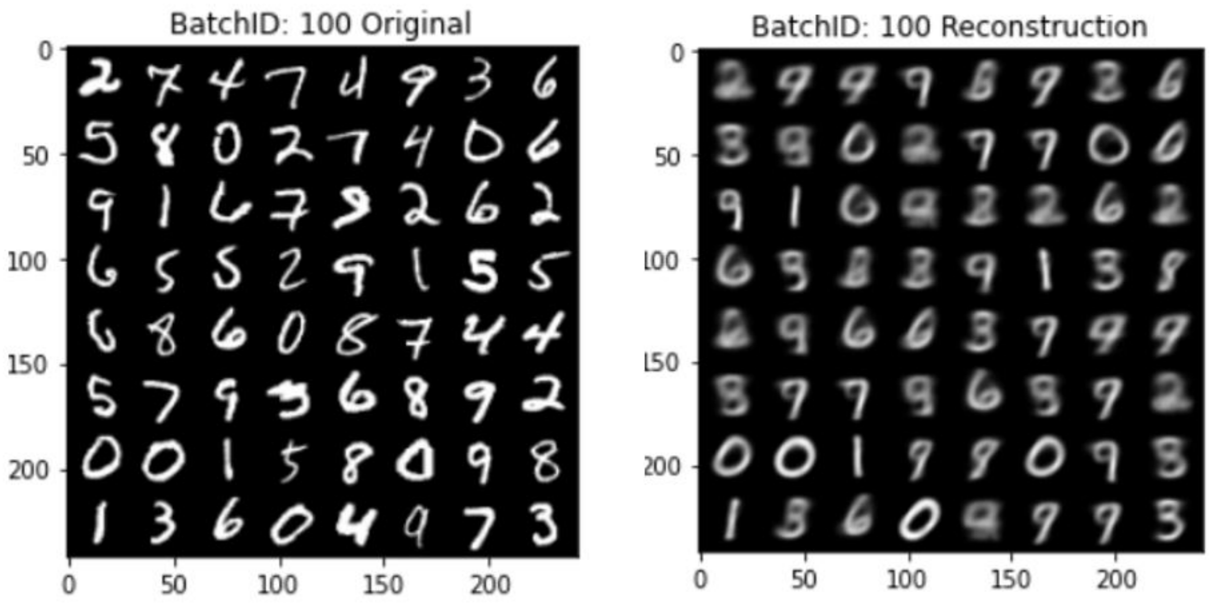


Figure 2: MNIST-VAE Reconstruction

Perhaps the most exciting part about VAEs is that it enables us to manipulate the latent space itself (e.g. by choosing certain numbers of the z-vector), interpolating them, and then using just the decoder to observe how these changes affect the generated results. One simple interpolation can be implemented via following code below. Remember that we choose a dimension of 2 for our z-vector, so each element in the list 'z' here represents one interpolation of the latent space. Here, we simply change the second element in the vector, from -2 to 1.5 in steps of 0.5. Figure 3 shows how this interpolation looks after decoding. We can see that the representation begins as a 0, then merges into a 6 and then into a 9. This gives us better intuitions on what the specific 'code' in the latent vector means.

```
with torch.no_grad():
    z = [[0, -2],[0, -1.5],[0, -1],[0, -0.5], [0, 0], [0, 0.5], [0, 1], [0, 1.5]]
```

```
sample = vae.decode(torch.FloatTensor(z))
sample_ = sample.view(len(z), 1, 28, 28)
show_images(sample_, 'Transformation␣in␣the␣Latent␣space')
```

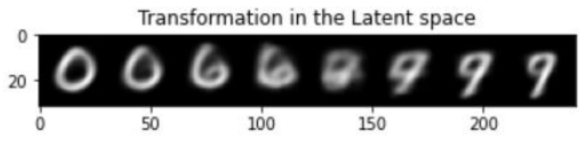


Figure 3: MNIST-VAE Interpolation

Since the Imagenette data is more complex as MNIST, it contains 3 color channels and depicts complex actions like men catching fish, the VAE will be trained for 100 epochs, using a learning rate of 1e-5. Figure 4 shows the loss behavior over the training period.

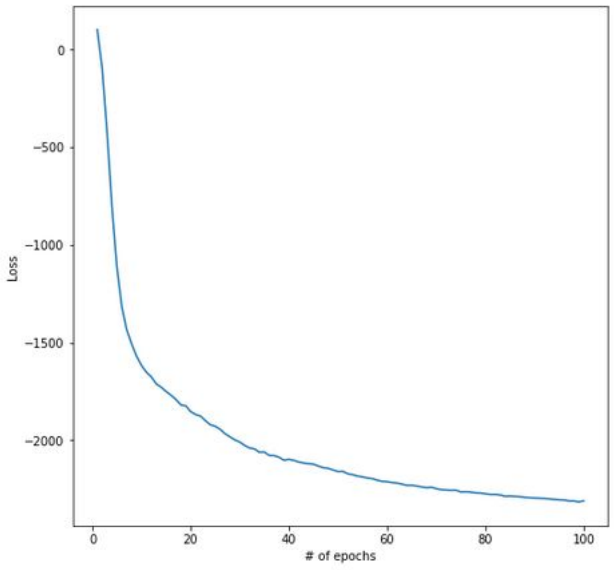


Figure 4: IMAGENETTE-VAE Loss Curve

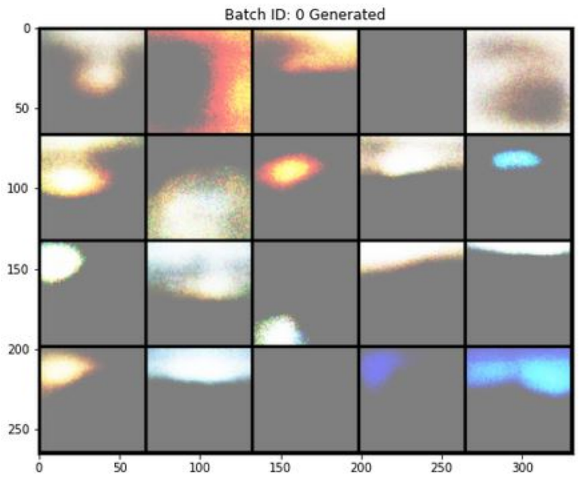


Figure 5: IMAGENETTE-VAE Test Reconstruction

Unfortunately, even after this long training session the VAE is not able to generate realistic images representing the Imagenette dataset. Figure 5 above shows that the generated images do not in any meaningful manner represent the input. We tried different learning rates, number of epochs and latent vector sizes, but these did not significantly improve the performance. It is beyond the scope of this paper, but using different architectures (e.g. CNNs), or larger models could perhaps improve the generation.

GAN The results of MNIST GAN and Imagenette GAN are given below:

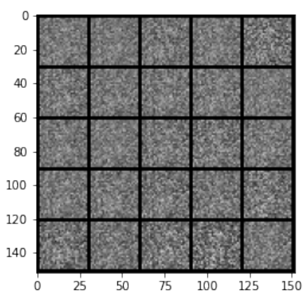
- **MNIST GAN**

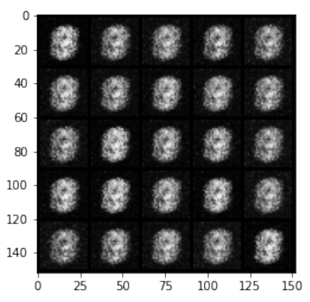
  The desinged GAN for the MNIST data set performed reasonably well. In the figure 7 we can view that the generator started improving gradually. In the beginning the output of the generator was simply noise. From 500 to 1000 steps we can see that generator started identifing the location to plot in the centre. from 2500 to 5000 steps the hazy handwriting can be seen. After 50000 steps the model created acceptable results for MNIST data set.

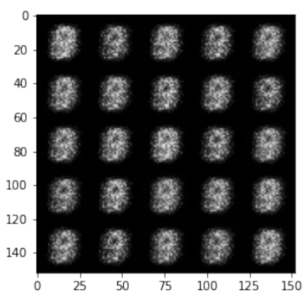
  We got 206.87 as the frechet distance score for MNIST GAN. As we know the lower the score better it is. Centrally this cold be improved with more complex network. In the figure 9.(a) we can the inception feature results for this network. From the distribution we can also see that the distribution of real and fake images are very close to each other.

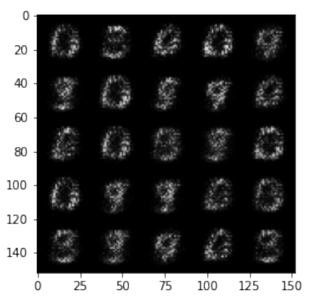
- **Imagenette GAN**

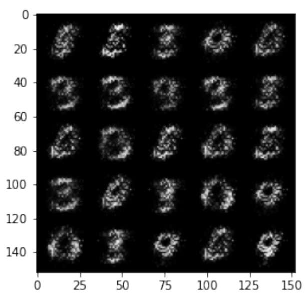
  Despite using convolution generative adversarial networks, the model did not able to reproduce any good results to reproduce imagenette images. We have tuned the hyper parameters in may ways to get good result, however due to complexity of the sources and the processing time we could not implement a well trained network to reproduce images. Although many sources were verified to build the network the output didn't show any sort of improvement. GAN is very sensitive network and requires very sophisticated training to perform well. The frechet distance score for Imagenette data set is 826.69, which means the the generator did not able to reproduced good enough images. We can also see that reflection in the inception feature of the real and fake images in the figure 9.(b). The distribution and real and fake images are very different which reinforce that the model didn't able to reproduce good output.

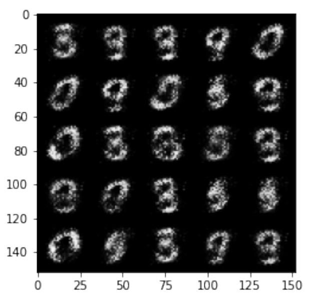

(a) Initial output

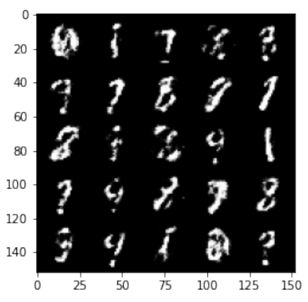

(b) 500 steps


(c) Steps 1000


(d) Steps 2500


(e) Steps 3500


(f) Step 5000


(g) 50000 steps

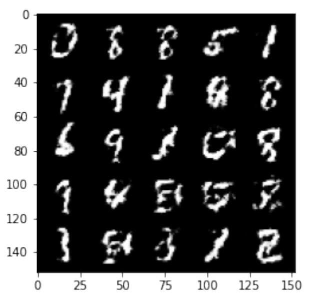

(h) Final Output

Figure 6: Sample output of MNIST in different steps

(a) Real

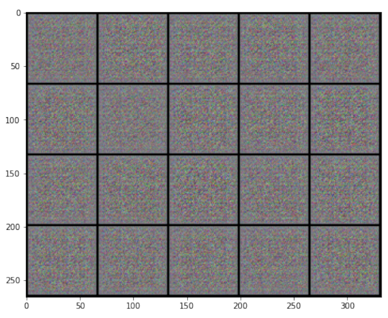
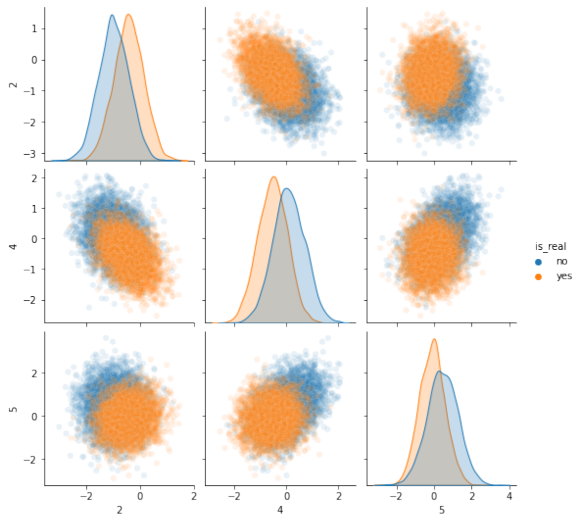
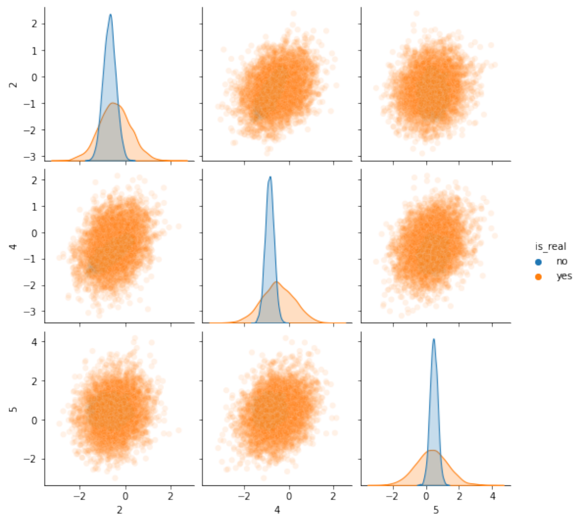

(b) Generated

Figure 7: Performance of GAN for ImageNet dataset



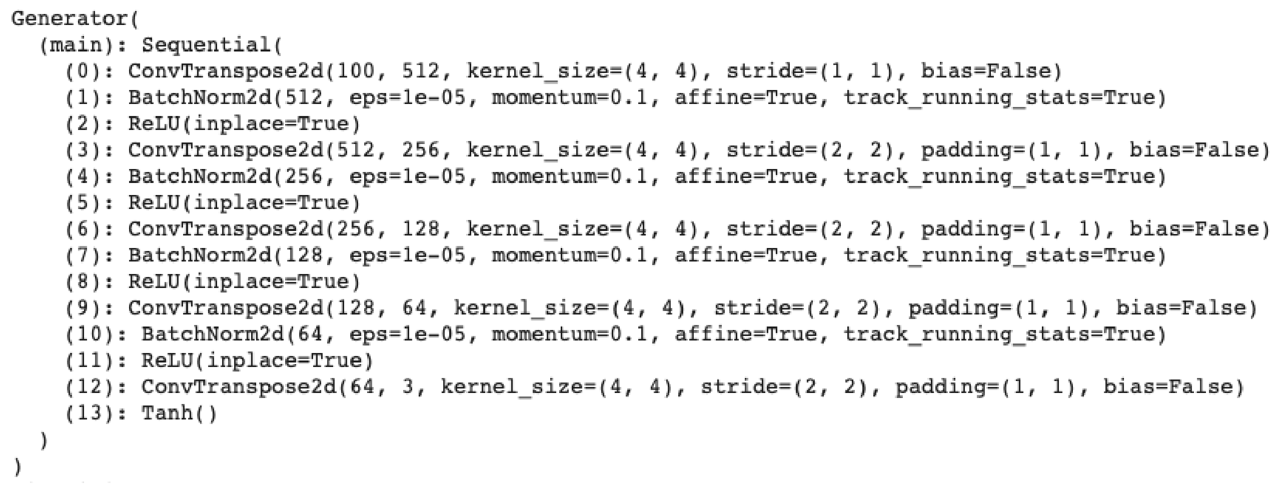
(a) MNIST inception features



(b) ImageNet inception features

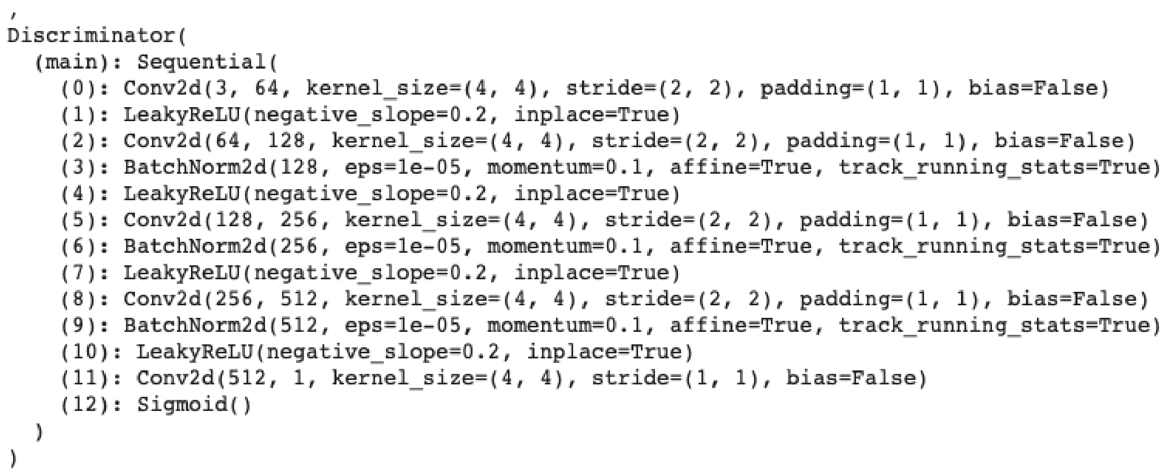
Figure 8: Pairwise multivariate distributions of the inception features

4 Appendix



```
Generator(
  (main): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)
```

(a) Generator Summary



```
,
Discriminator(
  (main): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (12): Sigmoid()
  )
)
```

(b) Discriminator Summary

Figure 9: Network Summary for Generative networks