About this submission

- All code has been written in Python.
- The submission consists of a single Jupyter notebook (*.ipynb), which consists of all the code used to generate results.
- Please note that it may not be possible to run and generate same results from the jupyter notebook, since entire codebase was condensed to a single file.
- For **Variational Auto Encoders (VAE)**, Questions 1 (Vanilla-VAE) and 3 (VQ-VAE) have been attempted.
- For **Generative Adversarial Networks (GAN)**, Questions 1 (Vanilla-DC GAN) and 3 (Conditional GANs) have been attempted.
- This report documents the implementation briefly.
- Plots and printed values obtained for each question are also documented below.

1 Vanilla Variational Auto Encoder (VAE)

1. Vanilla VAE uses a simple CNN based architecture for both Encoder and Decoder modules. Model summary is shown below:

```
----------------------------------------------------------
ENCODER MODEL
----------------------------------------------------------
Encoder(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (bnorm1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu1): LeakyReLU(negative_slope=0.01)
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (bnorm2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu2): LeakyReLU(negative_slope=0.01)
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (bnorm3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu3): LeakyReLU(negative_slope=0.01)
  (conv4): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (bnorm4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu4): LeakyReLU(negative_slope=0.01)
  (fcMu): Linear(in_features=4096, out_features=128, bias=True)
  (fcCov): Linear(in_features=4096, out_features=128, bias=True)
  (relu5): ReLU()
)
----------------------------------------------------------
```

Figure 1: VAE Encoder

```
----------------------------------------------------------
DECODER MODEL
----------------------------------------------------------
Decoder(
  (fc1): Linear(in_features=128, out_features=4096, bias=True)
  (convT1): ConvTranspose2d(256, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), output_padding=(1, 1))
  (bnorm1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu1): LeakyReLU(negative_slope=0.01)
  (convT2): ConvTranspose2d(128, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), output_padding=(1, 1))
  (bnorm2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu2): LeakyReLU(negative_slope=0.01)
  (convT3): ConvTranspose2d(64, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), output_padding=(1, 1))
  (bnorm3): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu3): LeakyReLU(negative_slope=0.01)
  (convT31): ConvTranspose2d(32, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), output_padding=(1, 1))
  (bnorm31): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu31): LeakyReLU(negative_slope=0.01)
  (convT4): ConvTranspose2d(32, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (tanh): Tanh()
)
```

Figure 2: VAE Decoder

2. For **CelebA** dataset, following training parameters are used:

Batch size	100
Optimizer	Adam()
Learning Rate	0.001
Input dimension	64x64x3
Latent dimension	128

3. For **dSprites** dataset, all training parameters remain the same as above, except that the Input Size is 64x64x1.

4. **Training objective** The training is setup to solve the following optimization problem:

$$\begin{aligned} \min_{\theta,\phi} \quad & E(X_i, \theta, \phi) \\ \text{where,} \quad & E(X_i, \theta, \phi) := \sum_{i=1}^{N} \frac{||X_i - \hat{X}||^2}{2\sigma^2} + D_{KL}[q_\phi(z|X_i)||p_\theta(z)] \\ \text{and} \quad & D_{KL}[q_\phi(z|X_i)||p_\theta(z)] = -\frac{1}{2}\sum_{j=1}^{J}(1 + log(\sigma_j^2)) + (\mu_j^2) - (\sigma_j^2) \end{aligned} \tag{1}$$

   Here, ϕ, θ refer to Encoder and Decoder parameters respectively. $X_i, \hat{X}$ are input training and reconstructed sample respectively. The variance σ^2 in reconstruction is emperically set by computing the variance of input data points. Also, $p_\theta(z)$ is taken to be $\mathcal{N}(0, I)$.

5. **Reparametrization** is done using a simple update $z = \mu(X_i) + \Sigma(X_i) \odot \epsilon$. Only one latent sample per input data point.

6. Sample generation results for **CelebA** dataset are shown below.

Figure 3: Sample generated images (celebA)

7. Sample generation results for **dSprites** dataset are shown below.

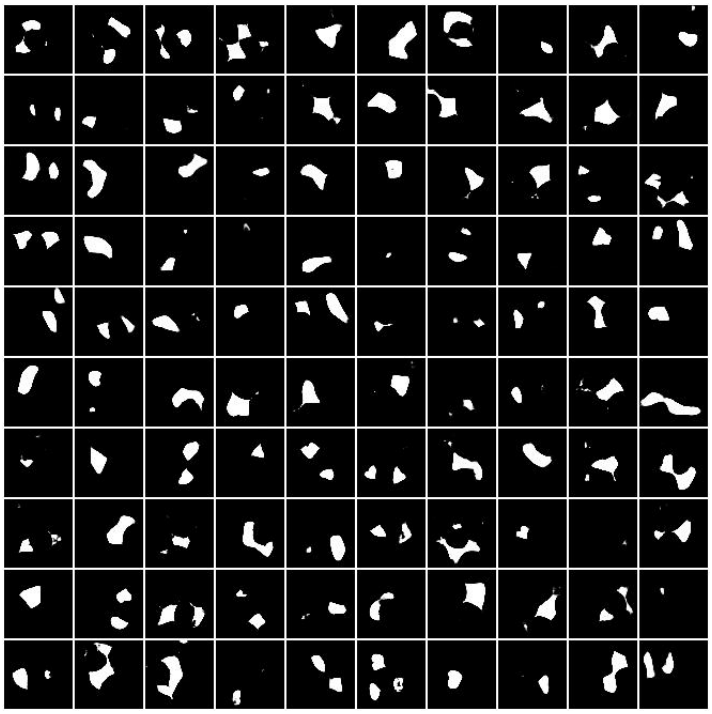


Figure 4: Sample generated images (dSprites)

2 Vector Quantized VAE (VQ-VAE)

1. The encoder, quantizer and decoder architecture for VQ-VAE is shown below:

2. Vanilla VAE uses a simple CNN based architecture for both Encoder and Decoder modules. Model summary is shown below:

```
----------------------------------------------------------
ENCODER MODEL
----------------------------------------------------------
Encoder(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (bnorm1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu1): LeakyReLU(negative_slope=0.01)
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (bnorm2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu2): LeakyReLU(negative_slope=0.01)
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (bnorm3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu3): LeakyReLU(negative_slope=0.01)
  (conv4): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (bnorm4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu4): LeakyReLU(negative_slope=0.01)
)
```

Figure 5: VQ-VAE Encoder

```
----------------------------------------------------------
VQ LAYER
----------------------------------------------------------
VQ(
  (mse_loss): MSELoss()
  (dictionary): Embedding(128, 16)
)
```

Figure 6: Vq-VAE Quantizer

```
----------------------------------------------------------
DECODER MODEL
----------------------------------------------------------
Decoder(
  (convT1): ConvTranspose2d(128, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), output_padding=(1, 1))
  (bnorm1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu1): LeakyReLU(negative_slope=0.01)
  (convT2): ConvTranspose2d(64, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), output_padding=(1, 1))
  (bnorm2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu2): LeakyReLU(negative_slope=0.01)
  (convT3): ConvTranspose2d(32, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), output_padding=(1, 1))
  (bnorm3): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu3): LeakyReLU(negative_slope=0.01)
  (convT31): ConvTranspose2d(16, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), output_padding=(1, 1))
  (bnorm31): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu31): LeakyReLU(negative_slope=0.01)
  (convT4): ConvTranspose2d(16, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (tanh): Tanh()
)
```

Figure 7: VQ-VAE Decoder

3. The training parameters for VQ-VAE are shown below:

Batch size	128
Optimizer	Adam()
Learning Rate	0.002
Input dimension	64x64x3
Dictionary vector dimension	16
Dictionary size	64, 128 and 256
Beta	0.25

4. **Training objective** for VQ-VAE is shown below:

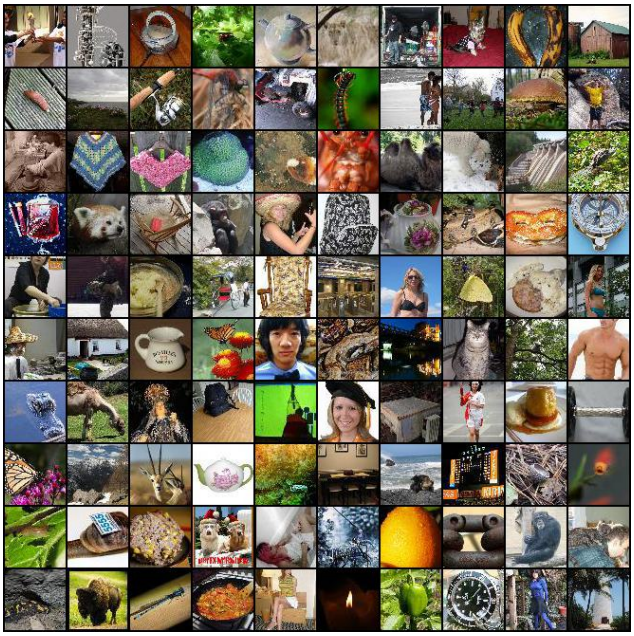
$$\min_{e,\theta,\phi} E(X_i, e, \theta, \phi)$$

$$\text{where,} \quad E(X_i, e, \theta, \phi) := \sum_{i=1}^{N} E_{BCE}(X_i, \hat{X}_i) + ||sg[z_e(X_i)] - e||_2^2 + \beta||z_e(X_i) - sg[e]||_2^2$$

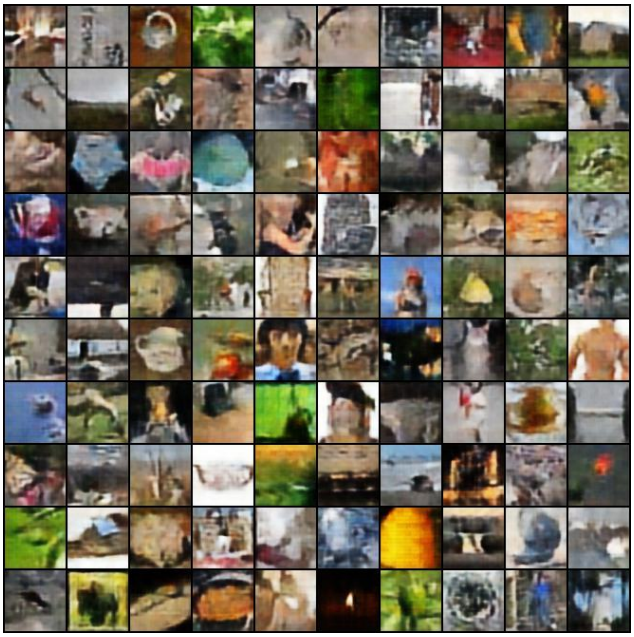
$$\text{with} \quad E_{BCE}(X_i, \hat{X}_i) := \sum_{m,k=1}^{M,K} X_i^{mk} log(\hat{X_i}^{mk}) + (1 - X_i^{mk}) log((1 - \hat{X_i}^{mk})) \tag{2}$$

Hence, binary cross-entropy loss is used for reconstruction. $sg[x]$ refers to stop gradient. (m, k) refers to a specific pixel in an image.

5. Sample reconstruction results for **tinyImagenet** dataset, and dictionary size=64, 128 and 256 are shown below.

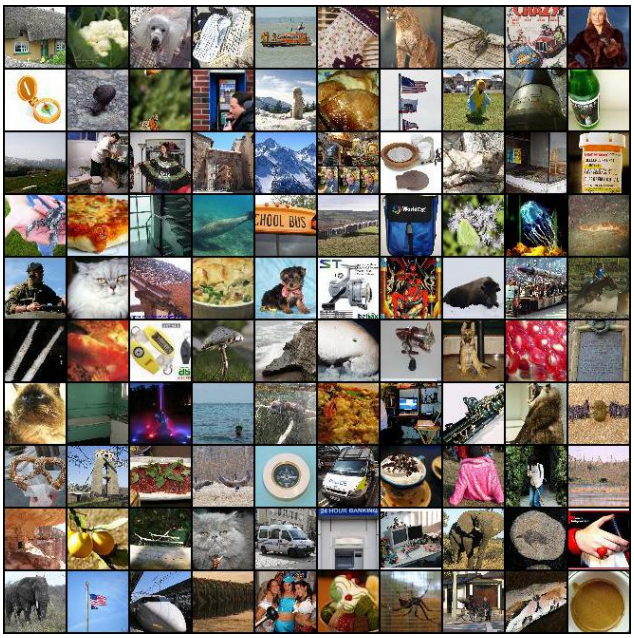


(a) Original data points

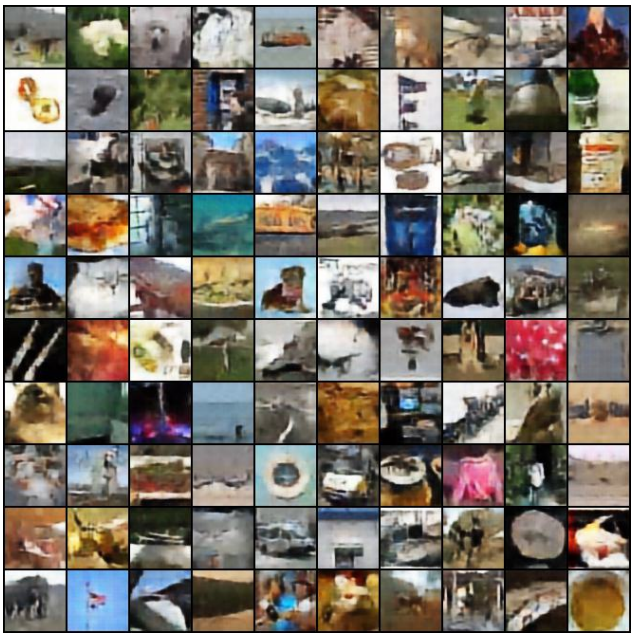


(b) Reconstructed samples

Figure 8: Sample reconstructed images (Dictionary size = 64)

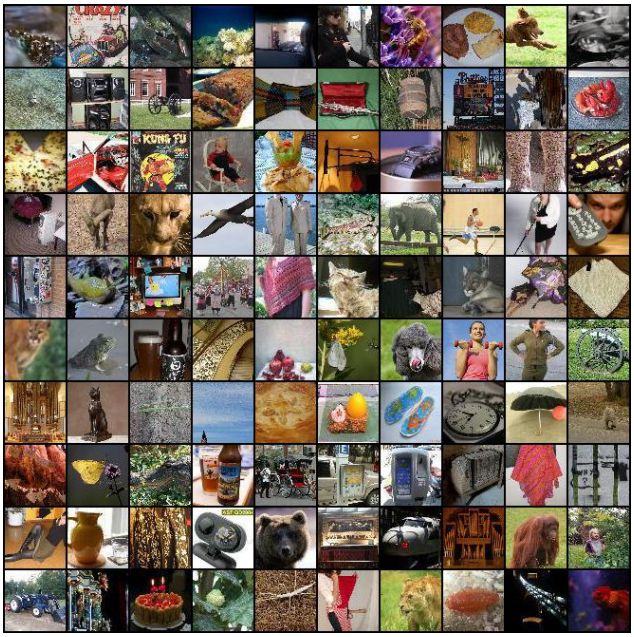


(a) Original data points

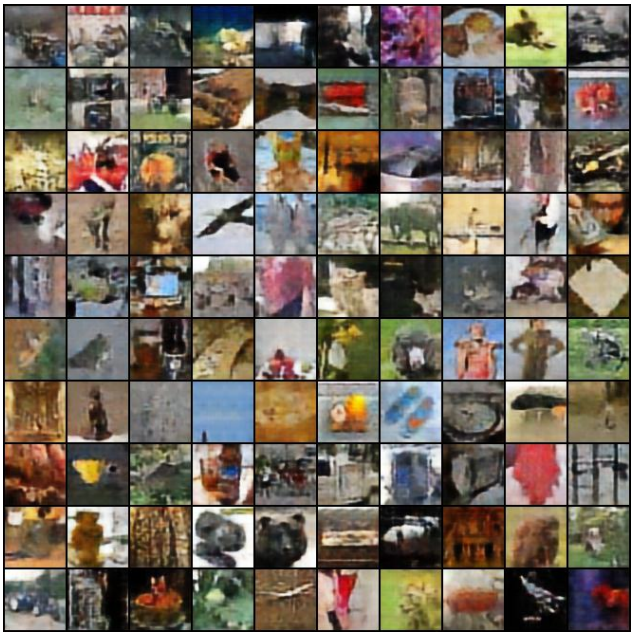


(b) Reconstructed samples

Figure 9: Sample reconstructed images (Dictionary size = 128)



(a) Original data points



(b) Reconstructed samples

Figure 10: Sample reconstructed images (Dictionary size = 256)

6. **Fitting Gaussian Mixture Model (GMM) on latent space** was done using the following parameters. Please note that Principal Component Analysis (PCA) was done to reduce dimension of the latent space before fitting.

Number of GMM Components	16
Number of data points to fit GMM	5000
Dimension of reduced space (for PCA)	256

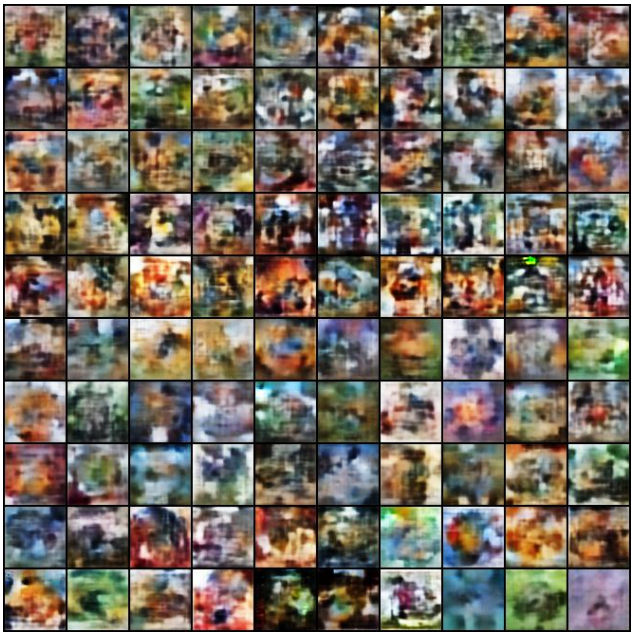


Figure 11: Sample images generated from GMM fit on latent space

7. Finally, **Vanilla VAE was fit on the latent space** of the trained VQ-VAE model. The relevant parameters for this exercise are shown below.

Batch size	100
Optimizer	Adam()
Learning Rate	0.001
Latent dimension	32

Figure 12: Sample images generated from VAE fit on latent space

3 Generative Adversarial Networks (GAN)

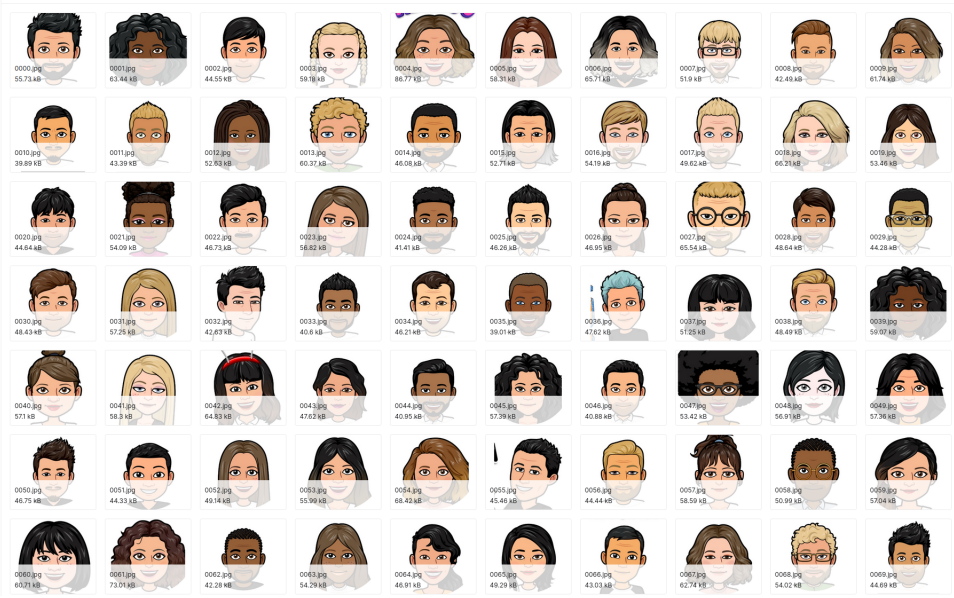


Figure 13: Bitmoji Dataset for DC-GAN

0.1 Model

Hyperparameters:

Batch size	3200
Optimizer	Adam()
Beta1 for Adam	0.5
Learning Rate	0.0002
Input dimension	64x64x3
Size of z latent vector	100
Size of feature maps in generator	64
Size of feature maps in discriminator	64

0.1.1 Generator

The generator, G, is designed to map the latent space vector (zzz) to data-space. Since our data are images, converting (z) to data-space means ultimately creating a RGB image with the same size as the training images (i.e. 3x64x64). In practice, this is accomplished through a series of strided two dimensional convolutional transpose layers, each paired with a 2D

batch norm layer and a relu activation. The output of the generator is fed through a tanh function to return it to the input data range of [1,1]. It is worth noting the existence of the batch norm functions after the conv-transpose layers, as this is a critical contribution of the DCGAN paper. These layers help with the flow of gradients during training. An image of the generator from the DCGAN paper is shown below.

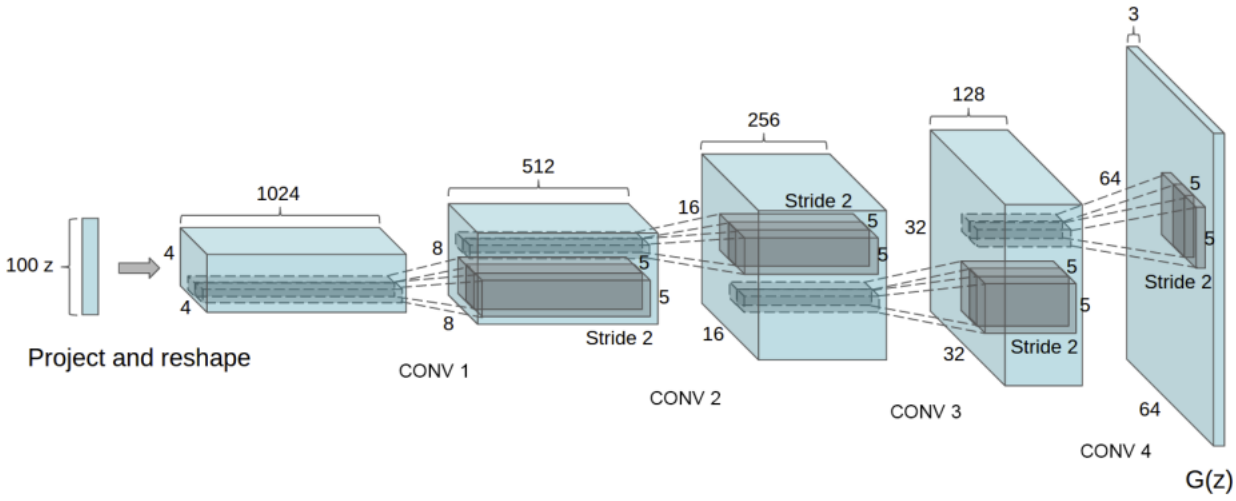


Figure 14: Generator for DC-GAN

0.1.2 Discriminator

The discriminator, D, is a binary classification network that takes an image as input and outputs a probability that the input image is real (as opposed to fake). Here, D takes a 3x64x64 input image, processes it through a series of Conv2d, BatchNorm2d, and LeakyReLU layers, and outputs the final probability through a Sigmoid activation function. This architecture can be extended with more layers if necessary for the problem, but there is significance to the use of the strided convolution, BatchNorm, and LeakyReLUs. The DCGAN paper mentions it is a good practice to use strided convolution rather than pooling to downsample because it lets the network learn its own pooling function. Also batch norm and leaky relu functions promote healthy gradient flow which is critical for the learning process of both G and D.

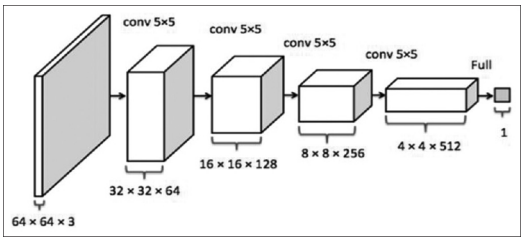


Figure 15: Discriminator for DC-GAN

0.1.3 Loss Functions and Optimizers

With D and G setup, we can specify how they learn through the loss functions and optimizers. We will use the Binary Cross Entropy loss (BCELoss) function which is defined in PyTorch as:

$$(x, y) = L = l_1, \ldots, {l_N}^T, l_n = -[y_n * \log x_n + (1 - y_n) * \log(1 - x_n)]$$

this function provides the calculation of both log components in the objective function (i.e. log(D(x)) and log(1-D(G(z))). We can specify what part of the BCE equation to use with the y input.

0.2 Loss Curves

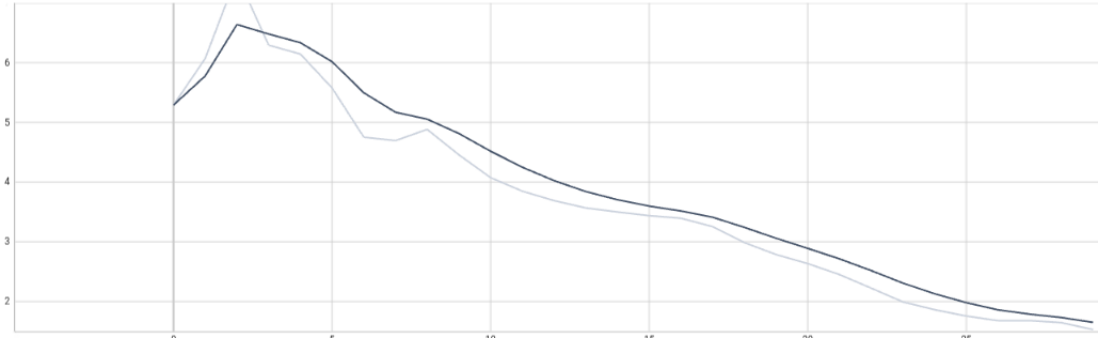


Figure 16: generator loss for DC-GAN

Note: Here we plotted curves by smoothing provided in tensorboard, thus we can view the exact pattern and get away from just the jittering patterns in the loss.
Here from the loss curve of generator, we can observe that the generator loss is in general decreasing over time. This means that our Generator is learning properly and **generator is able to learn to generate images that are of similar nature as real images.**
This loss convergence would normally signify that the GAN model found some optimum, where it can't improve more, which also should mean that it has learned well enough. (Also

note, that the numbers themselves usually aren't very informative.)
DCGAN contain some features like convolutional layers and batch normalisation, that are supposed to help with the stability of the convergence.

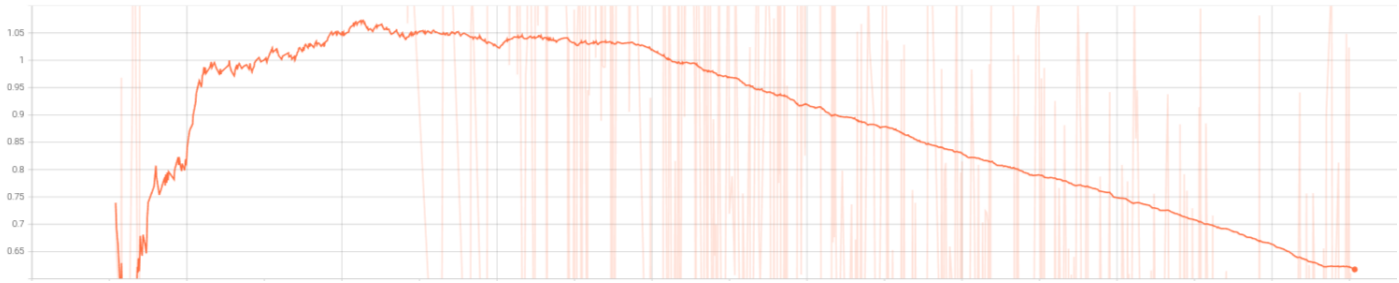


Figure 17: discriminator loss for DC-GAN

Note: Here we plotted curves by smoothing provided in tensorboard, thus we can view the exact pattern and get away from just the jittering patterns in the loss.
The loss of Discriminator shows that over time the loss is decreasing and it saturates **near to 0.6** thus we can conclude that now this is very near to optimal Discriminator as it is confused between real and generated image and outputs **similar probability** for both. Which means the training of DCGAN discriminator is till optimality.
Intuition: In simultaneously of G and D training. At the beginning, G is really bad at generating images, but D is also really bad at discriminating them. As time passes, G gets better, but D also gets better. So after many epochs, we can think that D is really good at discriminating between fake and real. Therefore, even if G "fools" D only 5% of the time (i.e. D(x)=0.95 and D(G(z))=0.05) then it can mean that G is actually pretty good because it fools sometimes a really good discriminator.

Figure 18: Generated Images from DC-GAN

This 10x10 grid displays the output of our final model. We generated images similar in appearance to the dataset. They are not as sharp as the original image and there are many random brush strokes but still the output shows our Generative Model has learnt well the datset.

0.3 Generated Images

Few observations that we can make looking at images is that Generative Model has well learnt:

- **The Skin texture:** We can find emojis of varied skin tone from dark to pale. Though somewhere it has learnt yellow also as skin colour but it seems overall model is going in right direction as yellow is near about to actual skin colour.
- **Gender:** It seems that fair amount of both the genders are represented.
- **Age:** We can see a spectrum of youth to late middle aged people in the generated grid.

- **hair Colour:** We have many shades of black and brown here. But soem emojis have red too.

Few artifacts:

- We see random colour strokes at many places.
- We observe the faces to be smooth.
- At many regions there are combinations of many different colours mixed.
- At some regions non natural face colours are also seen.

0.4 FID

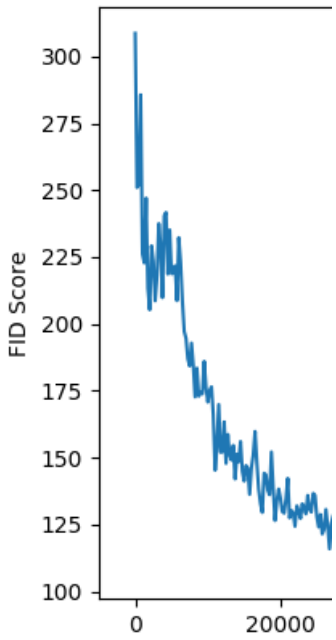


Figure 19: FID for DC-GAN

We observe that the FID is in general decreasing, thus we can believe that our model is learning good and the minimum value is 120.

4 Conditional Generative Adversarial Models

Dataset: SVHN
Hack: The training + testing data was: 73257 digits for training, 26032 digits for testing = 99289 . But there were 531131 additional samples, to use as extra training data. Thus we made our dataset over 6 lakhs.
In CGAN (Conditional GAN), labels act as an extension to the latent space z to generate and discriminate images better. Our conditional GAN changes adds the label y as an additional parameter to the generator and hopes that the corresponding images are generated. We also add the labels to the discriminator input to distinguish real images better. We feed the vector and the noise z to the generator to create an image that resembles "3" (for example). For the discriminator, we add the supposed label as a one-hot vector to its input.

How we used labels for conditioning: just made label as an extra channel with the image, thus for 10 digits we have 10 channels representing a digit and we pass this with image to discriminator and make this channel as an additional output from generator.

Figure 20: Generated Image by conditional WGAN (left) and conditional DCGAN (right) both conditioned on same digit [Top to bottom 0 to 9 all 9 digits conditioned]

0.5 Model Details:

0.5.1 conditional WGAN

We used a WGAN-GP (Gradient Penalty) and just made label as an extra channel with the image for both discriminator and generator.
Model Parameters and Algorithm We used the parameters in the original WGAN-GP

paper given below:

Algorithm 1 WGAN with gradient penalty. We use default values of $\lambda = 10$, $n_{\text{critic}} = 5$, $\alpha = 0.0001$, $\beta_1 = 0$, $\beta_2 = 0.9$.

Require: The gradient penalty coefficient λ, the number of critic iterations per generator iteration n_{critic}, the batch size m, Adam hyperparameters α, β_1, β_2.
Require: initial critic parameters w_0, initial generator parameters θ_0.

while θ has not converged **do**
  for $t = 1, ..., n_{\text{critic}}$ **do**
    for $i = 1, ..., m$ **do**
      Sample real data $\boldsymbol{x} \sim \mathbb{P}_r$, latent variable $\boldsymbol{z} \sim p(\boldsymbol{z})$, a random number $\epsilon \sim U[0, 1]$.
      $\tilde{\boldsymbol{x}} \leftarrow G_\theta(\boldsymbol{z})$
      $\hat{\boldsymbol{x}} \leftarrow \epsilon \boldsymbol{x} + (1 - \epsilon)\tilde{\boldsymbol{x}}$
      $L^{(i)} \leftarrow D_w(\tilde{\boldsymbol{x}}) - D_w(\boldsymbol{x}) + \lambda(\|\nabla_{\hat{\boldsymbol{x}}} D_w(\hat{\boldsymbol{x}})\|_2 - 1)^2$
    end for
    $w \leftarrow \text{Adam}(\nabla_w \frac{1}{m} \sum_{i=1}^{m} L^{(i)}, w, \alpha, \beta_1, \beta_2)$
  end for
  Sample a batch of latent variables $\{\boldsymbol{z}^{(i)}\}_{i=1}^{m} \sim p(\boldsymbol{z})$.
  $\theta \leftarrow \text{Adam}(\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} -D_w(G_\theta(\boldsymbol{z})), \theta, \alpha, \beta_1, \beta_2)$
end while

Figure 21: Algorithm for conditional WGAN

Batch size	3200
Optimizer	Adam()
Beta1 for Adam	0.0
Beta2 for Adam	0.9
Learning Rate	0.0001
Input dimension	64x64x3
Size of z latent vector	100
Size of feature maps in generator	64
Size of feature maps in discriminator	64

Architecture is same as DCGAN and conditional DCGAN but instead of sigmoid at end of Discriminator, we now output real number.

0.5.2 Conditional DCGAN

Same Architecture and hyperparameters as of vanilla DCGAN, just now also has label as extra channel with discriminator and generator.

0.6 Loss Curves

Figure 22: Loss Curve for Conditional DC-GAN

Figure 23: Loss Curve for conditional Conditional W-GAN

Interpretations: The differences in loss curves of DCGAN and WGAN is very stark. Where DC GAN uses a Jansen-Shannen divergence, WGAN introduces its own loss functions Wasserstein distance.
conditional WGAN loss of discriminator is clearly increasing and going to 0. And due to this generator is also very smoothly going to minimum.
conditional DCGAN loss of discriminator increases somewhat but stays same and thus generator loss is also not changing much.
Thus, learning is clearly seen in conditional WGAN but it is uncertain in conditional DC GAN.

0.6.1 Generated images

Figure 24: Images Generated by conditional DCGAN

0.6.2 Generated images

0.7 Vanishing Gradients

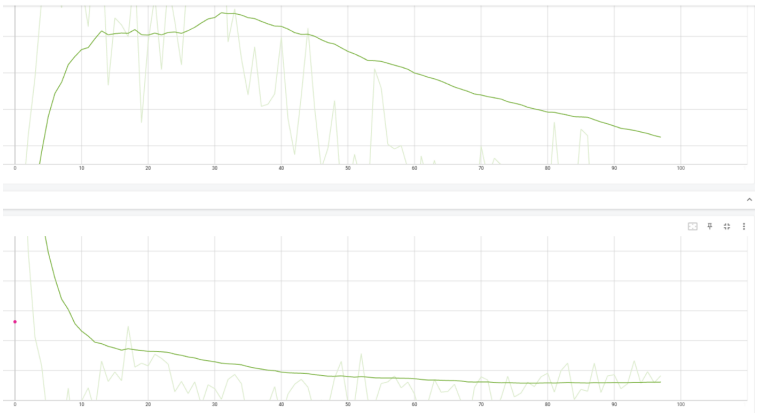


Figure 25: Norm of Gradient of Weights of First layer, c-WGAN (Top), c-DCGAN(Bottom)

This is the plot of norm of the gradient of the weights in the first layer of Critic network. the top plot is for conditional-WGAN and the bottom is one for conditional-DCGAN. We can observe that in initial few epochs only c-DCGAN has gradients of the first layer weights going to very low value and hence on decreasing only.
Whereas WGAN has much larger gradients propagated to initial layer and they eventually decrease as model training is going to convergence.

We plot the loss surface of conditional DCGAN corresponding to this gradient curve.

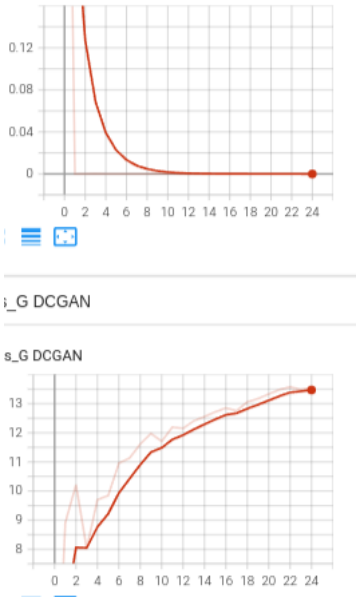


Figure 26: Loss curves of conditional DCGAN on vanishing gradient case.

0.8 FID

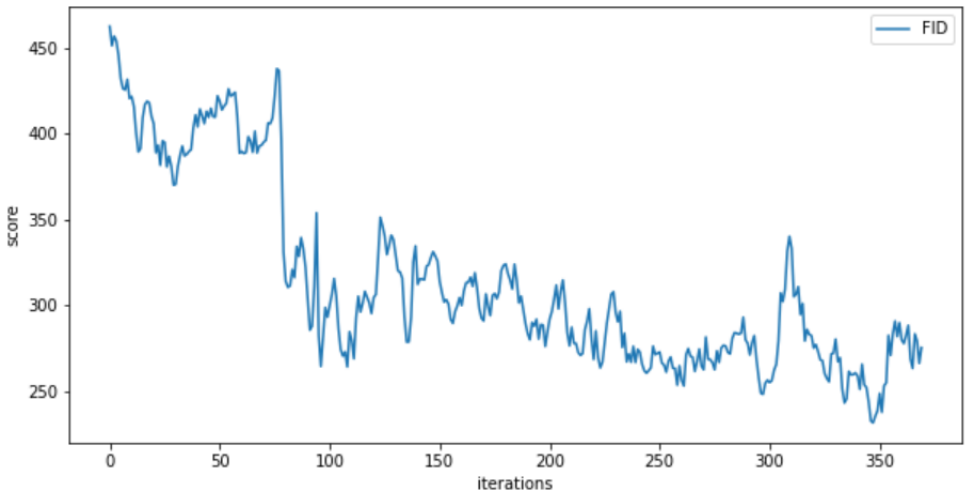


Figure 27: FID of conditional DCGAN

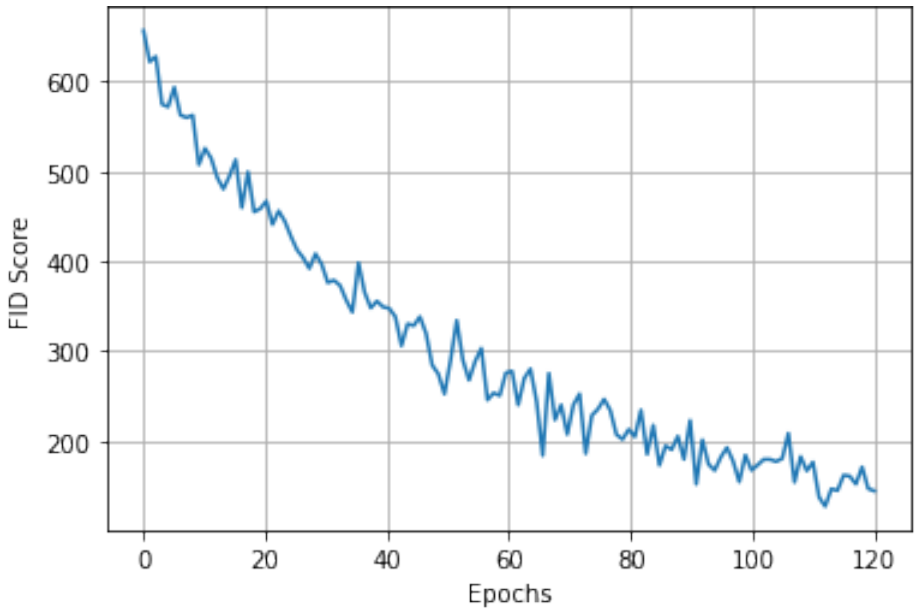


Figure 28: FID of conditional WGAN

Observations:

- The plots of FID evolution through epochs is a distinctive argument that **conditional WGAN trainnig is more stable than conditional DCGAN** training. In condi-

tional DCGAN FID plot we can observe that the plot is in general decreasing but there is no pattern. Whereas, in conditional WGAN we see a distinctive decay in FID.

- Secondly, we see conditional WGAN achieves a lower minima (less than 100) but conditional DCGAN though decreasing but is above 100 only.

0.9 Lipschitz Bound Enforcement

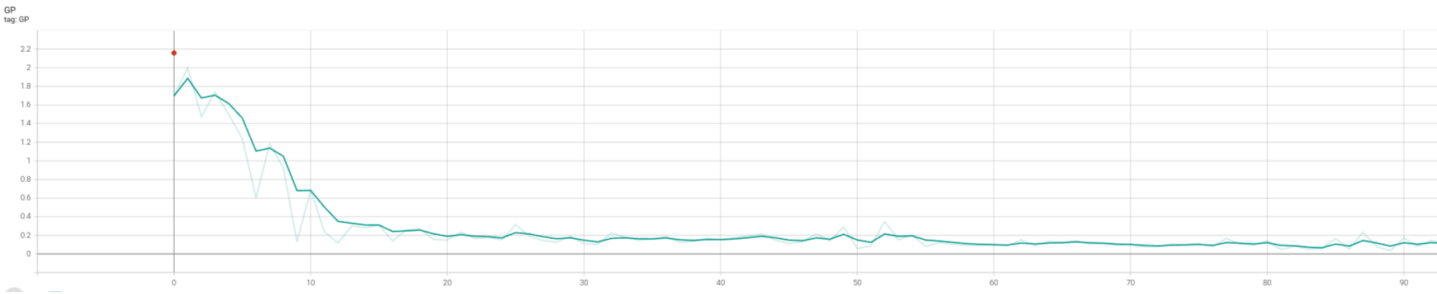


Figure 29: Gradient Penalty of conditional WGAN

This plot emperically shows that the Gradient Penalty is near to 0 only and it means that the critic of conditional WGAN-GP is nearly 1-Lipschitz. Thus **Gradient Penalty** to loss is actually bounding the lipschitz of critic network.

1 Appendix
Improve Gan training stability

- Avoid Sparse Gradients: ReLU, MaxPool: use LeakyReLU
  - For Downsampling, use: Average Pooling, Conv2d + stride
  - For Upsampling, use: PixelShuffle, ConvTranspose2d + stride
- Use the ADAM Optimizer
- Use Dropouts in G in both train and test phase
  - Provide noise in the form of dropout (50
  - Apply on several layers of our generator at both training and test time
- BatchNorm
  - Construct different mini-batches for real and fake, i.e. each mini-batch needs to contain only all real images or all generated images.
  - when batchnorm is not an option use instance normalization (for each sample, subtract mean and divide by standard deviation).
- **Failure Cases**
  - D loss goes to 0: failure mode
  - check norms of gradients: if they are over 100 things are screwing up
  - when things are working, D loss has low variance and goes down over time vs having huge variance and spiking
  - if loss of generator steadily decreases, then it's fooling D with garbage

□