Understanding Machine Learning Generative Models - GANs and VAEs

Srishti Goel

Contents

Chapter 1

Generative Adversarial Networks (GANs)

Generative adversarial networks are a class of generative models that use unsupervised learning to generate data (in this case, 2D images) similar to the input data.

The question then arises, what does "similar" even mean in this context? The problems that GANs are applied on do not have a fixed framework of similarity, and thus, allow the model to learn its own meaning of similarity. The problem statement does not give it any signals of whether it is doing better or worse (this is what it means to do "unsupervised learning").

GANs acheive this goal by setting up an adversarial game between 2 neural networks - the Generator and the Discriminator. The core ideology behind this? *Competition drives progress.*

The two neural networks are discussed below:

- **Generator** : The generator gets an input random vector of a fixed size (understood as a set of a fixed number of random variables).[1] It then uses these random variables to generate an image. In doing so, its objective is to make images that the discriminator cannot distinguish from the actual input images.

- **Discriminator** : The discriminator gets an input image (from the generator or the input images), and assigns it a particular real valued score.

[1]This is the latent vector.

In doing so, it tries to distinguish input images from input images by assigning a high score to the input images, and a low score to the generated images. This score can thus be seen as a measure of "realness" as understood by the discriminator.

During training, the generator and discriminator engage in a competitive process. The generator aims to produce realistic data, and the discriminator tries to understand key features in the input data to distinguish input data from generated data.

GANs are considered a type of implicit density models. They do not assume any likelihood on either the input random vector, or the data. Thus, it is more difficult to use a likelihood estimation, and use a deterministic mapping from noise to samples (the mapping is the discriminator).

GANs have demonstrated impressive capabilities in various domains. They have been successfully applied to tasks such as image synthesis, style transfer, image-to-image translation, text generation, and even video generation. GANs have also been instrumental in advancing the field of computer vision, as they can learn to generate highly realistic images that resemble the training dataset.

Upon applying a simplistic GAN model on the MNIST dataset, which has handwritten digits in every image, and are classified by digits ('0', '1', etc.), the evolution of generated images over the course of training was observed. (Fig. 1.1, Chapter 1)[2]

However, training GANs can be challenging. Finding the right balance between the generator and discriminator networks is a delicate process, and GANs are known for being sensitive to hyperparameter settings. Training instability manifest as mode collapse[3], and convergence issues[4], gradient van-

[2]While the training was not perfect at the end of epoch 60, the losses seemed to have converged, perhaps due to the simplicity of the model, or other factors as discussed.

[3]The generator appears to get fixated on the same or a small subset of images like the input images.

[4]The apparent improvement of one of the models leads to an apparent worsening of the other and this causes a sort of oscillation as seen in Fig. 1.4. Combined losses implies the sum of the losses of the generator and discriminator and is a function of just the discriminator acting on a morphed version of input images.

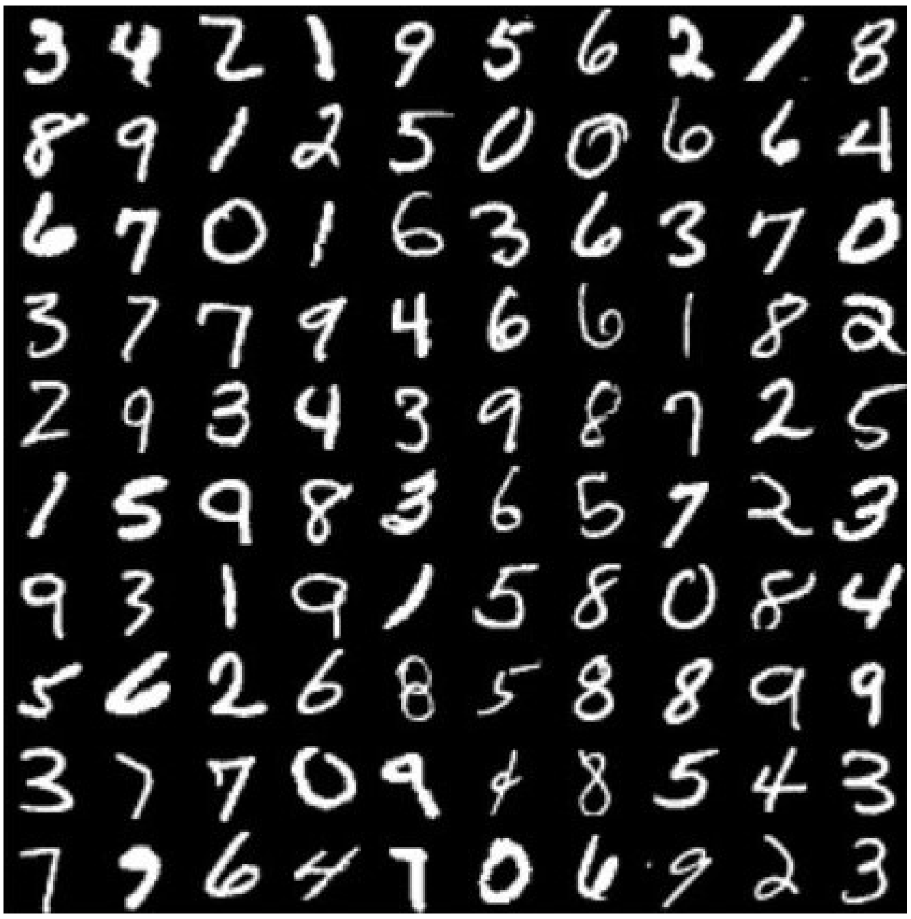


Figure 1.1: Original images from the MNIST dataset

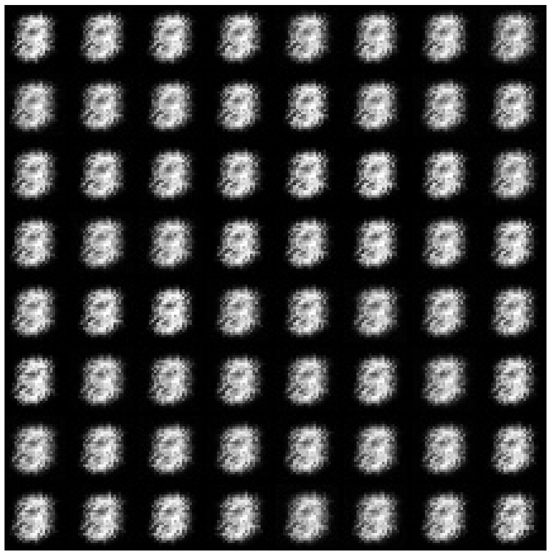


Figure 1.2: Output after 1 epoch

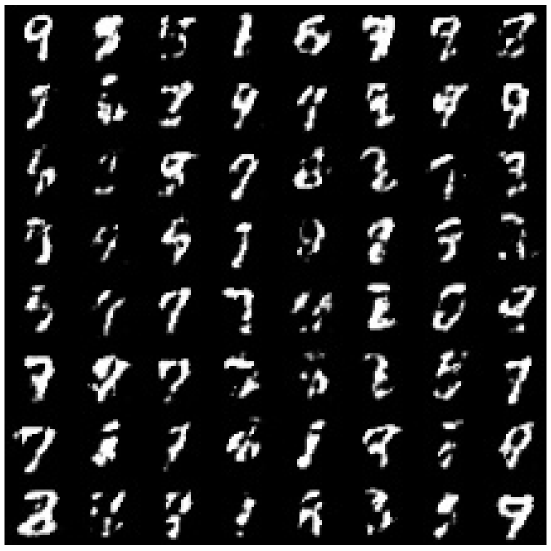


Figure 1.3: Output after 60 epochs

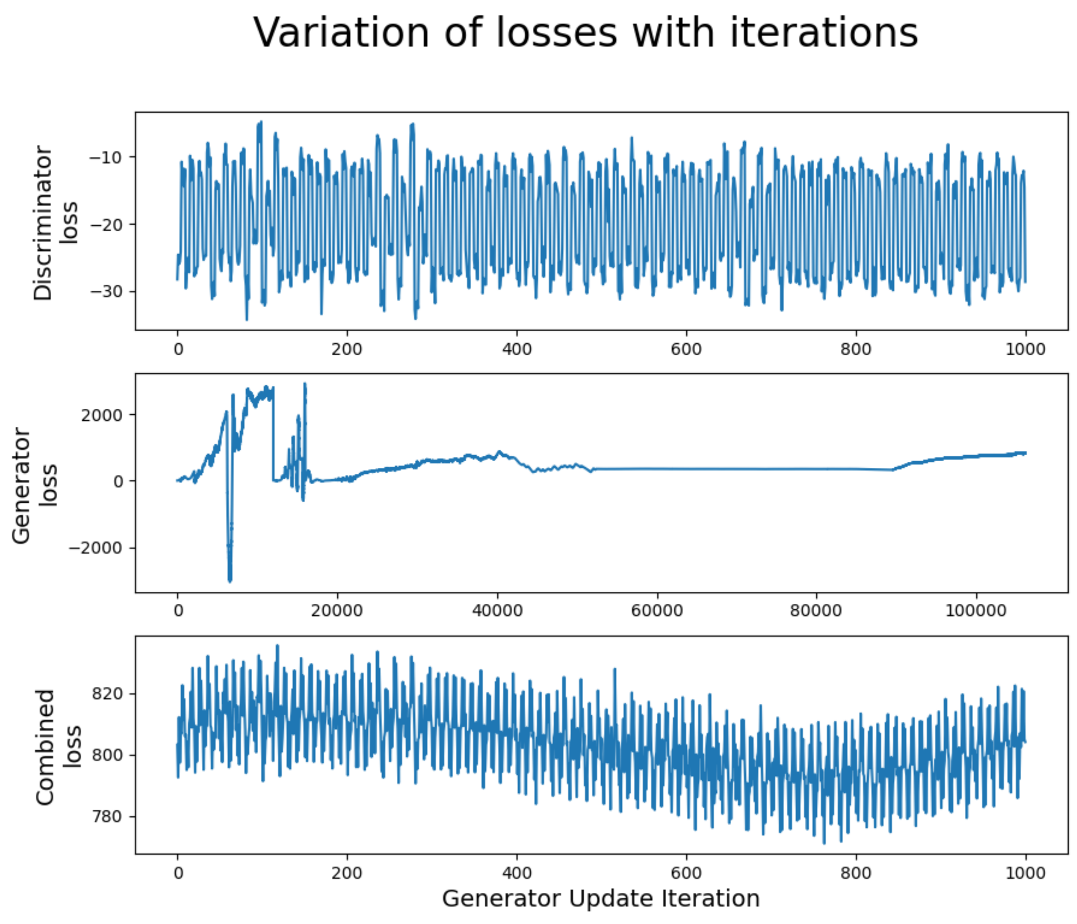


Figure 1.4: Example of the situation where the losses begin to oscillate

ishing[5], mode dropping[6] and hyperparameter sensitivity[7]are some common challenges faced when working with GANs.

Addressing training instability in GANs is an active area of research, and several techniques have been proposed to mitigate these issues. Some approaches include architectural modifications, improved loss functions, regularization techniques, and more stable training algorithms such as Wasserstein GANs or spectral normalization. Researchers are continually exploring new methods to enhance the stability and convergence of GAN training.

[5]If the gradients become too small to make meaningful upates, or too large, leading to unstable training updates. This was removed from our models by adding a factor of the gradient to the loss itself.

[6]The discriminator becomes fixated on particular features that are not actually relevant, but the generator accordingly fixates on them too, missing some key modes/patterns in the input data.

[7]GANs are sensitive to hyperparameter settings such as learning rates, network architectures, regularization techniques, and more. Suboptimal choices of these hyperparameters can cause training instability, making it challenging to achieve good results.

Despite these limitations, GANs provide stunning images , given all the parameters tuned right. This is why they still remain an area of active research to this day.

Chapter 2

Wasserstein Loss in Generative Adversarial Networks(GANs)

The loss function used in the Han et al. paper was the Wasserstein loss function (for all the 3 models).

Using the notation:

- $D(i)$ is the discriminator score for an image i (which may be an input image, or a generated image)
- I is an input image
- $\tilde{I}$ is a generated image
- λ_{GP} is a programmer-defined (hyperparameter) weight to the regularization term
- I_{interp} is the interpolated image described below
- $\nabla_{\mathbf{y}} f(\mathbf{y})$ represents the vector gradient of the function $f(\mathbf{y})$ acting on a multi-dimensional input $\mathbf{y}$. In case this is an image, we mean the gradient of $f(\mathbf{y})$ with respect to each pixel of $\mathbf{y}$. This is calculated by using the library function in *Pytorch*:

  ```
  torch.autograd.grad(input_value, output_value)
  ```

- $\|\mathbf{y}\|$ represents the vector norm of $\mathbf{y}$

The vanilla Wasserstein loss defines the loss of the Generator and Discriminator as:

$$\text{Loss of the Generator} = 0 - D(\tilde{I}) \tag{2.1}$$

$$\text{Loss of the Discriminator} = D(\tilde{I}) - D(I) \tag{2.2}$$

Thus, the Generator aims to increase the Discriminator's score on generated images. The Discriminator, on the other hand, aims to increase the divide between the score of generated images and score of input images. It does this by trying to maximize the score of input images, while decreasing the score of generated images. This is how the score is a measure of "realness" of an image.

However, to counter the mode-collapse faced by GANs, some works introduce an extra regularization factor to the loss of the Discriminator. This extra regularization is related to the gradient of the discriminator score on an interpolated image between input and generated image.

First, to generate this interpolated image, for each generated image, a random weight ϵ is selected uniformly between 0 and 1.

$$I_{\text{interp}} = \epsilon I + (1 - \epsilon)\tilde{I} \tag{2.3}$$

Thus, the interpolated image is a weighted average of the input and generated images in the image space.

Thus, this variation of the Wasserstein loss function (called the Wasserstein Loss function with Gradient Policy in the future) is calculated as:

$$\text{Loss of the Generator} = 0 - D(\tilde{I}) \tag{2.4}$$

$$\text{Loss of the Discriminator} = D(\tilde{I}) - D(I) + \lambda_{GP} \|\nabla_{I_{\text{interp}}} D(I_{\text{interp}})\| \tag{2.5}$$

The regularization term, thus penalizes the Discriminator for large gradients between generated and input images. If there are large gradients, the

Discriminator may be over-fitting to differentiate input and generated images. It would appear to introduce more twists and turns to differentiate all the generated images.

During training, MINI-BATCHES are batches of input images over which the loss is calculated, after which an update step is taken). While using these mini-batches, the regularization term uses its mean value over a mini-batch-long number of generated images matched to a corresponding input image each. The discriminator score is also similarly averaged over the mini-batch.

The question of how to choose the hyperparameter λ_{GP} remains an unanswered question for us right now.

Chapter 3

Variational Auto-Encoders (VAEs)

Variational Auto-Encoders are probabilistic generative models. They try to infer the distributions of the latent variables representing the observed data. The encoder, instead of mapping input images to the latent space (which is chosen to be 'n' dimensional, as a hyper-parameter of the model), maps input images to distributions of latent-space variables. The red box returns a single sample point from the (normal) distribution. This point (called the latent vector) is passed to the decoder, which then generates an image similar to the input image. However, in describing the distributions output from the encoder, matters can quickly become intractable. The key is to assume a multivariate Gaussian distributions (which simplifies the math, while easily mapping a large subset of distributions). The encoder must then simply "encode" each input image into a multi-dimensional mean and variance vector (the dimensions of which are at the choice of the engineer designing this machine).

While sampling an element of the distribution encoded by the encoder, we use a normal random vector with mean 0, variance 1. We then convert this into a vector from the encoded distribution as:

$$\mathbf{X} \sim \mathcal{N}(0, I)$$

$$\rightarrow \mathbf{Z} = \sigma \mathbf{X} + \mu \sim \mathcal{N}(\mu, \sigma^2 I)$$

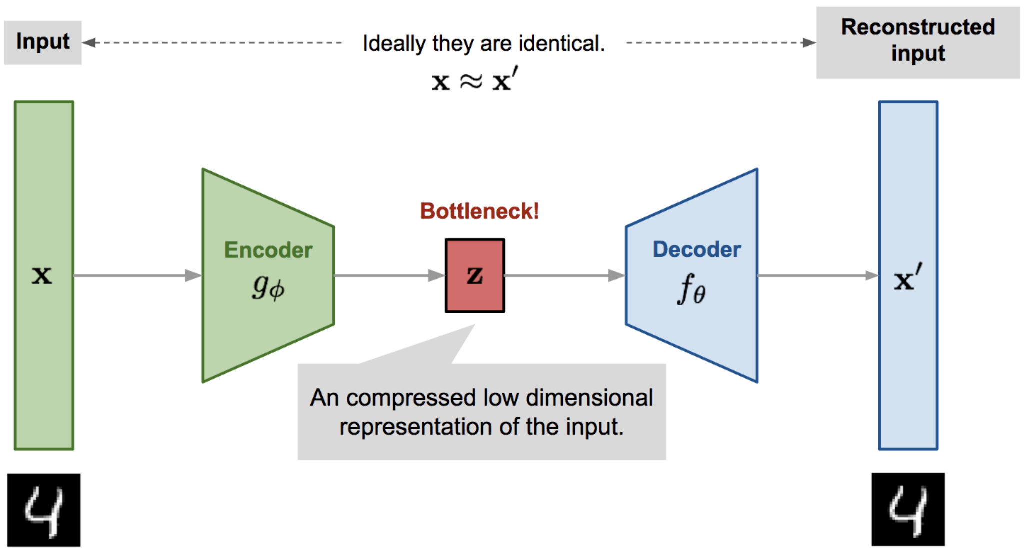

Input
Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Reconstructed input
$\mathbf{x}$
Encoder
g_ϕ
Bottleneck!
$\mathbf{z}$
Decoder
f_θ
$\mathbf{x}'$
An compressed low dimensional representation of the input.

Figure 3.1: The general pipeline of a VAE

(In our implementation, we ensure our encoder gives us an output of $\log \sigma^2$ instead of σ to simplify calculations and avoid non-linearities.)

Using the sample of $\mathbf{Z}$ (using the above trick, which is called the "reparameterization trick"), the decoder uses its CNN model to reproduce the desired image.

In the generation of images with many classes, we want to separate out different classes in the distribution space. To understand this, we plot the variation of each latent variable, as encoded by the encoder. A trained model, should start differentiating the classes as in Fig. 3.2. This model assumes a 2 dimensional latent vector for visualization clarity, but it need not be the case.

Further, I trained a model on the Fashion MNIST dataset which contains images of various clothing items like shoes, shirts, pants. I then created a map, where each grid-box contains an image generated by a sample of the distribution by varying the sample variance in 2 dimensions of the 10 latent variables. In a trained model (Fig. 3.6, the adjacent boxes of this map appear similar, but the gradual change across the map is apparent. On the top right corner of the variance map shows images of shoes, and the bottom left shows

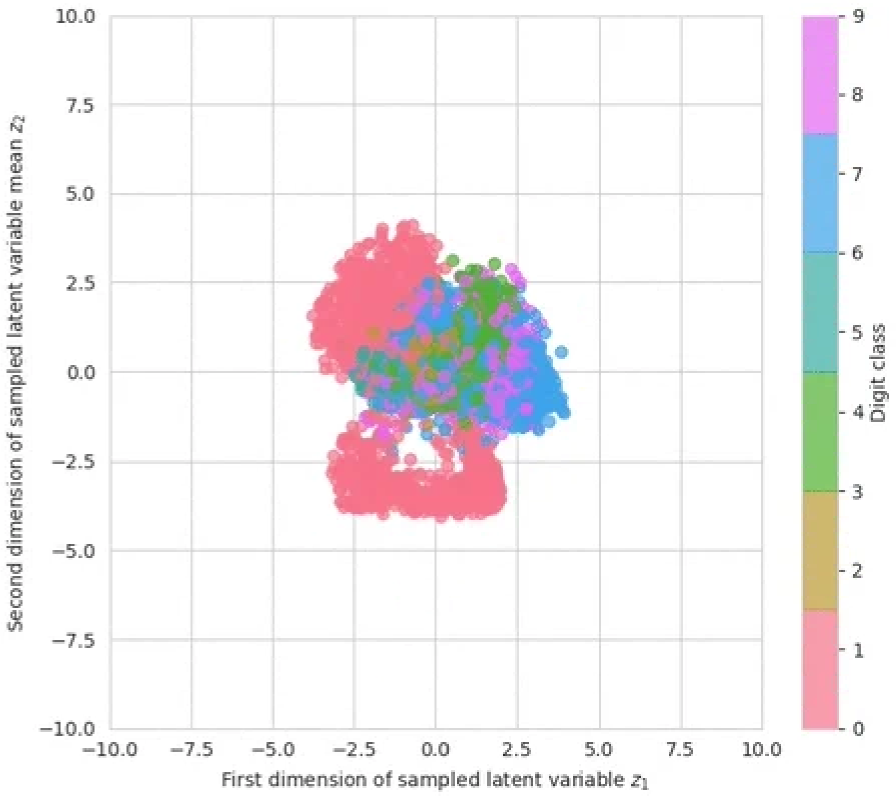


Figure 3.2: Class-wise separation of encoder output variation
Disclaimer, this image was taken from this link

images of shirts. Clearly, this difference has been consistently developing with training. Initially, no matter the distribution, all the images looked alike, but over time, these differences emerge. Similar variations occur at other pairs of dimensions as well.

Moving on to the loss function used.

At any given snapshot / instance of training, there is a fixed relationship between the latent vector $\mathbf{z}$ and the generated image $\hat{x}$. This is the decoder, and there are no stochastic layers in the decoder.

The source of the stochasticity is the red box in Fig. 3.1. This box takes in a distribution (further represented by a parameter λ), and generates a sample latent vector $\mathbf{z}$ from this multivariate (here Gaussian) distribution.

We want to ensure that there is least loss of information across this stochastic process. Since the parameters of the encoder and decoder are separately updated (in the sense that the update of a parameter 'a' assumes that another parameter 'b' ($b \neq a$) is fixed), they collectively minimize the loss function. This method is in contrast to the GANs, which have the generator and discriminator work against each other. This also ensures that we will find an optimum mappings of images to distributions, and from samples in this distribution to generated images.

Let us denote the distribution "encoded" by the encoder for an input image x by a placeholderλ. For the Gaussian distributions, $\lambda = (\mu, \Sigma)$[1]. The probability of a latent vector output from the red box given the distribution λ is denoted as:

$$q_\lambda(\mathbf{z}|x) = \frac{1}{\sqrt{2\pi \prod_i \sigma_i^2}} \exp\{\sum_{i=1}^{n} -\frac{(z_i - \mu_i)^2}{2\sigma_i^2}\} \tag{3.1}$$

Since the latent vector is a random vector following a certain distribution, the output of the decoder acting on these latent vectors ($f_\theta(\mathbf{z})$ is also a random image following another distribution). Assume there is a true

[1]The mean and Variance describe everything about the Gaussian distribution. Further, we assume that the components of the multivariate distribution are independent (covariance = 0), and replace the matrix Σ by a vector σ

probability distribution $p(\mathbf{z}|x)$ of latent-vectors (given input images) that generates images with a distribution similar to the input images set.[2].

Thus, the notation is summarized as:

- ϕ parameterizes the encoder mapping (as this mapping evolves with training)
- θ parameterizes the decode mapping (as this mapping evolves with training)
- x is an input image
- The encoder g_ϕ takes an input image x and computes a mean μ and (co-)variance Σ.
- λ parameterizes the Gaussian distribution. Thus, $\lambda = (\mu, \sigma)$
- $q_\lambda(y|x) = \mathcal{N}(y|g_\phi(x) = \lambda = (\mu, \sigma))$
- $\mathbf{z}$ is the latent variable, a single sample point drawn from the multivariate normal distribution q.
- The decoder f_θ takes the latent variable z and generates an image x'.
- $p(y|x)$ is the true desired distribution of the latent variable, at a fixed θ, or equivalently f_θ.

We want to find a distribution among our class of distributions (Gaussians with varying mean and variance) which most closely resembles this true distribution. Thus, we want to minimize the Kullback–Leibler (KL) divergence

[2]Ideally, these distributions would be the same for every input image of a certain class (the CMB dataset, or the Gaussian Blob dataset has only 1 class).

between $q_\lambda(\mathbf{z}|x)$ and $p(\mathbf{z}|x)$

$$
\begin{aligned}
D_{KL}(q_\lambda(\mathbf{z}|x) \mid\mid p(\mathbf{z}|x)) &\equiv \int_{\mathbf{z}} q_\lambda(\mathbf{z}|x) \log \frac{q_\lambda(\mathbf{z}|x)}{p(\mathbf{z}|x)} \, d\mathbf{z} && (3.2)\\
&= E_q\{\log \frac{q_\lambda(\mathbf{z}|x)}{p(\mathbf{z}|x)}\} && (3.3)\\
&= E_q\{\log q_\lambda(\mathbf{z}|x) - \log p(\mathbf{z}|x)\} && (3.4)\\
&= E_q\{\log q_\lambda(\mathbf{z}|x)\} - E\{\log \frac{p(\mathbf{z},x)}{p(x)}\} && (3.5)\\
&= E_q\{\log q_\lambda(\mathbf{z}|x)\} - E\{\log p(\mathbf{z},x)\} + \log p(x) && (3.6)
\end{aligned}
$$

Thus, minimizing $D_{KL}(q_\lambda(\mathbf{z}|x) \mid\mid p(\mathbf{z}|x))$ is equal to minimizing $E_q\{\log q_\lambda(\mathbf{z}|x)\} - E\{\log p(\mathbf{z},x)\}$.

We define the negative of this term as the Evidence Lower Bound:

$$\text{ELBO} = -(E_q\{\log q_\lambda(\mathbf{z}|x)\} - E_q\{\log p(\mathbf{z},x)\}) \tag{3.7}$$

$$\boxed{\text{ELBO} = E_q\{\log p(\mathbf{z},x)\} - E_q\{\log q_\lambda(\mathbf{z}|x)\}} \tag{3.8}$$

Thus, the loss used is ELBO loss.[3]

The reason this is called the Evidence Lower Bound is because we can rearrange terms in:

$$D_{KL}(q_\lambda(\mathbf{z}|x) \mid\mid p(\mathbf{z}|x)) = -\text{ELBO} + \log p(x)$$

As:

$$\log p(x) = \text{ELBO} + D_{KL}(q_\lambda(\mathbf{z}|x) \mid\mid p(\mathbf{z}|x))$$

If we think of:

[3] Calculated equivalently as:

$$
\begin{aligned}
\text{ELBO} &= E_q\{\log p(x|\mathbf{z})\} + E_q\{\log p(\mathbf{z})\} - E_q\{\log q_\lambda(\mathbf{z}|x)\} && (3.9)\\
&= E_q\{\log p(x|\mathbf{z})\} + E_q\{\log p(\mathbf{z})\} - E_q\{\log \frac{q_\lambda(\mathbf{z}|x)}{p(x|x)}\} && (3.10)\\
&= E_q\{\log p(x|\mathbf{z})\} + E_q\{\log p(\mathbf{z})\} - \text{Cross-entropy between x and x'} && (3.11)\\
& && (3.12)
\end{aligned}
$$

These are terms we can now easily compute.

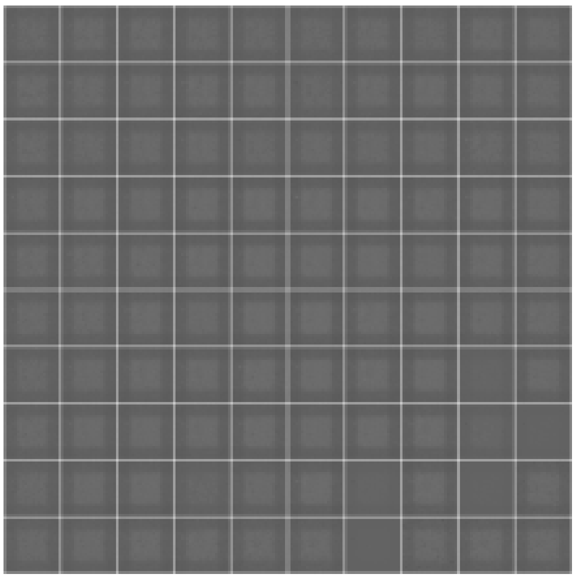


Figure 3.3: Untrained model map

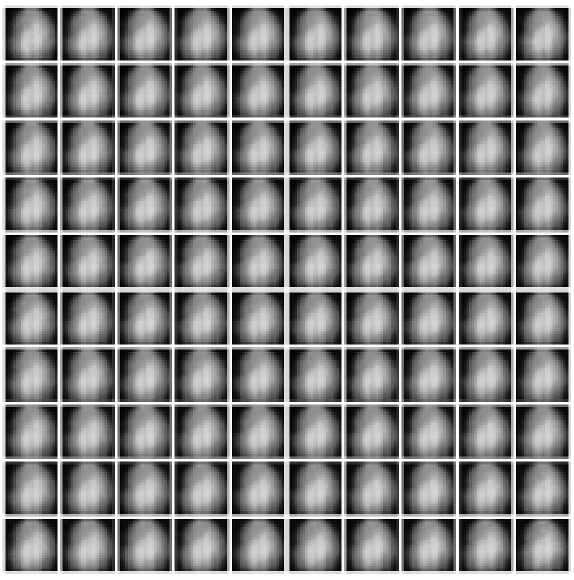


Figure 3.4: Training Epoch 10

- $p(x)$ as evidence ($\mathcal{Z}$). This is the true distribution of the data.
- $q_\lambda(\mathbf{z}|x)$ as posterior distribution ($\mathcal{P}$). Given data x, in the model λ, what is the probability of the sample $\mathbf{z}$. Since the input and model are connected deterministically, this may be equivalently seen as $q(\mathbf{z}|\lambda)$?
- $p(\mathbf{z}|x)$, which is again, equivalent to saying $p(\mathbf{z}|\lambda^*)$ as the prior on the model (Π)

this becomes:

$$\log \mathcal{Z} = \mathbf{ELBO} + D_{KL}(\mathcal{P} \mid\mid \Pi)$$

Which is recognizable as the Bayes' Theorem

To calculate the loss more efficiently, we use a single sample Monte-Carlo estimate of the expectation over the decoder. We do this believing that the huge number of iterations allows us to freely use the laws of large numbers. Thus, even if the model takes a few bad steps because of a bad estimate, we will ultimately converge to the optimal value. However, since we are no longer using the KL divergence, this loss need not be convex. There may exist non-global minima.

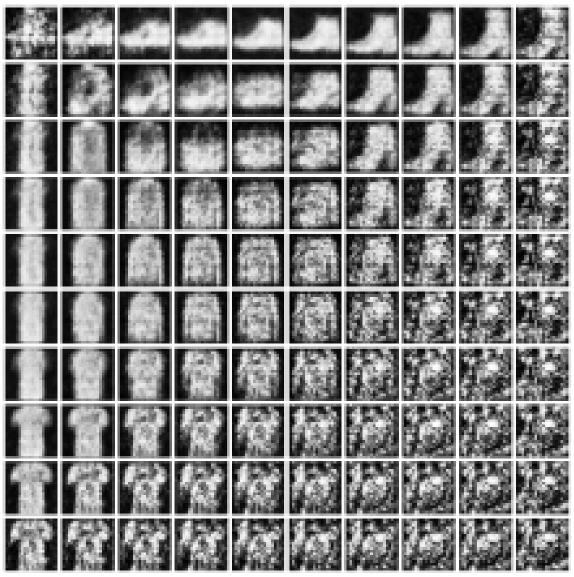


Figure 3.5: Training Epoch 20

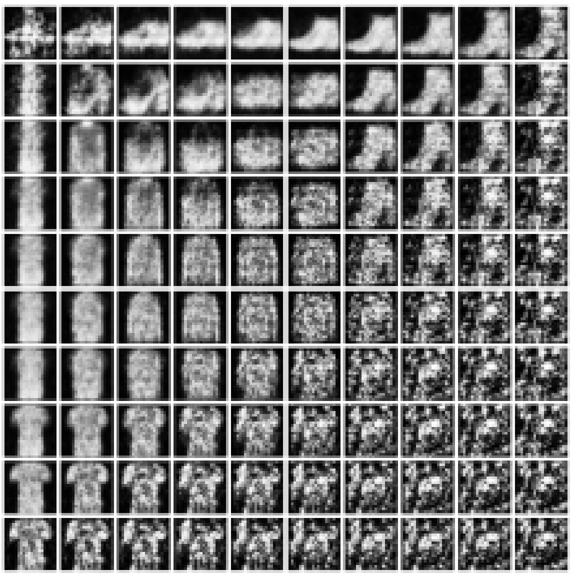


Figure 3.6: Training Epoch 30 (i.e., maximum training)

We now try to see the benefit of using a Gaussian distribution:
Representing $\mathbf{z}$ as the encoder output distribution of input image x.

$$\begin{aligned}
\log p(x|x') &= \log p(x' \sim z) && (3.13)\\
&= \log \left\{ \prod_i \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp \frac{-(x_i - \mu_i)^2}{2\sigma_i^2} \right\} && (3.14)\\
&= \sum_i \left[\frac{(x_i - \mu_i)^2}{2\sigma_i^2} - \frac{1}{2}\log(2\pi) - \log \sigma_i \right] && (3.15)
\end{aligned}$$

This term is clearly easier to compute and learn across, since it no longer contains log terms over the mean, or the decoder.

.

Despite this, VAEs do face some challenges - most notably *interference suboptimality* (i.e. faults caused due to premature stopping of the encoder). This manifests itself in the form of:

- Reduced expressiveness of the decoder due to limited variance of the latent variables (This is called **variational approximation**.)

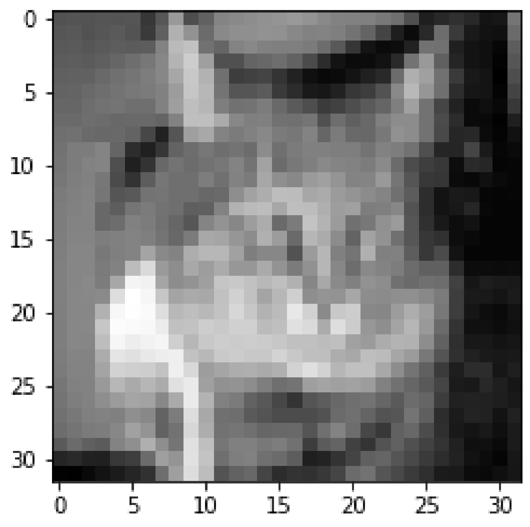


Figure 3.7: Original CIFAR10 image, which is not very clear.

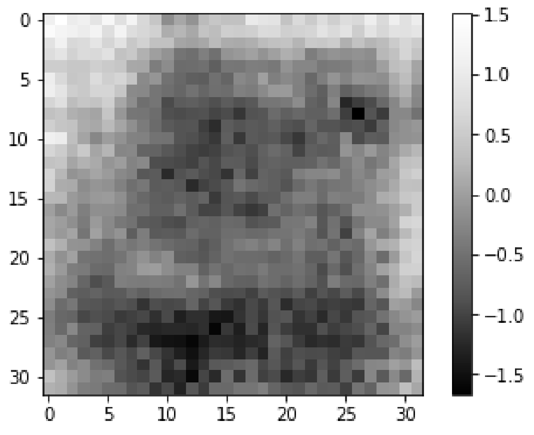


Figure 3.8: Generated image on the CIFAR10 dataset that may look like a small teddy-bear

- Missed discrepancy between the input and the generated images (This is called **KL Divergence approximation**.)
- Decoder becoming fixated on a small subset of the input data (This is called **model collapse** and may be caused due to small probability variations in the input data being blown up by the model, or other factors.)
- Latent space distortions/blurry images as seen in the training of a simpler model on the CIFAR10 dataset of objects from everyday life in Fig. 3.7, Fig. 3.8.

Another dataset this model was tried on was on the MNIST dataset (of handwritten digits). The generated and real images are as shown in Fig. 3.9 Fig. 3.10. Note that while neither of the Fashion-MNIST or MNIST images were taken after the model had completed training, these were taken after the images appeared to take the form they started resembling the original dataset. The CIFAR10 images were taken at convergence (i.e., end of model training).

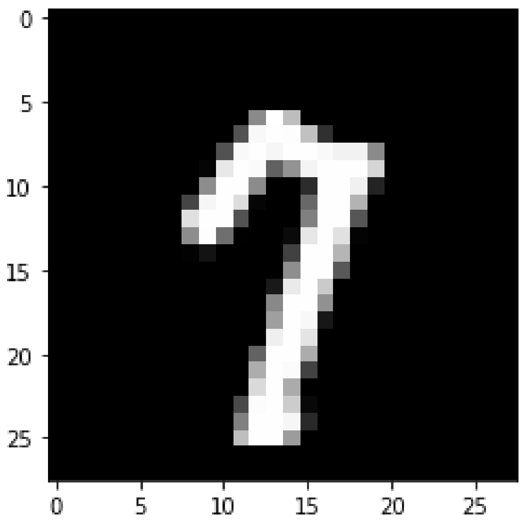


Figure 3.9: Original MNIST image

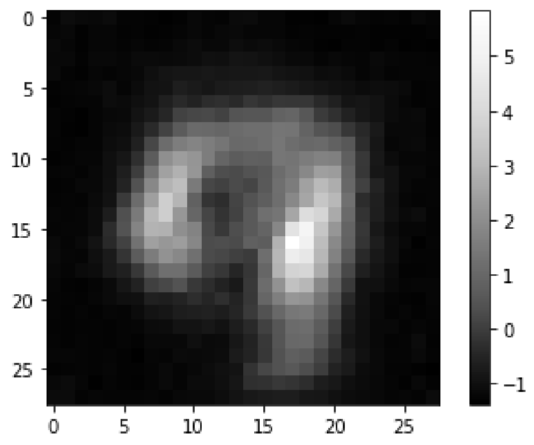


Figure 3.10: Generated image on the MNIST dataset that may look like a 9

Chapter 4

A Comparison of the Models on a Gaussian Blob Datasets

Thus, we come to compare the two models on two datasets we believe we understand well. These are:

- **Single Gaussian Blob**: There is a single Gaussian blob placed randomly on the image with its center lying anywhere on the image with a uniform probability. This can be thought of as a single point source convolved with a Gaussian point-spread function. Ideally, there would be 2 terms to encode the entirety of the image - the X and Y coordinates of the center of the blob.
- **Multiple Gaussian Blobs**: There are exactly 10 Gaussian blobs placed throughout the image (centers are again uniformly chosen), with some overlapping blobs causing distortions that may be interpreted as multiple separate circular blobs, or a single abnormally-shaped blob. Ideally, there would be 20 terms to encode the entirety of this image - the X and Y coordinates of the centers of all the blobs.

Neither of these models seemed to reach convergence (end of training), so instead, I decided to compare them at equal number of updates to the generative part of the model (generator for GANs and decoder for VAEs). This is different from saying the same number of training epochs because in the way the GANs are trained, the number of generator updates is not the same as that of decoder updates in VAEs for the same epoch. This limit was arbitrarily chosen at 250,000 updates. Further training will no-doubt make these models better, but this was chosen keeping in mind that this is not the final outcome of the project, but we still need to understand why the authors of the paper chose the model they did.

4.1 Model Architectures

The architecture used for the VAE (encoder and decoder connected) is shown in Fig. 4.1. Between the Single-Blob and Multiple-Blob architectures, we vary the length of the latent vector, meaning we alter the length of the mean vector, the variance vector and the sample on the right part of the image. It is 4x1 for the Single-Blob and 40x1 for the Multiple-Blob.

The Generator architecture used for the generator of the GAN is shown in Fig. 4.2. The same model was used for both the models, with the exception that the Single-Blob model starts with a 32-length vector instead of 256 as on the left of the picture, and the Multiple-Blob model does best with an inital 64-length vector.

4.2 Outputs

4.2.1 Single Gaussian Blob

The original input images to the models look similar to Fig. 4.3, but with the blob in different positions of the image.

The output of the VAE model with a latent-vector size of 4 is like in Fig. 4.4. Clearly, it has learnt to output single blobs, but there is some other cleaning up that is required. In addition, the learnt blobs look bigger than they should be in this image, and their sizes are not consistent across all generated images. Further, the model sometimes generates images which have many streaks across them as the single well-defined feature.

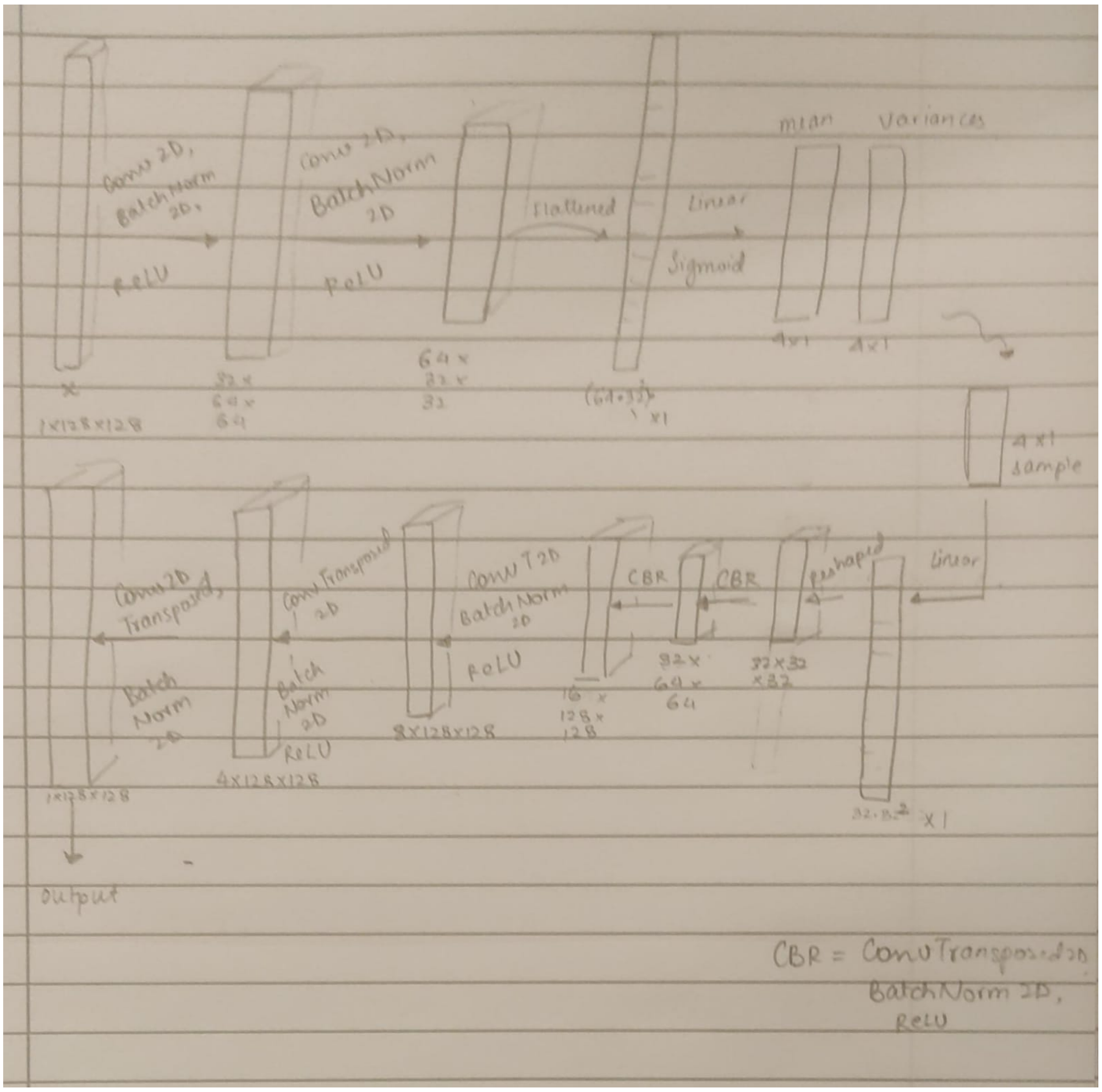

Conv 2D, BatchNorm 2D, ReLU
x
1x128x128
32x64x64
Conv 2D, BatchNorm 2D ReLU
64x32x32
Flattened
(64*32²) x1
Linear
Sigmoid
mean
variances
4x1
4x1
4x1 sample
Linear
32·32² x1
reshaped
32x32x32
CBR
32x64x64
CBR
16x128x128
ConvT2D BatchNorm 2D ReLU
8x128x128
ConvTransposed 2D Batch Norm 2D ReLU
4x128x128
Conv2D Transposed, Batch Norm 2D
1x128x128
output
CBR = ConvTransposed2D BatchNorm 2D, ReLU

Figure 4.1: Architecture of VAE used

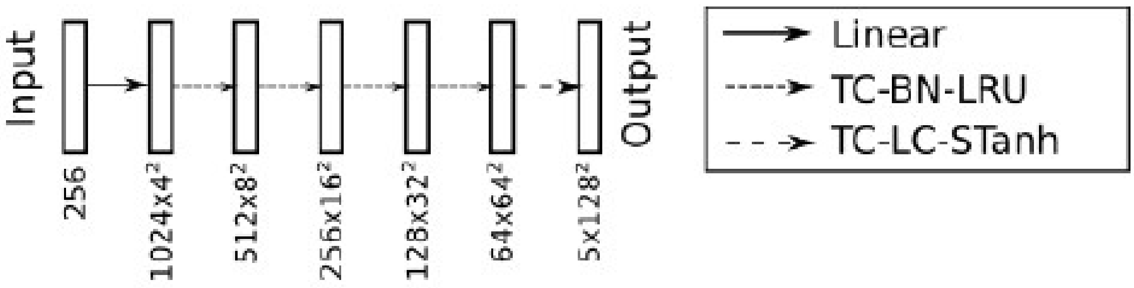


Figure 4.2: Architecture of GAN generator used.

The output of the GAN model is like in Fig. 4.5. Clearly, this has learnt a single blob, with a clear background, but with a more complex blob than expected. In addition, this blob looks bigger than expected in this image, and is also not consistent across the generated images.

4.2.2 Multiple Gaussian Blobs

The original input images to the models look similar to Section 4.2.2, but with the blobs in different positions of the image. We see in Fig. 4.7 how the image looks when the blobs are close together causing distorted single-blob like structures.

The output of the VAE model with a latent-vector size of 40 is like in Fig. 4.8. Clearly, it has learnt to output some blobs of varying intensities. They remain the same size across all images. However, they are more complex than expected, and their count also varies across images. Further, the model sometimes generates images which have many streaks across them as the single well-defined feature.

The output of the GAN model is like in Fig. 4.9. Clearly, this has learnt multiple blobs, with a clear background, but with a more larger blob than expected. In addition, the count of these blobs and their shapes is not consistent across all generated images.

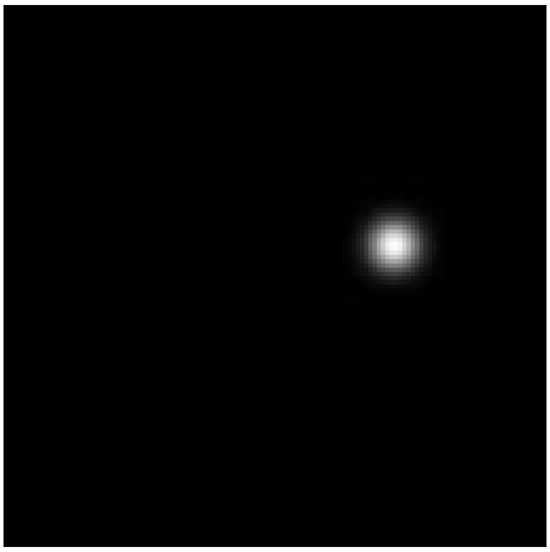
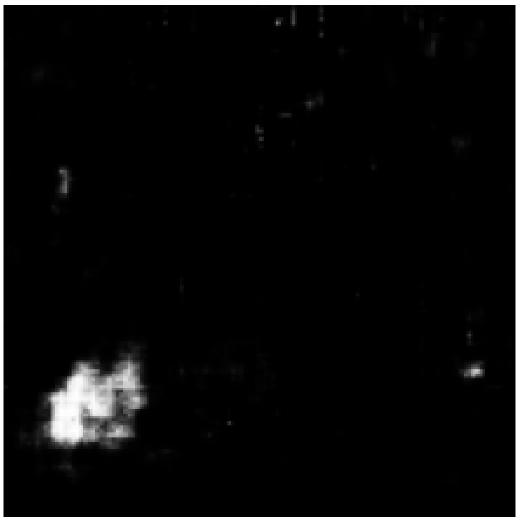


Figure 4.3: Example input image of a Single Gaussian Blob



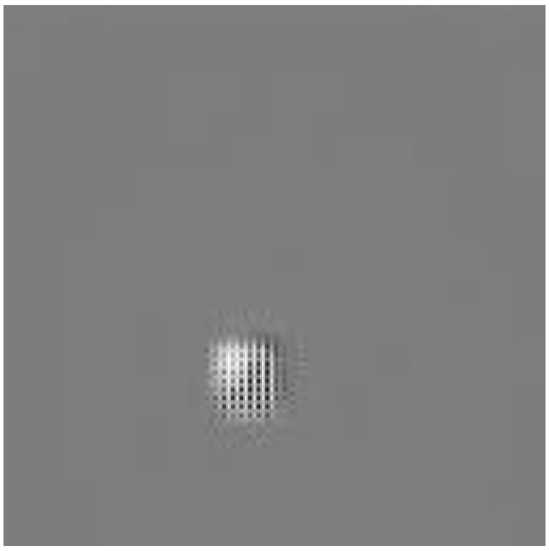


Figure 4.4: VAE generated image of a Single Gaussian Blob

Figure 4.5: GAN generated image of a Single Gaussian Blob

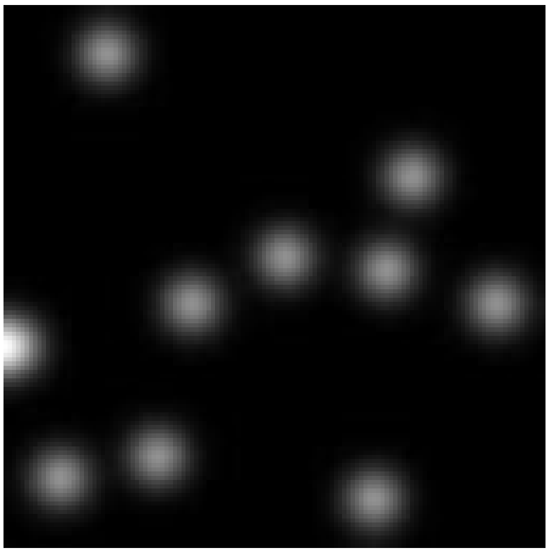


Figure 4.6: An example of multiple Gaussian Blobs where every blob is distinctly separate

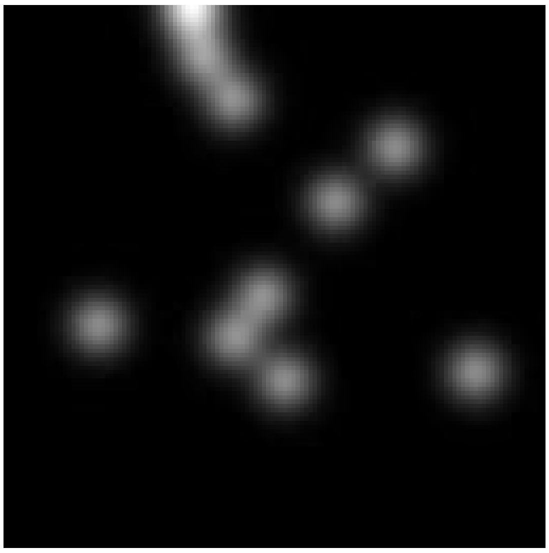


Figure 4.7: An example of multiple Gaussian Blobs where some blobs seem unresolved.

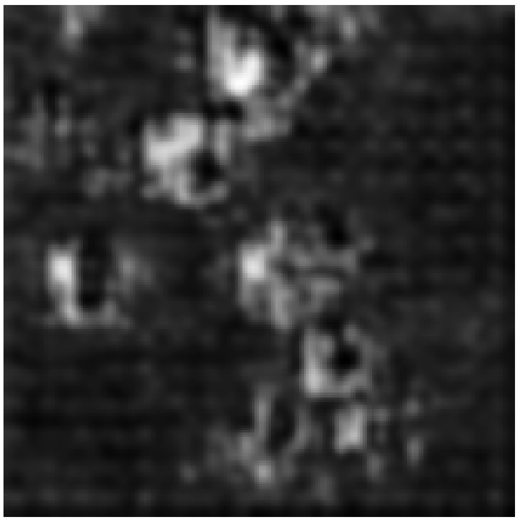


Figure 4.8: VAE generated image of a Multiple Gaussian Blob

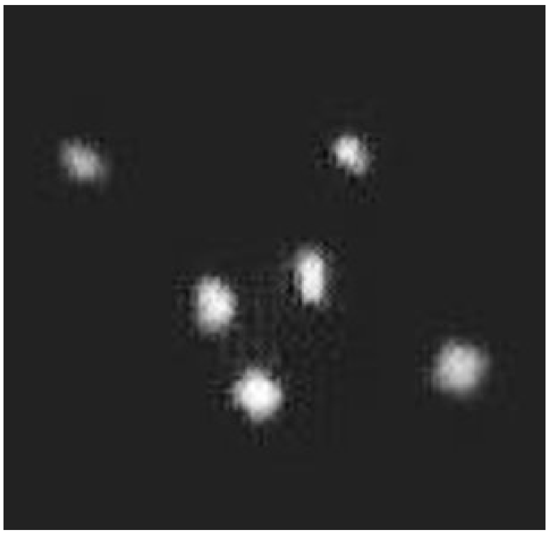


Figure 4.9: GAN generated image of a Multiple Gaussian Blob

Bibliography