GAN vs VAE: A Comparison of Deep Generative Models

Arpan Banerjee
9359-9083
CISE Department
University of Florida
arpanbanerjee@ufl.edu

Abstract

Generative models have seen tremendous success in recent years across a wide range of problem domains mainly involving images and speech. In this paper, we compare two of the most popular generative models, Generative Adversarial Networks (GANs) and Variational Autoencoders (VAEs). We work with the MNIST dataset of handwritten digits and explore different configurations of these models which allow them to train successfully.

1 Introduction

Generative models belong to the unsupervised learning class of machine learning models. They aim to learn the true distribution of the sampled data allowing them to generate brand new examples. This is a much more difficult task than supervised learning where we are provided with labels and must fit weights. Generative models try to model a distribution that is similar to that of the input data either implicitly or explicitly. Two of the most popular generative models are Variational Autoencoders (VAE) and Generative Adversarial Networks (GAN) and they differ slightly in what they are traditionally used for.

1.1 Variational Autoencoders

An autoencoder comprises of an encoder and a decoder. The encoder is used to encode data, such as images, into a much smaller dimensional space called the latent variable space. These latent variables can then be mapped to the original input image using the decoder. A vanilla autoencoder, however, doesn't learn the distribution of the data and can't be used to generate brand new examples.

A variational autoencoder is commonly used to learn the distribution. It is a probabilistic model which uses Bayesian inference and it's explicit goal is latent variable mapping. The main idea is to connect the latent variables to a probability distribution such as a Gaussian. The encoded latent variables can then be used to estimate the parameters of the Gaussian distribution using likelihood maximization allowing us to fit it to the input distribution. In the inference stage, this learned Gaussian can be sampled to yield new latent vectors which can then be passed through the decoder to generate completely new samples.

1.2 Generative Adversarial Networks

GANs use a clever technique called adversarial training to turn the unsupervised learning problem into a supervised scenario. It comprises of two sub-networks, the Generator and the Discriminator. The generator model is used to generate new examples. Its input is sampled from a latent probability distribution. The discriminator's job is to classify examples as real (from sample data) or fake (created by the generator).

Training a GAN is tricky and involves several parts. It is a two player game where both the generator and discriminator models are trained together. The discriminator is trained using some samples from the training set and some samples created by the generator. It is then updated like a usual classification model. The generative model is also updated based on how well the discriminator was able to classify its images as fake (generated).

This is like a zero-sum game where if either of the two components gets too powerful, the entire training can fall apart. For example, if the discriminator is too good at the start, it will correctly classify all generated images as fake before the generator has a chance to learn what kind of images can fool the discriminator. One must ensure that both the models train at a similar rate so that they can stay competitive.

2 Models

Two variations each of GAN and VAE were implemented and tested, one with linear layers and the other with convolutional layers. Some of the nomenclature used is *Linear* - Fully connected layer, *Conv2D* - 2D convolutional layer, *TConv2D* - 2D transpose (fractionally strided) convolutional layer.

2.1 Linear GAN

Generator			**Discriminator**		
LAYER TYPE	INPUT DIM	OUTPUT DIM	LAYER TYPE	INPUT DIM	OUTPUT DIM
Linear LeakyReLU (negative slope = 0.2)	100	128	Linear LeakyReLU (negative slope = 0.2) Dropout($p = 0.4$)	784	512
Linear LeakyReLU (negative slope = 0.2)	128	256	Linear LeakyReLU (negative slope = 0.2) Dropout($p = 0.4$)	512	256
Linear LeakyReLU (negative slope = 0.2)	256	512	Linear LeakyReLU (negative slope = 0.2) Dropout($p = 0.4$)	256	128
Linear Tanh	512	784	Linear	128	1

2.2 DCGAN (Deep Convolutional GAN)

Generator					**Discriminator**				
Layer	Input	Output	Kernel	Stride	Layer	Input	Output	Kernel	Stride
Linear	100	12544			Conv2D LeakyReLU (negative slope = 0.2) Dropout ($p = 0.25$)	1	32	3	2
TConv2D ReLU	256	128	3	2	Conv2D LeakyReLU (negative slope = 0.2) Dropout ($p = 0.25$)	32	64	3	1
TConv2D ReLU	128	64	3	1	Conv2D LeakyReLU (negative slope = 0.2) Dropout ($p = 0.25$)	64	128	3	1
TConv2D ReLU	64	32	3	1	Conv2D LeakyReLU (negative slope = 0.2) Dropout ($p = 0.25$)	128	256	3	2
TConv2D Tanh	32	1	3	2	Linear	12544	1		

2.3 Linear VAE

Encoder			Decoder		
LAYER TYPE	INPUT DIM	OUTPUT DIM	LAYER TYPE	INPUT DIM	OUTPUT DIM
Linear ReLu	784	512	Linear ReLU	16	128
Linear ReLu	512	128	Linear ReLU	128	512
Linear	128	32	Linear Sigmoid	512	784

We can notice a difference between the output dimension of the last encoder layer (32) and input dimension of the first decoder layer (16). This is because half of the features from the encoder are used for the mean of the Gaussian latent space and the other half is used for variance.

2.4 Convolutional VAE

Encoder					Decoder				
Layer	Input	Output	Kernel	Stride	Layer	Input	Output	Kernel	Stride
Conv2D LeakyReLU (negative slope = 0.2) Dropout ($p = 0.25$)	1	32	3	2	Linear	16	12544		
Conv2D LeakyReLU (negative slope = 0.2) Dropout ($p = 0.25$)	32	64	3	1	TConv2D ReLU	256	128	3	2
Conv2D LeakyReLU (negative slope = 0.2) Dropout ($p = 0.25$)	64	128	3	1	TConv2D ReLU	128	64	3	1
Conv2D LeakyReLU (negative slope = 0.2) Dropout ($p = 0.25$)	128	256	3	2	TConv2D ReLU	64	32	3	1
Linear	12544	32			TConv2D Sigmoid	32	1	3	2

3 Experiments

As this is unsupervised learning, we do not have metrics to measure how good the results are. We can only compare the results by eye and discuss.

3.1 GANs

Training of GANs is very sensitive to hyperparameters. These were the hyperparameters used for both the linear and convolutional GANs.

- Optimizer: Adam
- Learning rate: 0.0002
- Batch size: 128
- Epochs: 100

As we can see in Fig 1 and Fig 2, we managed to train both the components of the GAN at a similar rate maintaining competitiveness between the two. The discriminator loss does start to grow towards

Figure 1: Linear GAN losses

Figure 2: DCGAN losses

Figure 3: Linear GAN generated samples over epochs

the end possibly indicating that some parameters of the discriminator could be tuned for better performance.

Figure 3 shows some samples generated by the Linear GAN as it trains. We notice a consistent improvement over epochs. However the digits are quite noisy and discontinuous with black and white patches around the digits. Linear models do not show great results with images so this kind of result is expected.

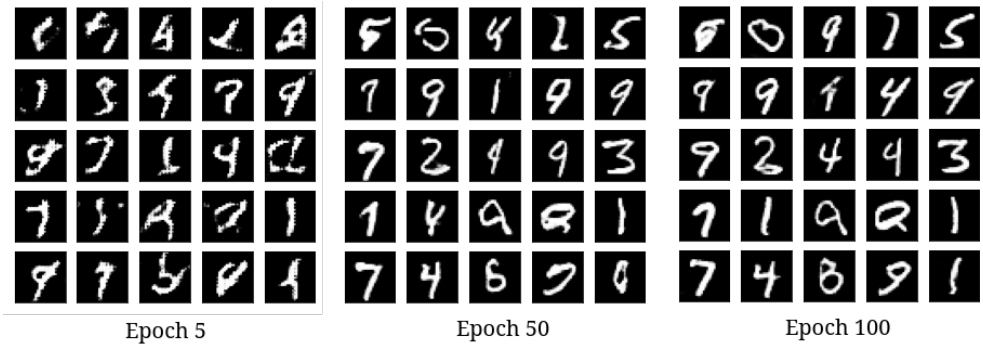

Figure 4: DCGAN generated samples over epochs

In figure 4, samples generated by the DCGAN are demonstrated. Looking at the epoch 50 and 100 results, there does not seem to be much improvement. This is possibly due to the generator and discriminator losses diverging indicating some more parameter tuning may be required. We also notice some distribution sampled inputs being mapped to different digits such as the second column in the fourth row, which is a 4 in epoch 50 and a 1 in epoch 100. The probability space is being continuously remapped over epochs as the generator trains.

Overall, the digits generated by the DCGAN are much smoother and better than the LinearGAN digits. Convolutional layers allow the model to learn spatial features much better, which is probably why we don't see the noisy patches in DCGAN images.

3.2 VAEs

In the case of VAEs, we can actually use a kind of supervised learning by computing the binary cross entropy loss of the reconstructed image against the original image as a target. This allows us to compute validation loss and detect overfitting of any kind.

The hyperparameters used for training the VAEs were

- Optimizer: Adam
- Learning rate: 0.0001
- Batch size: Linear VAE - 64, CNN VAE - 128
- Epochs: 100

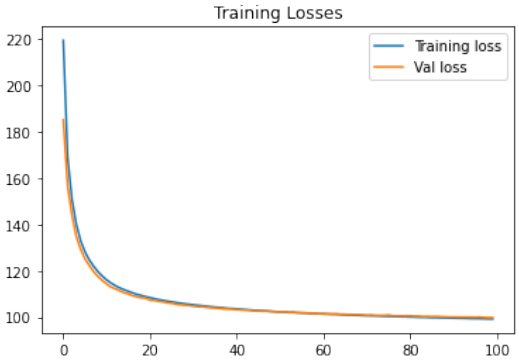

Figure 5: Linear VAE loss

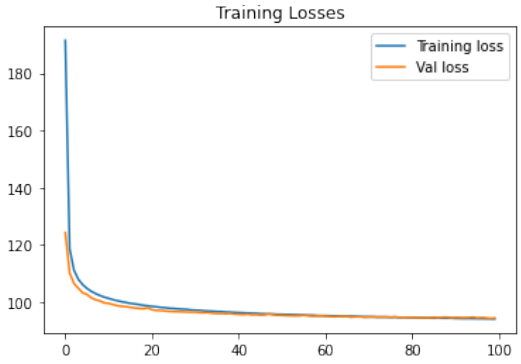

Figure 6: Convolutional VAE loss

Figures 5 and 6 show the loss of the two VAE models across epochs. They indicate that the model learns well and does not show any overfitting. There is also not much change in the losses after about 50 epochs which could mean that the models have already converged.

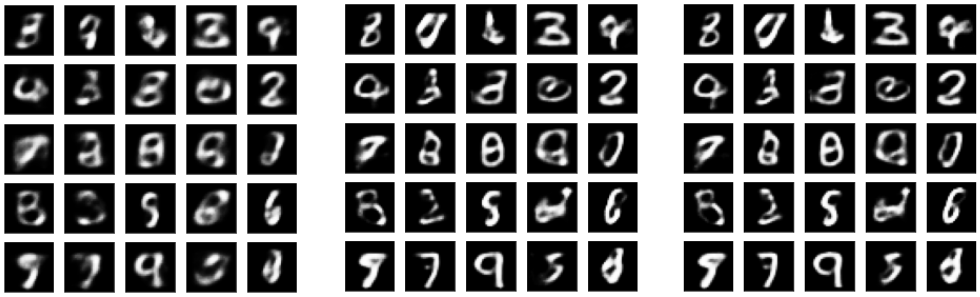

Figure 7: Linear VAE generated samples over epochs

Figure 7 shows the samples generated by the linear VAE across epochs. In epoch 5, the digits are very blurry but some of them can be made out. By epoch 50, the model has converged as there is barely any difference with the image at epoch 100. The model is able to learn a few digits but the results are very blurry.

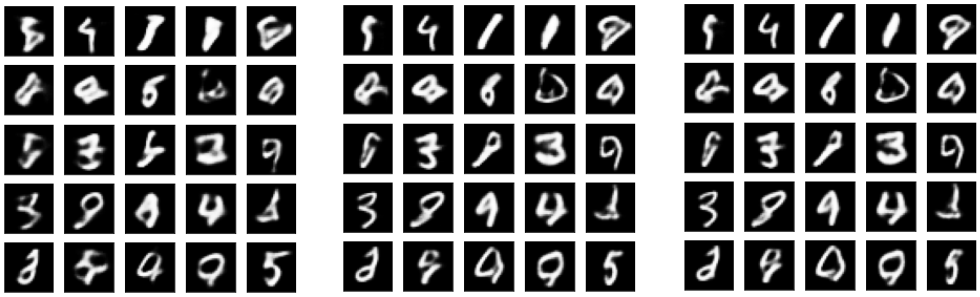

Figure 8: Convolutional VAE generated samples over epochs

The results for the convolutional VAE are demonstrated in figure 8. Like the linear VAE, it also converges around epoch 50 and there is not much improvement after that. Compared to the linear VAE, the images are less noisier but are still quite blurry. It also seems to have been able to map more digits into its latent space. Overall, though, there is not much improvement on adding convolutional layers to the VAE.

4 Conclusion

The best results were produced by DCGAN among all the models we tested. The images were sharp and not noisy. We also observed that linear models lead to noisy results in the case of both GANs and VAEs. When convolutional layers are added, they lead to smoother results and continuous digits instead of broken up ones as they learn spatial features better.

The GAN outperforms the VAE and the difference is most evident in the convolutional case. The CNN VAE produces blurrier images hinting that it has difficulty in learning edges. Some of its produced digits can also be confused with one another. For example: Some of the 1s and 4s and 9s are very similar, also 0 and 6. Some of the digits may also be transitions between two digits as the distribution is continuously mapped and the output digits come from randomly sampled points from the distribution.

Code

The code for this project has been written using Jupyter notebook and can be found on Github at `https://github.com/arpan98/mnist-generative-models`. All the models have been implemented using PyTorch in Python and run on Google Colab GPUs.

References

[1] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville & Y. Bengio (2014) “Generative Adversarial Networks”.

[2] Y. LeCun and C. Cortes, (2010) “MNIST handwritten digit database”

[3] *towardsdatascience.com*