Comparison of DCGAN, CapsuleGAN and Variational Autoencoder for Image Generation

V V Sreerajatha, Akanksha Gupta, Naveen Murali

Northeastern University, Boston, Massachusetts, USA
sreerajatha.v@northeastern.edu, gupta.aka@northeastern.edu, murali.n@northeastern.edu

Abstract

Image generation is a relatively new and complex problem in deep learning. There are currently several techniques used for this task, so it becomes increasingly important to find processes that are both fast and yield the best result. This project aims to compare the results of Variational Autoencoders (VAE), Deep Convolutional Generative Adversarial Networks (DCGAN) and Capsule Generative Adversarial Networks (CapsuleGAN) in generating images of handwritten digits from the MNIST dataset. Their performance is evaluated via loss computation and the generated images are compared after a certain fixed number of epochs.

Introduction

Deep generative models and Variational autoencoders have widely been used because of their ability to learn the latent features represented in the images as well as generate realistic images. Since 2012, when CNNs showed significantly better performance than any other previous methods for image classification in ImageNet, they have been utilized widely and have become a standard model for image classification. As the main approach behind this paper is Image Generation, which is a subcategory of Computer Vision, it can be achieved by using Neural networks and CNNs. This has been incorporated in Generative models that use semi supervised learning approaches like Variational Autoencoders or Generative Adversarial Networks (GANs).

VAEs are simple yet sophisticated networks that have been proven to generate complex images. They are popular models that make use of probability distributions for generation.

GANs were first introduced in 2014 and grew widely popular for generating images that share the same distribution as the real data. It could be used for unsupervised and semi supervised learning. For instance, a GAN model is originally composed of 2 neural networks: 1) Generator and 2) Discriminator that are trained by playing an adversarial game where they find a Nash equilibrium between the generator and discriminator. This concept showed great promise as traditional deep generative models are constrained by multiple factors like approximation inference and Markov chains while GANs require only backpropagation to obtain gradients. However, on their own, they are difficult to train on complex datasets as generator must be synchronized well with the discriminator during training, otherwise it would collapse and result in instability. Some research papers have provided a set of architectural guidelines while building CNNs that have been extensively used to create DCGANs.

DCGANs focus on using Deep Convolutional networks and find areas of correlation within an image, that is, they look for spatial correlations. This means that a DCGAN fits better for image/video data, whereas a general GAN can be used in wider domains.

However, more recent papers have introduced Capsule Networks (CapNets) as a more promising solution. In the CapNets training algorithm, the training involves replacing the scalar output feature detectors with vector-output capsules and a routing-by-agreement mechanism between capsules of successive layers. In CapsuleGAN, we implement CapNets framework for the discriminator structure. Results show that CapsuleGANs outperform other CNN-based GANs and VAEs and yield better qualitative and quantitative results on the MNIST dataset.

Background

Variational autoencoders (VAE) are built off a type of neural network called autoencoders. Autoencoders transform a vector of a certain size down to a vector of a smaller size. This part of the network is called the encoder. The transformed vector is called the latent vector. This latent vector is passed to the decoder network that then reconstructs the original vector.

The encoder and decoder are separate neural networks which are trained to reconstruct the vector with minimum loss. VAEs build off this by making use of a similar structure. They too have an encoder and decoder network, however in VAEs the encoder does not output a regular vector which is passed to the decoder. Instead, the encoder gives a distribution based on the inputs given to it. It is a normal gaussian distribution. Samples are taken from this and passed to the decoder which then reconstructs the vector. Once the VAE has been trained, samples can be taken randomly from this distribution to generate images that were not a part of the training dataset.

Convolutional Neural Networks, or CNNs, are designed to map image data to an output variable. A CNN is a neural network that has one or more convolutional layers. A convolution is essentially represented as sliding a filter over the input and producing an activation at every slide position. These activations produce a feature map, which represents how much the data at that region contributed towards building the output. This is a special type of neural network that is designed for data with spatial structure. In short, CNNs can detect image features and classify the existence of an object by feeding this knowledge forward.

However, it performs better in detecting features and keeps less track in the spatial aspect of the features present in the image. The knowledge acquired earlier about the geometrical relationship needs to be encoded into the neural network. Basic concept during training a CNN is that higher level features are obtained by combining lower level features as a weighted sum (sum of each activation of the previous layer multiplied by the subsequent layer neuron's weight) and then passed through a nonlinear activation function. A GAN exploits this network extensively by arranging CNN layers in a minimax game. Out of the two neural networks in GAN, the generator creates new data from a noise vector and is trained using the other network, discriminator, which distinguishes between generated and real data. The generator learns to produce realistic images through this adversarial process, and the discriminator learns to distinguish real vs. fake, until the latter can no longer make confident distinctions. Both the models are trained simultaneously.

DCGAN combines the two algorithms, by implementing Deep Convolutional layers in a GAN architecture. While the generator of the simple GAN is a simple fully connected network, DCGAN uses the dense layers and transposed convolution technique to perform up-sampling and down-sampling of 2D image size. We can roughly consider the transposed convolution as a reverse operation of normal convolution. The generator network generally uses one or more dense layers, followed by transposed convolutional layers to produce an image in the desired shape. The generated images when passed to the discriminator along with the real images, the discriminator identifies if the given images are real or fake. The discriminator architecture is generally defined using only the convolution transpose layers lastly followed by a dense layer to yield a probability that the image is real.

One disadvantage is that CNNs keep only most important features from the lower level while making higher level features, using Max Pooling, which also means that they may lose important spatial relationships between features. CapNet architecture has a potential to take care of this by learning to represent the images in an equivariant manner so as to not lose information regarding the spatial relationships of parts of objects in the image.

Related Work

Traditionally GANs have been implemented as multi-layer perceptron with a feedforward network. They did not perform well with complex datasets like images or videos. There are many studies on using unsupervised learning for computer vision research. In the context of images, one can also do a hierarchical clustering of image patches in order to learn the feature representations better.

Another method that has been implemented in this paper and that provides powerful image representation is Variational Autoencoders. In this implementation, dense layers are used to create the encoder and decoder networks; the network can be improved by using CNNs.

Deep Belief Networks (DBNs) have also been shown to perform well in hierarchical representations but have not been discussed further here as they have other computational complexities associated with it. In terms of constructing generative image models, methods like incorporating Recurrent Neural Networks as generators instead of CNNs in GAN architecture giving rise to Generative Recurrent Adversarial Networks (GRANs) have shown good results. Additionally, another approach that incorporates Laplacian pyramid extension into a conditional GAN have resulted in good quality images too but have some unsteadiness in the objects as a result of noise in multiple chaining of models. Here, we compare three well performing models VAEs, DCGANs and CapNets both quantitatively and qualitatively.

Project Description

Dataset

The MNIST dataset has been used to train all the models and compare results. This dataset contains 60000 black-white, 28x28 pixel, labeled handwritten digit images. All numbers between 0-9 are represented and hence they have 10 distinct classes.

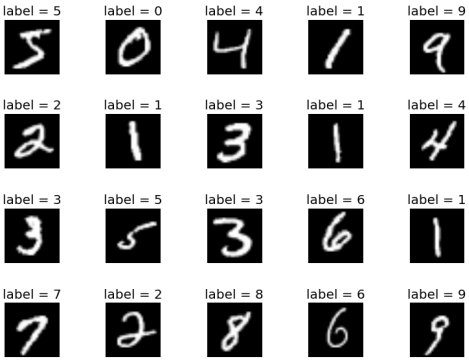

Fig 1. Example of an image from MNIST Dataset

Variational Autoencoders (VAE)

As mentioned before VAEs consist of two distinct networks, an encoder and a decoder. In the training phase, both networks are trained. The encoder is tasked with forming a distribution given the image input and the decoder is tasked with reconstructing an image after an input for it has been sampled from the previous distribution. Once trained, only the discriminator is required for image generation. Since the distribution that the encoder creates is a simple gaussian normal distribution with a mean and standard deviation, any inputs sampled from this will yield a valid image from the decoder.

Since the encoder converts the image to a latent distribution, the input layer has a size of 784. Each part of the input represents a pixel of an image in the MNIST dataset. The encoder then passes the information through its network. The hidden layers are constructed to accomplish this. The output of the encoder consists of two layers, both the same size. These layers represent the mean and standard deviation of latent distribution.

The decoder must take a value from this latent distribution. So, its input layer's size is the same as the latent distribution layer's size. The hidden layers of the decoder then build up such that the final output layer represents the reconstructed image, so it has a size of 784.

The VAE is formed by connecting the output of the encoder to the input of the decoder through a sampling layer. The distribution that is formed represents an expectation of z given an image x. So, the decoder requires a sample from this z in order to reconstruct an image x'. Instead of directly sampling from the distribution, which is not possible, the reparameterization trick is used. Now instead of sampling from z directly, samples are taken from a unit normal distribution called epsilon(ϵ). The sample to the decoder is then computed as

$$z = \mu + \sigma * \epsilon$$

μ is the mean of the distribution

σ is the standard deviation

ϵ is the sample taken from the unit normal distribution

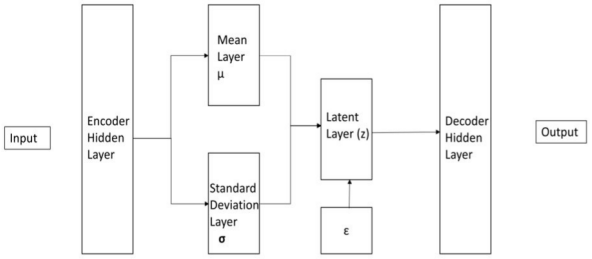

Fig 2. Architecture of a variational autoencoder

In order to train this VAE, the loss must be calculated. The loss function can be divided into 2 parts. The reconstruction loss, which is how close the generated reconstructed image is to the original image and the KL divergence, which is how close the learnt distribution is to a unit normal distribution. This ensures that the encoder is optimized to create a distribution that is close to a unit normal one and the decoder is trained to create images that are close to the original training images.

Reconstruction loss is the binary cross entropy loss and the KL divergence is

$$\text{KL Loss} = -0.5 * \sum (1 + \sigma - \mu^2 - e^{\sigma})$$

The total loss is the sum of reconstruction loss and KL divergence.

DCGAN

DCGAN comprises of two networks, generator and discriminator. Generator has been trained on a 100-dimensional normal distribution with one dense layer of 12544 units and three transposed convolutional layers of 128, 64 and 1 units respectively. Striding has been used in each convolutional layer and additionally, padding has also been used, in order to keep the dimensions of each convolutional layer's output image same as the dimensions of the layer's input image. Each layer uses batch normalization and leaky rectified linear activation (LeakyRelu), except for the last layer, which uses hyperbolic tangent activation and no normalization.

The discriminator has been trained simultaneously using the real images from the dataset and the fake images generated by the generator. The discriminator network has a similar structure as the generator but reversed. It replaces the number of units in the dense layer with a single unit. Each

layer in the discriminator uses leaky rectified linear activation and dropout. The architecture is modeled in Fig 3.

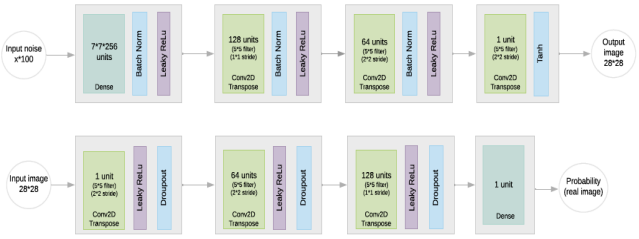

Fig 3. DCGAN model architecture, generator (top) and discriminator (bottom)

The model was trained by sampling one minibatch of real images through the discriminator and generator and then computing binary cross entropy losses for both networks for each sample and then backpropagating. Given datapoints x from the target distribution generated by the generator G, and z from the noise distribution, the loss function for the discriminator network D, is given by,

$$L_D(x, z) = \log(D(x)) + \log(1 - D(G(z)))$$

and the loss function for generator is given by,

$$L_G(z) = \log(1 - D(G(z)))$$

In each training step, noise is fed as input to the generator which outputs an image. Discriminator takes the generated image and real image dataset to emit a probability how real the generated image is. Losses are calculated thereafter for both the models which are used to update gradients for generator as well as discriminator. The model learns with each training step and eventually begins to converge.

CapsuleGAN

Capsule Networks

Originally introduced in a paper by Hinton et al., capsules are group of neurons that capture the presence of a feature entity and also represents the properties of those feature entities in vector outputs. The length of the vector shows the probability of feature detection and, its spatial representation is denoted as the direction the vector is pointed to. Therefore, a nonlinear squashing function is used for discriminator learning. This function ensures that shorter vectors get reduced to almost zero length and longer ones to just below one.

When a feature is detected and its position with respect to an image changes, the length of the vector would remain the same and the direction would only change. Hence, as mentioned before, the equivariance of features makes it possible to transform the detected objects to one another easily. Each capsule would be trained to identify certain objects in the image. Hence while building the layers for neural networks, each layer can comprise of multiple capsules forming a capsule-layer and giving a vector representation for the output. The higher layers receive inputs from the lower capsule layers. If $\boldsymbol{u_i}$ is the vector received from the lower levels (previous capsule) and $\boldsymbol{W_{ij}}$ (transformation matrix) is the corresponding weight matrices, then multiplying them would result in encoding important spatial and other relationships between lower and higher-level features.

$$\hat{\mathbf{u}}_{j|i} = \mathbf{W}_{ij}\mathbf{u}_i$$

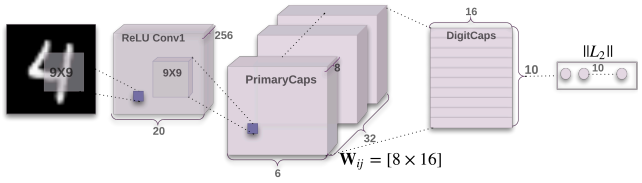

Fig 4. A CapNet architecture showing the PrimaryCaps Layer according to Sabour et al. This architecture has been implemented for the discriminator network. The input image maybe from the generator or an image from the real dataset.

Incorporating Capsule Networks in GANs for Discriminator Structure

A basic 28x28 pixel image is given as input to the model. First step is a simple convolutional layer with 256 filters, 9-pixel kernel size and a stride value of 1. This layer is followed by LeakyRelU activation and Batch Normalization to ensure better performance. The non-linearity of the activation is added is here in order to ensure that there is no vanishing gradient. This layer, before the CapNet architecture begins, converts the pixel data, captures all the local features and then passes it on to the PrimaryCaps (the first layer in the capsule network). This part is a combination of multiple layers of Convolution, Reshape and Squashing functions. At this point the capsule architecture shows that the convolution is an 8D image vector of 32 feature maps, representing all the 256 filter of the previous layer and reshape would split all the 32 neurons into 8D vectors.

$$\mathbf{v}_j = \frac{||\mathbf{s}_j||^2}{1 + ||\mathbf{s}_j||^2} \frac{\mathbf{s}_j}{||\mathbf{s}_j||}$$

where $\mathbf{v}_j$ is the vector output of capsule j and $\mathbf{s}_j$ is its total input

Fig 5. Squashing function equation

The second part of the Capsule network comprises of DigitCaps Layers which incorporates the Routing-by-agreement algorithm. This layer receives its input first from the flattened output of the PrimaryCaps. The first step is that this output is passed through a Keras Dense layer which acts as the prediction vector, $\hat{\boldsymbol{u}}$. The output vector from the Dense layer outputs 160 neurons and is passed through 3 routing iterations. This implies that the information is passed between capsules of successive layers. Hence, for the information in capsule at layer l, $\boldsymbol{h_i}^{\boldsymbol{l}}$ and each capsule in the layer above, $\boldsymbol{h_j}^{\boldsymbol{l+1}}$, a coupling coefficient $\boldsymbol{c_{ij}}$, which is a softmax over the bias weights of the previous layer and $\hat{\boldsymbol{u}}$, is computed iteratively. Each iteration is followed by a

LeakyRelU activation. The final layer is a Keras Dense layer with a sigmoid activation and a single neuron. This will output a score that determines how close the generated output is to the actual image.

Procedure 1 Routing algorithm.

1: **procedure** ROUTING($\hat{\mathbf{u}}_{j|i}, r, l$)
2: for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow 0$.
3: **for** r iterations **do**
4: for all capsule i in layer l: $\mathbf{c}_i \leftarrow \texttt{softmax}(\mathbf{b}_i)$
5: for all capsule j in layer $(l+1)$: $\mathbf{s}_j \leftarrow \sum_i c_{ij}\hat{\mathbf{u}}_{j|i}$
6: for all capsule j in layer $(l+1)$: $\mathbf{v}_j \leftarrow \texttt{squash}(\mathbf{s}_j)$
7: for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow b_{ij} + \mathbf{u}_{j|i}.\mathbf{v}_j$
return $\mathbf{v}_j$

Fig 6. Routing-by-agreement algorithm

Generator Structure

In this implementation of Generator, multiple deconvolutional layers have been used similar to DCGAN architecture. We take in a random noise to build the first layer which would be a fully connected layer. The second layer after reshaping sits on tops of this input noise and this is fed as input to a series of batch normalizations, UpSampling2D, which simply doubles the dimensions of the input, convolutions and activation functions. The main aim then is to resize and generate a 28x28 size image that can be then used to compare.

Experiments

Variational Autoencoder

To create the VAE, the encoder and decoder networks need to be configured according to the specifications mentioned previously. The images below have been obtained with the following training parameters. The encoder and decoder have a single hidden layer each and both are of size 512 and using Relu activation function. The latent distribution has a size of 2. The network is trained with a batch size of 64 and it uses Adam optimizer. The training was done for 5000 epochs.

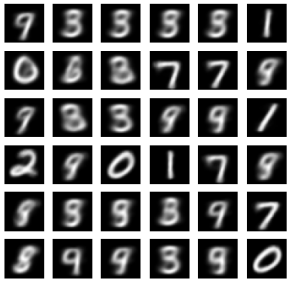

Fig 7. Image generated after 100 epochs

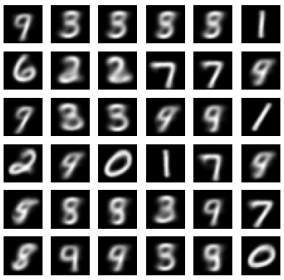

Fig 8. Image generated after 1000 epochs

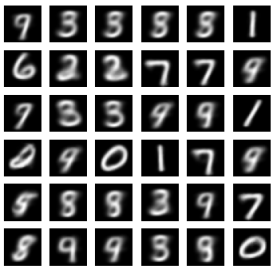

Fig 9. Image generated after 5000 epochs

The above images clearly show that the network converges quickly in a few numbers of epochs. The images that are generated are distinguishable as handwritten digits. The quality of the images that are generated however is not good, they are blurry. A similar result is obtained when modifying the parameters of the network.

Tweaking various parts of the network can help improve the performance of the VAE. For example, modifying the number of latent variables in the network can help the generator converge faster. The code has also been modified to include additional hidden layers in the encoder and decoder.

DCGAN

The model was trained on the MNIST dataset images for 150 epochs using the Adam optimizer with a learning rate of 0.0002 and a batch size of 128. All the minibatches were trained at each epoch sequentially. Fig 10,11,12 show the images generated after 1, 60 and 150 epochs. The images show a significant improvement from the first to 60th epoch. However, the difference between the images generated after 60 and 150 epochs is not as noteworthy. This is because the digits have started to show up in a good shape after 60 epochs and since the learning rate (0.0002) is constant, the image refines at a slower pace. Similar observation can be made by looking at the loss vs epoch graphs in Fig 13 and

14. The generator and discriminator losses change rapidly in the first few epochs but the change in loss subsides gradually thereafter.

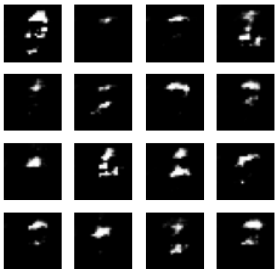

Fig 10. Image generated after 469 batch iterations (1st epoch)

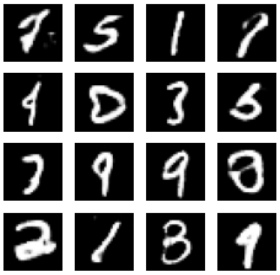

Fig 11. Image generated after 28,140 batch iterations (60 epochs)

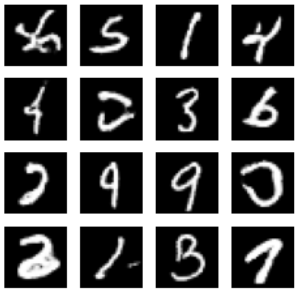

Fig 12. Image generated after 70,350 batch iterations (150 epochs)

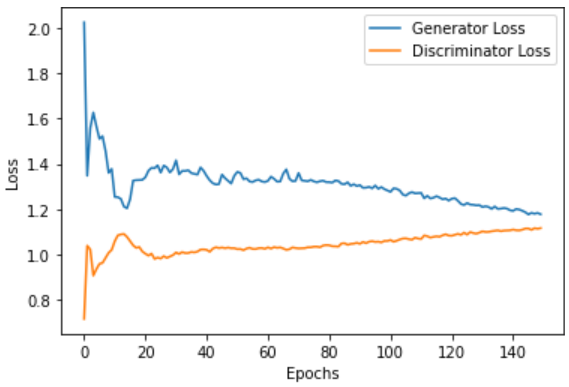

Fig 13. Generator and Discriminator Average Loss per epoch

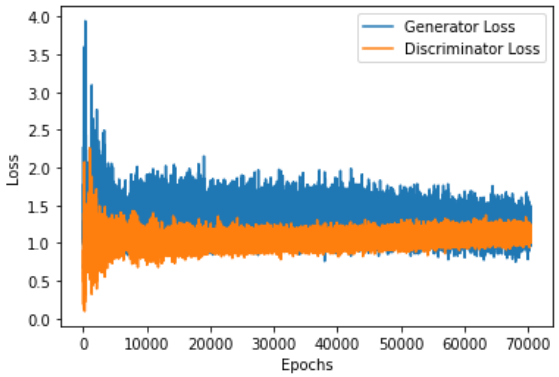

Fig 14. Generator and Discriminator Loss per batch per epoch

CapsuleGAN

The training of the generator was done using a combined model by stacking generator and discriminator. Initially a noise was fed to the generator to generate images. The discriminator would give the valid outputs for those generated images. A model was then constructed by stacking the generator and discriminator that takes noise as input, generates images and determines validity. This model was trained by taking the batch size as 32 by using the Keras training method- model.train_on_batch(). For training the discriminator, a random half of the batch was selected to create a noise. Keeping the generator static, the noise was used to generate fake images. Then the discriminator was trained on these fake images and real ones by feeding them into the training function one by one. The dataset was trained using GPU on Google Colab and training was done for 40000 epochs. The training loss is given by the abovementioned function- model.train_on_batch(). It runs a single gradient update on a single batch of data. Binary cross entropy loss was calculated, and Adam Optimizer has been used to train the data.

At each epoch, the progress is shown in the terms of the image generated, Discriminator Loss & Accuracy and the Generator Loss. It can be observed that the generator and discriminator loss fluctuate a lot at lower epochs although it decreases over time and becomes constant over larger number of epochs and flatten at about 2.5. This looks like the model has found an optimum performance limit and cannot improve more. The discriminator began with a higher loss and then decreased over time, flattening at 0.85. The accuracy was observed to be around 40-60% throughout the training. The MNIST dataset showed good and recognizable images from the 5000th epoch itself.

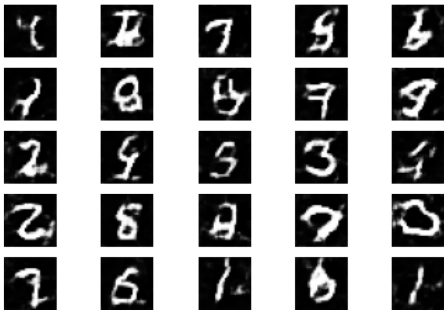

Fig 15. Image generated after 500 epochs

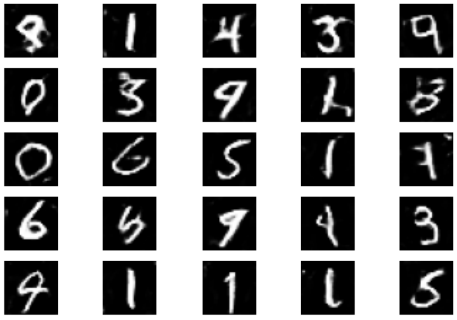

Fig 16. Image generated after 5000 epochs

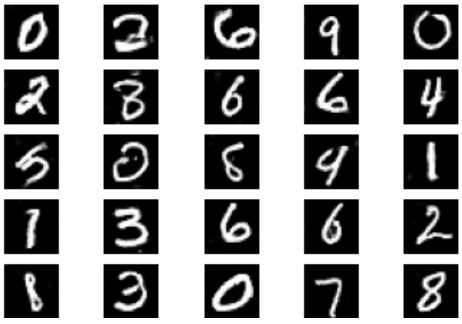

Fig 17. Image generated after 40000 epochs

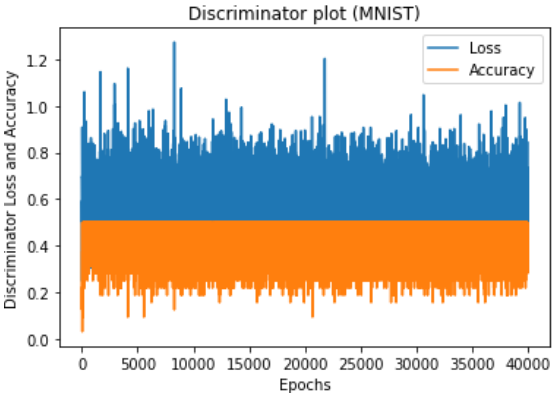

Fig 18. Discriminator loss and accuracy per epoch

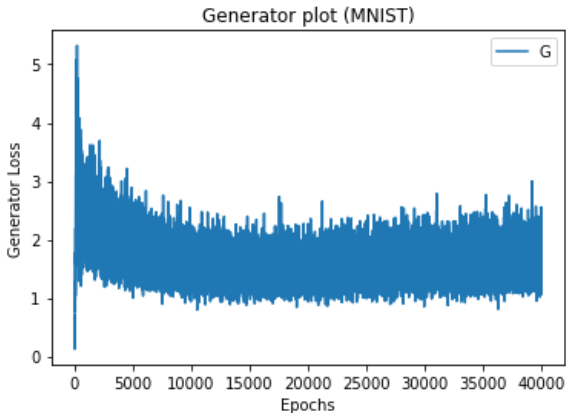

Fig 19. Generator Loss per epoch

Conclusion

This paper compares 3 different image generation models namely VAEs, DCGANS and CapsuleGANs. After performing the various experiments listed above and comparing the results of the different models, we can conclude that VAEs tend to get trained faster than the other 2 models and are able to generate images quicker. However, the images that are generated are blurry and not sharp. GANs on the other hand can generate much sharper images but the time taken to train these networks is much higher. We also conclude that the images generated with Capsule GANs are sharper and converge faster as compared to DCGANs.

References

Ayush Jaiswal, Wael AbdAlmageed, Yue Wu, Premkumar Natarajan. CapsuleGAN: Generative Adversarial Capsule Network. *USC Information Sciences Institute*. (2018)

Bang, D., & Shim, H. (2018). Improved training of generative adversarial networks using representative features. *arXiv preprint arXiv:1801.09195*.

Denton, E. L., Chintala, S., & Fergus, R. (2015). Deep generative image models using a laplacian pyramid of adversarial networks. In *Advances in neural information processing systems* (pp. 1486-1494).

Doersch, C. (2016). Tutorial on variational autoencoders. *arXiv preprint arXiv:1606.05908*.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y.: Generative Adversarial Nets. In: *Advances in neural information processing systems.* pp. 2672–2680 (2014)

Hernandez, E., Liang, D., Pandori, S., Periyasamy, V., & Singhal, S. Generative Models for Handwritten Digits

Kingma, D. P., & Welling, M. (2013). Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*.

Kingma, D. P., & Welling, M. (2019). An introduction to variational autoencoders. *Foundations and Trends in Machine Learning*, *12*(4), 307-392.

Marusaki, K., & Watanabe, H. (2020). Capsule GAN Using Capsule Network for Generator Architecture. *arXiv preprint arXiv:2003.08047*.

Radford, A., Metz, L., Chintala, S.: Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. In: *International Conference on Learning Representations* (2016)

Sabour, S., Frosst, N., Hinton, G.E.: Dynamic routing between capsules. In: *Advances in Neural Information Processing Systems*. pp. 3859–3869 (2017)