VARIABLY TRAINED GENERATIVE ADVERSARIAL NETWORKS

By

VIBHA SRIDHAR

A thesis submitted to the

Graduate School-Camden

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of Master of Science

Graduate Program in Computer Science

Written under the direction of

Professor Desmond Lun

And approved by

Professor Desmond Lun

Professor Rajiv Gandhi

Professor Jean-Camille Birget

Camden, New Jersey

May 2019

THESIS ABSTRACT

Variably Trained Generative Adversarial Network

By VIBHA SRIDHAR

Thesis Director:

Dr. Desmond Lun

Since their introduction in 2014, Generative Adversarial Networks (GAN), have been a hot topic in the AI field. Although the original implementation of GAN is for the generation of images, researchers have progressed beyond that and have created GAN variants that can generate music, perform style transfer and much more. One pitfall of GANs is that they are challenging to train. Many tricks have since been suggested for improved training. In this thesis, we combine these techniques with a state-of-the-art GAN variant in an attempt to improve GAN performance. We implement the Variably Trained GAN (VT-GAN) that combines the features of an Auxiliary Classifier GAN with a deep convolutional neural network architecture, and Label Smoothing and Minibatch Discrimination layer techniques to improve and stabilize GAN training, to generate realistic high quality images. The evaluation metric used is the Inception Score that gives a quantitative value measuring the realness of an image. Although it takes longer for VT-GAN to train due to the addition of complex mathematical operations, for the same amount of training, the VT-GAN performs approximately 3% better than the AC-GAN with respect to the Inception Score produced.

Contents

List of Figures

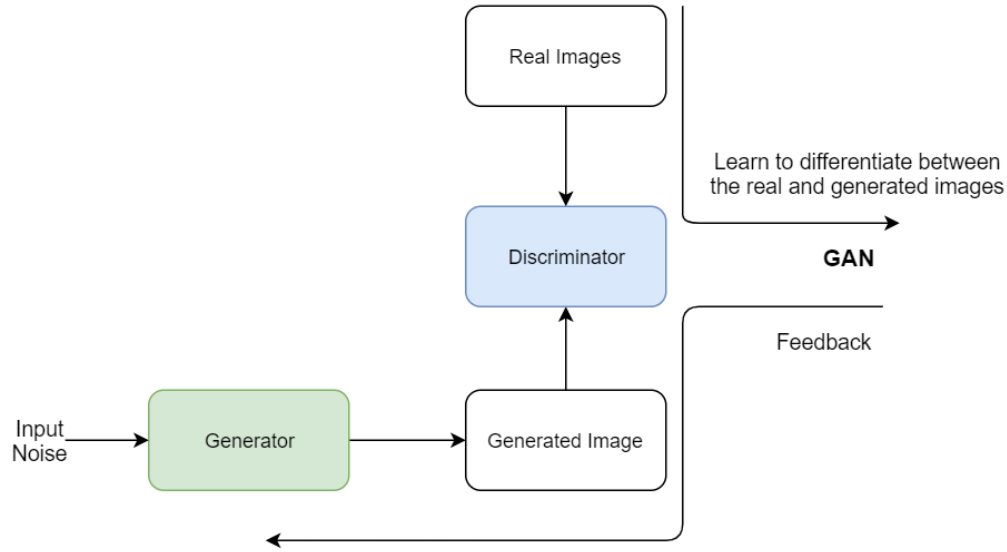
Chapter 1

Generative Adversarial Networks

Generative Adversarial Networks (GAN), first proposed in 2014 by Ian Goodfellow et al. [7] is an architecture based on a generative network that is adversarially trained.

GANs comprise of two neural networks (NN), a 'Generator' network G and a 'Discriminator' network D, that compete against each other. The Generator creates or generates data that is to be fed to the discriminator. The Discriminator predicts if its input data is real (i.e., from a test/train data set) or 'generated'. The Generator aims to fool the discriminator, for which it needs to generate data that looks identical to the real data as possible. The discriminator seeks to distinguish between the real and fake data as accurately as possible. This thus results in the generator learning to generate data that is indistinguishable from real data to the discriminator.

GANs can learn to imitate any distribution of data. So, in the perfect equilibrium, the generator would capture the general training data distribution. As a result, the discriminator would always be unsure of whether its inputs are real or not [25].



What GANs do is essentially play a minimax game between the two players, the Generator and the Discriminator. The Generator is the generative model, that performs the mapping $z \mapsto G(z) = x'$, where z is some random noise. The Generator's goal is to produce samples, x'. The Discriminator, performs a mapping $x \mapsto D(x) = (0, 1)$. Its goal is to look at input images, and determine if they are real samples ($D(x)$ closer to 1) or synthetic samples from the generator ($D(x)$ closer to 0). $D(x)$ can be interpreted as the probability that the image is a real training example. The Generator tries to win the game by producing images that look indistinguishable from the real images to the discriminator.

The Generator, $G(z)$, has parameters $\theta^{(G)}$, and the Discriminator, $D(x)$, has parameters $\theta^{(D)}$, where they control their respective parameters [5].

The solution to this zero-sum minimax game is called the Nash equilibrium. For $G(z)$ and $D(x)$, the Nash equilibrium corresponds to achieving the following:

- The Generator draws samples from, the original distribution of the data i.e the generated data is realistic enough to fool the Discriminator.

- The Discriminator cannot discriminate between what is real and what is generated so $D(x) = \frac{1}{2}$, for all x.

Let p_{data} denote the data distribution and let p_{model} denote the distribution of samples from the Generator. Note that the Discriminator does not have access to p_{data} and p_{model} initially, it is learnt as D goes through the training process. Then, the Discriminator's cost is:

$$\begin{aligned} L^{(D)} &= -\frac{1}{2}\ \mathrm{E}_{x\sim p_{data}}\ [\log\ D(x)] - \frac{1}{2}\ \mathrm{E}_{x'\sim p_{model}}\ [\log\ (1 - D(x'))] \\ &= -\frac{1}{2}\ \mathrm{E}_{x\sim p_{data}}\ [\log\ D(x)] - \frac{1}{2}\ \mathrm{E}_{x'\sim p_{model}}\ [\log\ (1 - D(G(z)))] \end{aligned}$$

This is just the standard cross-entropy cost that is minimized when training a standard binary classifier with a sigmoid output. The loss will be zero if $D(x) = 1$ for all $x \sim p_{data}$ and $D(x') = 0$ for all $x' \sim p_{model}$, i.e., generated from $G(z)$.

In this context, a Nash equilibrium is a tuple $(\theta^{(D)}, \theta^{(G)})$ that is a local minimum of $L^{(D)}$ with respect to $\theta^{(D)}$ and a local minimum of the Generator's loss $L^{(G)}$ with respect to $\theta^{(G)}$.

As this is a zero-sum minimax game, the sum of the Generator's loss and the Discriminator's loss is always zero, and hence the Generator's cost function $L^{(G)} = -L^{(D)}$ This minimax objective function can be realised as:

$$\min_{\theta^{(G)}}\max_{\theta^{(D)}} \frac{1}{2}\ \mathrm{E}_{x\sim p_{data}(x)}\ [\log D(x)]\ +\ \frac{1}{2}\ \mathrm{E}_{z\sim p_z(z)}\ [\log(1 - D((G(z))))]$$

Note that $\mathrm{E}_{z\sim p_z(z)}\ [\log(1 - D((G(z))))]$ is the log probability of D predicting that G's generated data is not genuine and that $\mathrm{E}_{x\sim p_{data}(x)}\ [\log D(x)]$ is the log probability of D predicting that real-world data is genuine. The discriminator wants to maximize the objective such that $D(x)$ is close to 1 and $D(G(z))$ is close to zero. The generator wants to minimize the objective so that $D(G(z))$ is close to 1.

The training process consists of simultaneous stochastic gradient descent. On each step, two minibatches are sampled: a minibatch of x values from the dataset and a minibatch of z values drawn from the model's prior over latent variables. Then two gradient steps are made simultaneously: one updating $\theta^{(D)}$ to reduce $L^{(D)}$ and one updating $\theta^{(G)}$ to reduce $L^{(G)}$

Figure 1.1 represents all the losses that are involved in this system.

DISCRIMINATOR			**GENERATOR**		
$\mathrm{Dloss}_{\mathrm{real}}$	=	log(D(x))	Gloss	=	log(1-D(G(z)))
$\mathrm{Dloss}_{\mathrm{generated}}$	=	log(1-D(G(z)))			
$\mathrm{Dloss}_{\mathrm{total}}$	=	log(D(x)) + log(1-D(G(z)))			

Figure 1.1: Losses

Generative Adversarial Nets [7] the paper that introduced GANs contain further explanation of the GAN value functions. NIPS 2016 Tutorial: Generative Adversarial Networks [5], is another excellent source that further delves into the mathematical functions, that are beyond the scope of this thesis.

Figure 1.2 captures the essence of a vanilla GAN, and showcases the most basic GAN architecture.

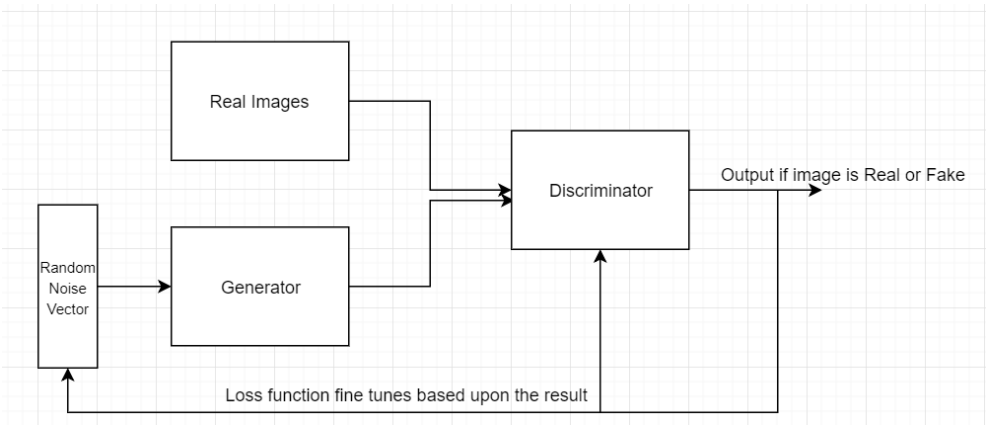


Figure 1.2: Basic GAN Architecture

Chapter 2

Related Work

Since their introduction, researches have proposed many different variants of GANs. Radford et al. in 2015 [22] introduced Deep Convolutional GAN (DCGAN) whose architecture is based on Deep Convolutional Neural Networks. Another extension relevant to this work is the introduction of Conditional GANs (cGAN) [17], where the Generator and Discriminator are conditioned on some external information like class labels. InfoGAN [1] is an information-theoretic extension of GAN that allows learning of meaningful representations, which are competitive with representations learned by some supervised methods. SRGAN [11] proposed by Christian et al. can recover photo-realistic textures from down-sampled images. SeqGAN [28] is a sequence generative framework, which can directly generate sequential synthetic data, and achieved significant performance in speech and music generation. Jiwei Li et al. [12] applied generative adversarial network approach to generate human-like dialogue. Let us take a quick look at the GAN variants on which the thesis is based.

2.1 Auxiliary Classifier GAN (AC-GAN)

The Auxiliary Classifier Generative Adversarial Network (AC-GAN) shown in Figure 2.1, is a GAN variant proposed by Odena et al. in their 2017 ICML paper [20]. In this, the Discriminator has two different outputs. The first output layer is to determine if the input image is real or generated. The second is the 'Auxiliary Classifier' that has the output dimension equivalent to the number of image classes and the values determine which class the input belongs to.

Let X_{real} represent the real images sample, X_{gen} the generated sample, S the probability distribution if real or generated, and C the probability distribution for the class labels. The objective function of the AC-GAN has two parts:

- The log-likelihood of the correct source, i.e. real or generated L_S
- The log-likelihood of the correct class, L_C

$$L_S = \mathrm{E}[\log\ P(S\ =\ real\ |X_{real})] + \mathrm{E}[\log\ P(S\ =\ generated\ |X_{gen})]$$
$$\mathrm{L}_C = \mathrm{E}[\log\ P(C\ =\ c|X_{real})] + \mathrm{E}[\log\ P(C\ =\ c\ |X_{gen})]$$

The minimax game played between the Discriminator and Generator is thus approximated by training the Discriminator/auxiliary classifier to maximize $L_c + L_s$ and training the Generator to maximize $L_c - L_s$.

Figure 2.1: AC-GAN

2.2 Deep Convolutional GAN

DCGANs build the Discriminator D with convolutional network and the Generator G with the transposed convolutional network [22]. The paper proposes a family of architecture based upon convolutional neural networks that increase the stability of GAN training.

The following has been suggested for improved training :

- Do not use Pooling layers. Instead, use strided and fractionally strided convolutions for the Discriminator and the Generator respectively.
- Batch Normalization is to be used
- Remove any connected layers
- Utilize ReLU activation function in all layers of the Generator except for Tanh function in the output layer
- Use LeakyReLU activation function in the Discriminator for all layers.

Apart from these suggestions, the paper also explores using the convolutional layers of the Discriminator as a feature extractor and further visualizing these features. They further explore the vector arithmetic properties of the Generator and combine properties of different images to generate new images that contain certain aspects of each of these images, as shown in Figure 2.2.

Figure 2.2: Exploiting Vector Arithmetic Properties of Generator

Chapter 3

Variably Trained Generative Adversarial Networks

Implemented in this thesis, is a Variably Trained GAN (VT-GAN), which combines the features of an AC-GAN, and the techniques suggested in Improved Techniques for Training GANs by Tim Salimans et al. [24] for a much superior performance and stable training. AC-GAN is being utilized because it has achieved state-of-the-art Inception Score performance of 8.24 even without these additional techniques for improved training. Hence, it can be seen that providing class information enables the model to learn better and generate a variety of realistic images. VT-GAN borrows the class conditioned property of the AC-GAN and in addition, includes Label Smoothing and a Minibatch discrimination layer. These additions have been implemented to stabilize GAN training and to try and improve the quality and diversity of the generated images. The addition of these two techniques has been explored in the

upcoming sections. VT-GAN consists of a Generator and a Discriminator network, both of these are designed as Deep Convolutional Neural Nets. The Discriminator has an auxiliary classifier, that classifies the input image into the class it belongs to.

The introduction of the Minibatch Discrimination layer also changes the way the Discriminator is trained. First, the real images are fed in together, and then the generated images are given. This is different from the way the input is typically given, where the real and generated images are mixed and fed as input.

The implementation of this thesis has been done in Keras [2] with Tensorflow [16] back end. Google Colaboratory was used for the execution. Google Colaboratory provides access to Google's GPU with a Virtual Machine instance that is active for 12 hours before it is completely reset.

3.1 Data set

This network has been designed to be trained on the CIFAR-10 [9] database, and subsequently generate images from the classes present in this. CIFAR-10 consists of 60000 colour images that are 32x32 pixels. Each image belongs to one of the 10 classes, with 6000 images per class. The images are split into 6 batches - 50000 training images and 10000 test images. The classes are mutually exclusive with no overlap between the classes. Figure 3.1 shows sample images from the CIFAR-10 data set.

The following information is present in each batch :

- Data: 10000x3072 as numpy array. Each row of the array stores a 32x32 colour

image.

- Labels: a list of 10000 numbers in the range 0-9.

Figure 3.1: CIFAR10 Data set

3.2 Discriminator

The Discriminator takes as input, real or generated images (CIFAR10 in this case) that pass through a series of strided - convolutional layers instead of pooling layers as it lets the network learn its own pooling function. This is essentially a binary classifier that is implemented as a CNN. The Discriminator design is shown in Figure 3.2. Leaky ReLU activation function is used in all layers here. In the end, we use a Sigmoid function is used in the end to generate the probability distribution for the input to be generated or real image. Followed by a Softmax function to determine the class that the input image belongs to.

Convolutional Layer

In order to fully understand the workings of a Convolutional Neural Network, the terms used to describe the networks are explained below :

- Kernel Size: It defines the field of view of the convolution. A kernel size of 3 indicates 3x3 pixels.

LAYER	OPERATION	KERNEL	STRIDE	FILTERS	BATCH NORMALIZATION	ACTIVATION FUNCTION
HIDDEN LAYER 1	2D CONVOLUTION	5	2	16	N	Leaky ReLU
HIDDEN LAYER 2	2D CONVOLUTION	5	1	32	Y	Leaky ReLU
HIDDEN LAYER 3	2D CONVOLUTION	3	2	64	Y	Leaky ReLU
HIDDEN LAYER 4	2D CONVOLUTION	5	1	128	Y	Leaky ReLU
HIDDEN LAYER 5	2D CONVOLUTION	3	2	256	Y	Leaky ReLU
HIDDEN LAYER 6	2D CONVOLUTION	3	1	512	N	Leaky ReLU
MINIBATCH DISCRIMINATION LAYER	1D CONVOLUTION	30				
DENSE	Real or Generated Classification					Sigmoid
Auxiliary Classifier	Classification into classes					Softmax

Figure 3.2: Discriminator Design

- Stride: It defines the step size of the kernel when traversing the image. A stride of 2, indicates that the kernel slides by 2.
- Padding: The padding determines how to handle the border of an input.

This layer helps in extracting features from the input, considering spatial pixel relationship.

The convolutional layer can be easily implemented via a library in Keras with Tensorflow backend. It is merely a matter of invoking the conv2D function with the required parameters for kernel size, padding, and strides.

3.3 Generator

The Generator takes as input a uniform D-dimensional noise vector and produces or generates sample images. The Generator implemented here has fractionally-strided convolutional layers [22].

Leaky ReLU activation function is used in all the layers except for the last one which uses the Tanh function. This is then followed by Batch Normalization to stabilize the outputs of each layer (except for the last), as shown in Figure 3.3. The architecture closely follows the one proposed in [22] to ensure stable training.

LAYER	OPERATION	KERNEL	STRIDE	FILTERS	BATCH NORMALIZATION	ACTIVATION FUNCTION
INPUT	DENSE			512	N	Leaky ReLU
HIDDEN LAYER 1	TRANSPOSED CONVOLUTION	5	2	256	Y	Leaky ReLU
HIDDEN LAYER 2	TRANSPOSED CONVOLUTION	5	2	128	Y	Leaky ReLU
HIDDEN LAYER 3	TRANSPOSED CONVOLUTION	5	2	64	Y	Leaky ReLU
HIDDEN LAYER 4	TRANSPOSED CONVOLUTION	5	2	3	N	Tanh

Figure 3.3: Generator Design

Transposed Convolutional Layer

Transposed Convolutional Layers or Fractionally Strided Convolutional Layers are used in the Discriminator for up-sampling an input image. The input image - be it a real one or a generated image is 32x32, and this needs to be expanded to enable the Discriminator to learn to identify the objects present. A transposed convolutional layer carries out a regular convolution but reverts its spatial transformation.

3.4 Activation Function

Activation functions enable a Neural Network to learn and understand the non-linear mappings between the inputs and response variables. They introduce non-linear properties into the Network. They are used so that the network learns to differentiate between the outputs and learns to adapt with the data.

The activation functions used in the creation of this system are: Sigmoid, Tanh, Leaky ReLU and Softmax activation functions. These have been described below.

Sigmoid Activation Function

Sigmoid Activation functions are used when the required output is a probability, as the sigmoid function only has values within the [0,1] range. Additionally, as the function is differentiable, the slope of the sigmoid curve can be found at any two points. This function has been plotted in Figure 3.4.

Tanh Activation Function

The Tanh(x) function, shown in Figure 3.5 can be viewed as a re-scaled version of the sigmoid, with output in the interval of (1, 1). Although the Tanh function saturates like the Sigmoid function, its output is zero-centered unlike the sigmoid. Therefore, in practice, the tanh non-linearity is always preferred to the sigmoid nonlinearity.

$$f(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

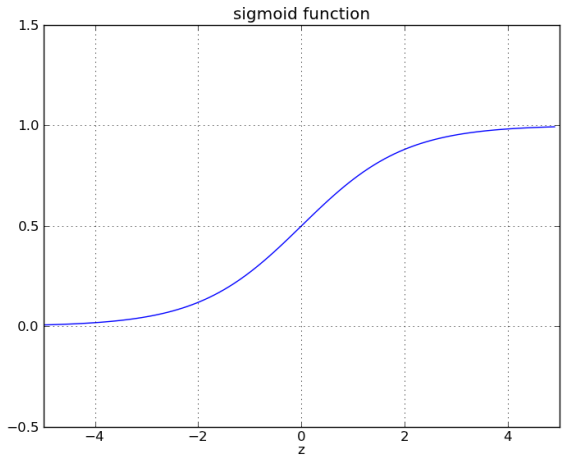


Figure 3.4: Sigmoid Function

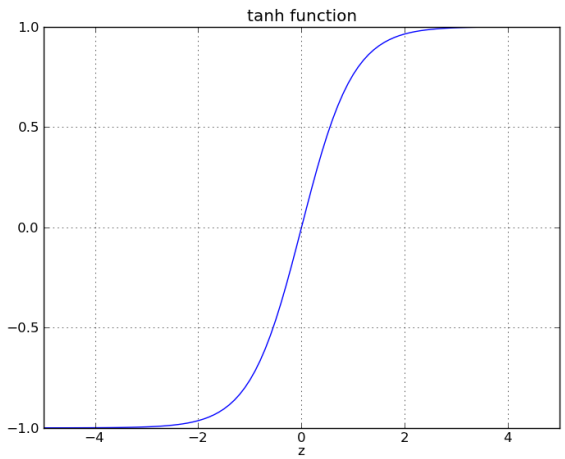


Figure 3.5: Tanh Function

ReLU (Rectified Linear Unit) Activation Function

The ReLU function as described in [18] was introduced as an activation function in [10], it is half rectified i.e, ReLU is linear for all positive values, and zero for all

negative values.

ReLU is sparsely activated since ReLU is zero for all negative inputs, it's likely for any given unit to not activate at all. Sparsity generates models that often have better predictive power and less overfitting as it's more likely that neurons are processing essential parts of the problem [13]

One issue with this function is that it suffers from the "Dying ReLU" problem. A ReLU neuron is "dead" if it's stuck in the negative side and thus unable to climb back up to the positive side keeps on generating 0 as the result. Because the slope of ReLU in the negative range is also 0, once a neuron gets negative, it's unlikely for it to recover. This might over the time end up causing a large part of the network to do nothing. The dying problem is likely to occur when the negative bias is significant or if the learning rate is very high. Figure 3.6 demonstrates the ReLU function.

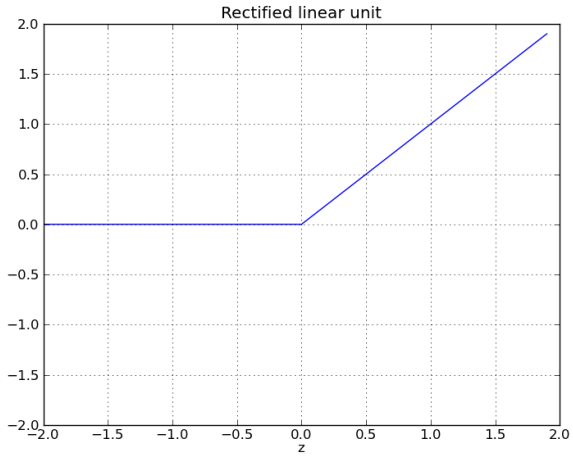


Figure 3.6: ReLU Function

LeakyReLU Activation Function

LeakyReLUs is a way to solve the dying ReLU problem when utilizing the ReLU activation function. LeakyReLUs have a small positive gradient for negative inputs, instead of 0, this can be seen in Figure 3.7.

$$y = \begin{cases} 0.01 * x, & \text{when } x < 0 \\ 1, & \text{when } x \leq 0 \end{cases}$$

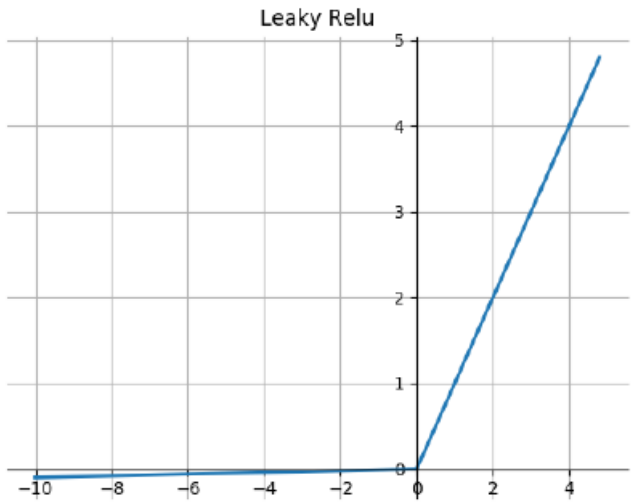


Figure 3.7: Leaky ReLU Function

Softmax Activation Function

The Softmax function is used when the desired output is the probabilities over a range of classes. These probabilities together sum to 1. In order to determine the

class of the input image, the softmax function is used. This function assigns larger probabilities to the classes that the network thinks the input image belongs to.

3.5 Batch Normalization

As the distribution of the activation constantly changes during training in the intermediate layers, each layer must learn to adapt itself to a new distribution, thus, slowing down the training process; this is called internal covariate shift. One way to overcome this would be to ensure that the input of every layer has approximately the same distribution in every training step.

Batch normalization [8] is a method we can use to normalize the inputs of each layer, to fight the internal covariate shift problem. Batch normalization potentially helps the neural network converge faster and get higher overall accuracy by shifting inputs to zero-mean and unit variance, which makes the inputs of each trainable layer comparable across features. Thus it also prevents early saturation of nonlinear activation functions such as TanH and Sigmoid not to get stuck in the saturation mode, i.e. where the gradient is almost 0. In practice, networks that use batch normalization are more robust to bad initialization [8].

During training time, a batch normalization layer does the following using:

- Calculate the mean and variance of the layer's input
- Normalize the layer inputs using the previously calculated batch statistics
- Scale and shift in order to obtain the output of the layer

Batch Normalization is implemented directly in Keras via a library. This function is invoked after implementing each layer except for the last, as there is no further input once at the end.

3.6 Minibatch Discrimination Layer

A common failure mode of Generative Adversarial Networks (GANs) is 'mode collapse' in which the Generator fools the Discriminator by synthesizing generated data that looks exactly the same across one batch. The Generator thus does not learn to create different images, and the Discriminator does not learn to identify this image as a generated sample.

To solve this problem, a concept called 'Minibatch Discrimination' was proposed. Minibatch discrimination [24] is defined to be any method where the Discriminator is able to look at an entire batch of samples in order to decide whether they come from the Generator or the real data, instead of looking to a single input. The single mode collapse can then be easily spotted, by the Discriminator which will understand that whenever all the samples in a batch are very close to each other, the data is fake. This will thus make the Generator synthesize different image in each batch.

The following steps [4] are implemented to realize the Minibatch Discrimination layer :

- Output of an intermediate layer, the desired number of kernels and its dimension is fed into the Minibatch Discrimination layer as input.

- This input is multiplied by a 3D tensor to produce a matrix of size $num\_kernels$ x $kernel\_dim$.
- L1-distance between rows in this matrix across all samples in a batch is computed. A negative exponential is then applied to this.
- The Minibatch Features are then computed as the sum of these exponentiated distances.
- The input to the minibatch layer is then concatenated with the newly created Minibatch Features
- This is the output of this layer which is then fed as input to the next layer of the Discriminator.

This is implemented using the existing Python implementation by Keras, which has been integrated here.

3.7 Label Smoothing

Label Smoothing [23] [26] regularizes a model based on a softmax with k output values by replacing the hard 0 and 1 classification targets with targets of $\frac{\epsilon}{k}$ and $1 - \frac{k-1}{k}\epsilon$, respectively. In other words, label smoothing is the practice of assigning each class some weight, instead of assigning the ground truth label the full weight. As the network will over fit less on the training labels, it should be able to generalize better, hence also making it easier for the network to recover better from a false label [6].

3.8 Loss Function

A loss function is the objective to measure the compatibility between a prediction and the ground truth label. Cross-entropy loss, or log loss, measure the performance of a classification model whose output is a probability value between 0 and 1 [14]. If the predicted probability diverges from the true label, the cross-entropy loss increases. Although Cross-entropy and log loss are slightly different depending on the context, they resolve to the same value when calculating error rates between 0 and 1 as they do in this context.

Binary Cross Entropy Loss

Binary Cross entropy loss is used when classifying the input image as real or generated - which is binary classification. It is also known as the Sigmoid Cross Entropy Loss as it incorporates both the Sigmoid Activation function and the Cross-Entropy loss.

Sparse Categorical Cross Entropy Loss

Sparse Categorical cross entropy loss is used when the problem is single-label multi-class. This is used so that the Discriminator can classify the input image to one of the 10 classes. We use this over just Categorical Cross Entropy Loss as the targets are not one-hot encoded, hence just integer results indicating the classes suffice.

Chapter 4

Results

A significant problem faced here is how best to evaluate these kinds of generative models. Although one way is to examine the quality of the generated samples physically, this does not entirely satisfy us due to its non-quantitative nature. In the text domain, this becomes even more challenging. Generative models based on maximum likelihood training may use some metric based on likelihood (or some lower bound to the likelihood) of unseen test data, but that is not applicable here. Some GAN papers have produced likelihood estimates based on kernel density estimates from generated samples, but this technique seems to break down in higher dimensional spaces. Another solution is to only evaluate on tasks such as classification, that are well studied.

4.1 Inception Score

First proposed by Salisman et al. in [24] the Inception score (IS) is based upon Google's Inception classifier and has since become a popular metric for judging the outputs of GANs that are trained to generate images based on Imagenet-1k classes. IS uses the pre-trained Inception Network classifier to capture the following properties of generated samples:

- The quality of the generated images
- The diversity of the generated images

As the IS is a metric that correlates with a human's judgement, it measures the realness of images created by a GAN.

The Inception classifier takes as input, images, and returns the probability distribution of labels for the image. The input to calculate IS is a list of images, and the result is a single floating point number, which is the score. The score will be high when the following two conditions are true :

- The distribution of classes for an image has low entropy - i.e. the image should contain clear objects that enable the Inception network to be confident that there is a single object in the image.
- The overall distribution of classes across the sampled data has high entropy, implying that there is diversity of images from all the classes.

The score drops low if either one of them is false or if both are false. A higher score is desired as that is an indication that the GAN can generate a large number of

different distinct images. The lowest score possible is zero, and theoretically, while the highest possible score is infinity, in practice there will probably emerge a non-infinite ceiling. Here, we take the IS over real images (CIFAR-10) as the upper bound.

To compute this score, the Kullback-Leibler (KL) divergence, D_{KL} is used. Given two probability distributions, KL divergence is a measure of how similar they are. D_{KL} is computed as follows given two probability distributions P and Q that are defined on the same probability space:

$$D_{KL}(P||Q) \;=\; \sum_{x \in X} \; P(x) \; \log\left(\frac{P(x)}{Q(x)}\right)$$

When the distributions are not similar, i.e. when each generated image has distinct labels and the overall set of generated images has a diverse range of labels, the KL divergence has a high value.

In order to compute the Inception Score, the KL divergence between the label distribution of an image and the marginal label distribution is computed.

Let y be the class label, x a generated image, E the expectation over the set of generated images, D_{KL} is the KL-divergence between the conditional class distribution $p(y|x)$ and the marginal label distribution $p(y)$. Then, the IS is computed as :

$$IS(G) \;=\; \exp(\mathrm{E}_x \;\; D_{KL} \; (p(y|x) \; || \; p(y)))$$

The marginal label distribution $p(y)$ is computed as

$$p(y) = \frac{1}{N} \; \sum_{n=1}^{N} \; p(y|x_n \;=\; G(z_n))$$

where N is the total number of images sampled and $G(z)$ is the Generator output, i.e. a generated image. Every individual sample's label distribution (from the inception network's classification output) is used to find the overall label distribution. If this is uniform, then it implies that the Generator can produce images across different labels showing diversity.

Figure 4.1: Variation of KL divergence value

Figure 4.1 [15] shows how the KL divergence value changes depending upon the conditional label distribution and the marginal label distribution. The one resulting in high KL divergence has low entropy for conditional label distribution and high entropy for marginal label distribution. This means the input data has images where each image has distinct labels and the overall set of generated images has a diverse range of labels. Once the KL divergence is computed, it's exponential is taken and finally, this is averaged over all of the images to produce the Inception score for the input images.

Due to the limited amount of resources available in terms of GPU and storage restrictions, the Inception Score for the different architectures implemented has been calculated for 1,507 images generated from networks that have been trained for approximately 500 epochs each. Ideally, the GAN should be trained for over several thousand epochs and approximately 50,000 images are to be fed into the Inception Network to compute the score so that the diversity between the images is captured well, [24] also acknowledges this fact.

Figures 4.2 and 4.3 exhibit samples of images from the AC-GAN trained for 500 epochs without the Minibatch Discrimination Layer.

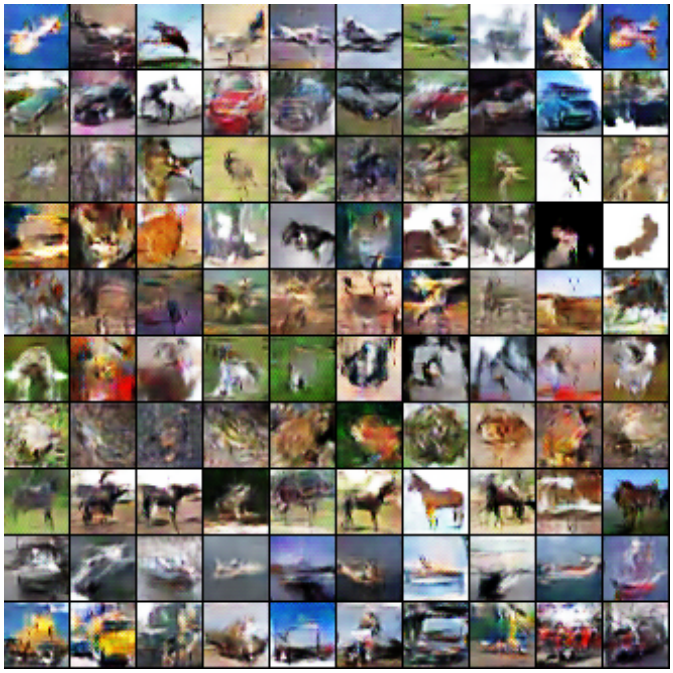


Figure 4.2: 500 epoch images without Minibatch Discrimination Layer(AC-GAN) - 1

Figures 4.4 and 4.5 exhibit samples of images from the VT-GAN implemented in this thesis that has been trained for 500 epochs and includes the Minibatch Discrimination Layer.

The original AC-GAN implementation, computed the Inception score for 50,000 32x32 resolution samples that are split into 10 groups at random. This score is produced by the best AC-GAN model obtained after performing a grid search across 27 hyper-parameter configurations. Splitting of the input into classes and averaging the Inception Score over these classes is done following the protocol of evaluating

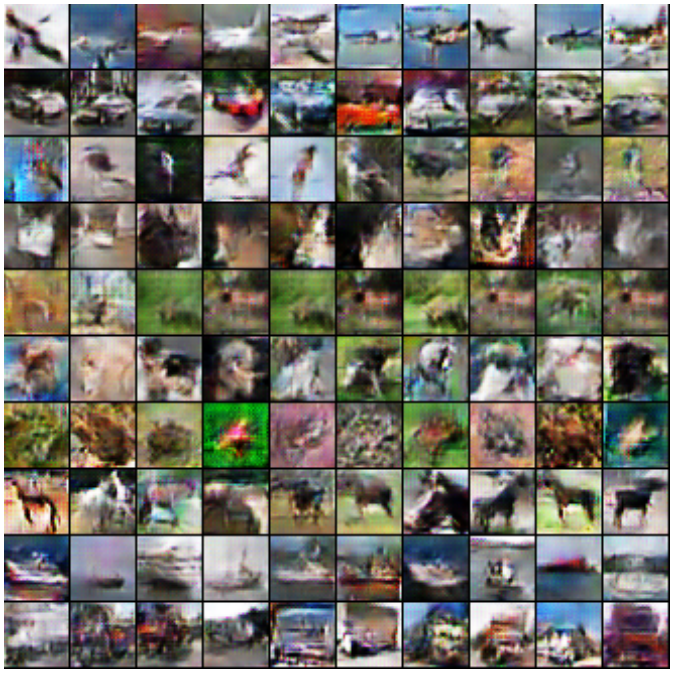


Figure 4.3: 500 epoch images without Minibatch Discrimination Layer(AC-GAN) - 2

the average Inception score over 10 independent groups of 5,000 samples each, as established by the paper that first proposed this score [24].

VT-GAN and AC-GAN were trained twice each for 500 epochs, and images were generated for each model. We compute the Inception score for 1507, 32x32 resolution samples from both the VT-GAN models, split into 10 groups at random. Similarly, we compute the Inception score for 1507, 32x32 resolution samples from AC-GAN, these are also split into 10 groups at random.

Following the method used by the original AC-GAN paper for reporting results, we report the best model's score and their respective standard deviations for VT-GAN and AC-GAN in Figure 4.6. We further mention that the other VT-GAN

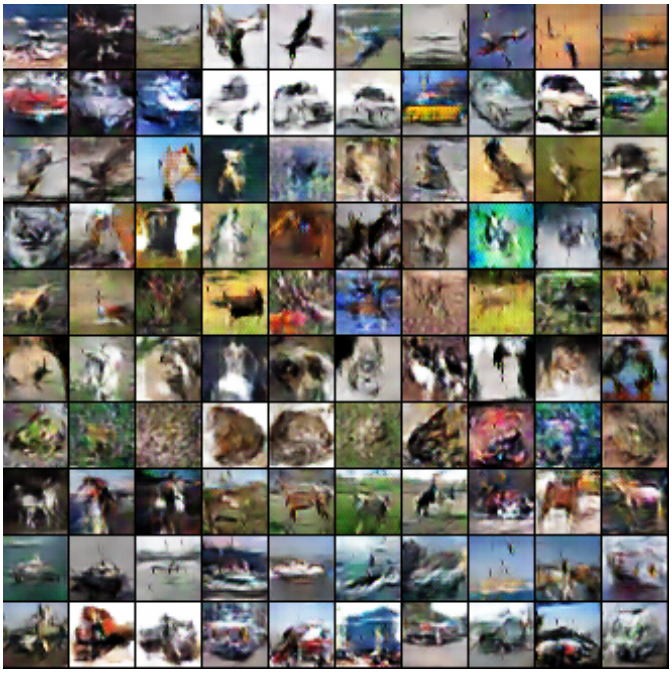


Figure 4.4: 500 epoch images with Minibatch Discrimination Layer(VT-GAN) - 1

model produced an IS of 4.13±0.29 and our AC-GAN implementation's other model produced a score of 4.01±0.22.

From this we can see that the best VT-GAN model performs approximately 3% better than our best AC-GAN. We believe that given more training, VT-GAN would give a performance that is similar to or slightly better than the original AC-GAN implementation, but given the hardware constraints, we are unable to increase the training.

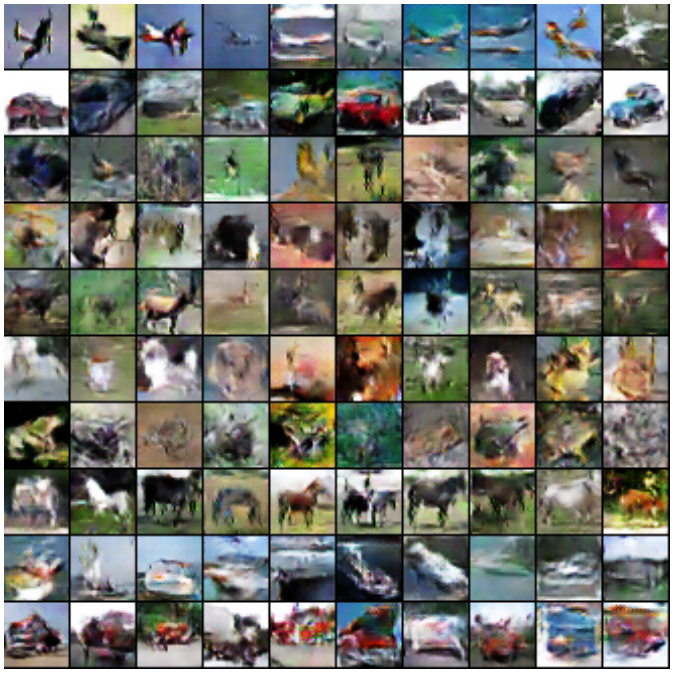


Figure 4.5: 500 epoch images with Minibatch Discrimination Layer(VT-GAN) - 2

4.2 Issues Faced During Training

Training GANs is extremely tricky. They are challenging to train and require countless hours of tweaking and modifying the architecture to prevent them from collapsing. Even then, a close eye has to be kept on them during training, to prevent mode collapse. During the course of training the network, the following issues were faced and subsequently most were overcome:

- Choosing the right Kernel size turned out to be a huge task. A Kernel size of 5 worked well with all layers of the Generator. Alternating between 3 and 5 in the Discriminator turned out to be ideal, as using 3 in all the layers caused the

MODEL/IMAGE	NUMBER OF INPUT	EPOCHS TRAINED	INCEPTION SCORE (Best Model)
CIFAR-10	50000	N/A	11.24
AC-GAN (Original Implementation)	50000	50000	8.25 $\pm$0.07
AC-GAN	1507	500	4.02$\pm$0.18
VT-GAN	1507	500	4.14 $\pm$0.20

Figure 4.6: Inception Scores

Discriminator to collapse.

- Just because a network is deep, it doesn't automatically mean that it will work perfectly. The Generator went through many rounds of altering the number of layers, the filter sizes before settling on the current architecture.

- One major issue faced due to the limited availability of time and resources, early stopping had to be implemented. We believe that if the network had been trained for several thousand epochs, instead of stopping at 500, the generated images would have definitely improved on quality.

4.3 Visualizing the Losses

In order to track and understand the GAN performance, the losses incurred by the Generator and the Discriminator may be visualized. Generation loss refers to the Generator's loss. This same term may be used to describe Discriminator's loss as well. This is the Binary Cross entropy Loss described in Chapter 3.

In a zero-sum game, the Nash Equilibrium is the same as the minimax solution. Simply put, Nash equilibrium is when a network's actions remain unchanged regardless of the other network's actions. Hence, the convergence of the losses to the Nash Equilibrium is desired as it implies that the system is stable and both the Generator and the Discriminator are strong.

- Generation Loss per epoch during the Training Phase has been plotted in Figure 4.7 for the Generator and the Discriminator. It can be seen that they end up converging pretty close to the Nash equilibrium as desired pretty early on.

- Generation Loss per epoch during Testing Phase is shown in Figure 4.8. An observation point here is the fact that the Discriminator's loss does not converge to 0 - which shows stable training. When Discriminator's loss goes to 0, it means that it has become much stronger than the Generator, thus implying that the Generator is performing poorly with respect to creating realistic images. Once it's loss goes to 0, the weights no longer change as the gradients being loss derivatives stagnate, and hence the Generator cannot improve in the following iterations.

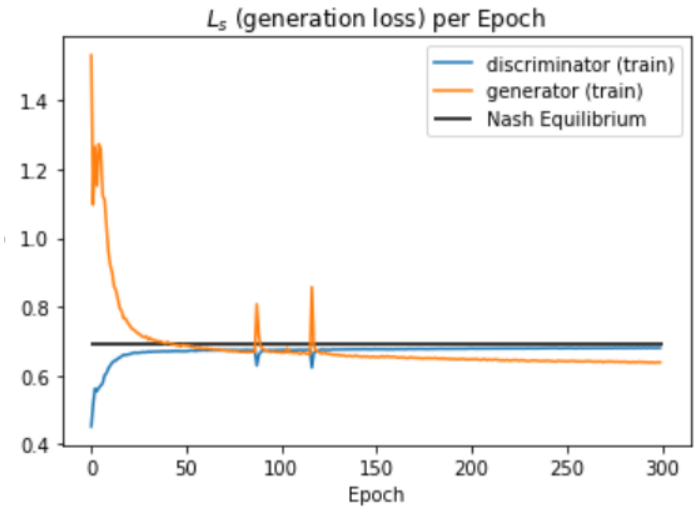


Figure 4.7: Generation Loss per epoch for Training Phase

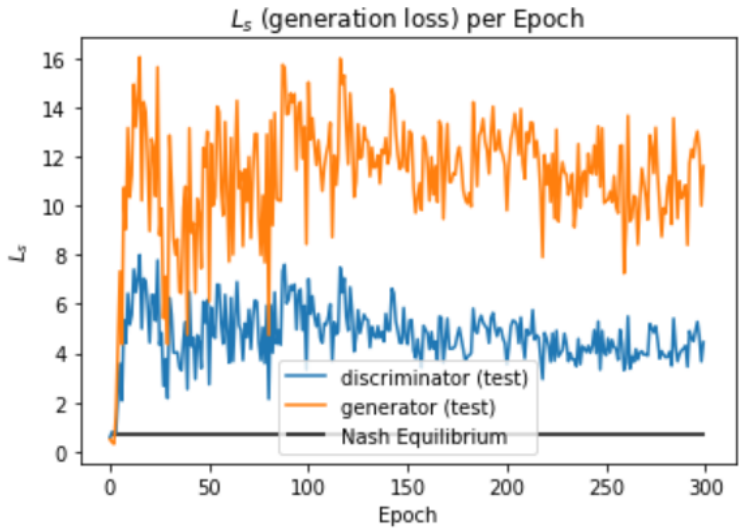


Figure 4.8: Generation Loss per epoch for Testing Phase

- Auxiliary or Classification Loss for the Discriminator is the loss associated with the Discriminator determining the correct label for the input image. This is

the Sparse Categorical Cross Entropy Loss explained in Chapter 3. Figure 4.9 shows that although the Discriminator performs well and minimizes the loss during the Training phase, it does not perform as well during the testing phase. This would once again correlate to early stopping. Decrease is seen usually when we hit a few thousand epochs.

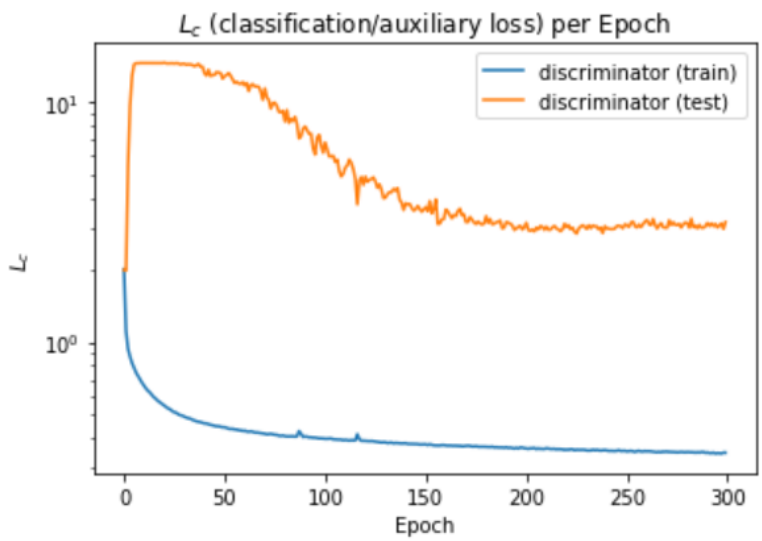


Figure 4.9: Discriminator's Classification Loss per epoch

- Auxiliary or Classification Loss for the Generator is shown in Figure 4.10. It has to be noted that both the Generator and the Discriminator try to minimize this value so that the Generator learns to generate better images that belong to each class and the Discriminator can distinguish between the classes.

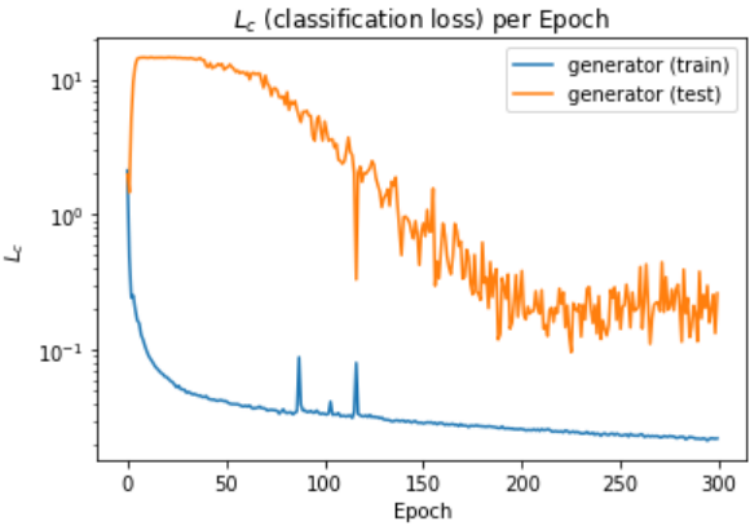


Figure 4.10: Generator's Classification Loss per epoch

Chapter 5

Conclusion and Future Extensions

From the results obtained, it is observed that the addition of the Minibatch Discrimination layer increases the execution time. It takes roughly the same amount of time to reach epoch 350 with this layer as it does to reach epoch 520 without it. This increase in execution time is to be expected as this additional layer incorporates matrix multiplication and exponentiation for each batch of images. This leads us to speculate whether the addition of this layer is meaningful or not. The answer to this would, of course, depend on the restrictions one has for run time availability. If time and computer resources for computation is plenty and not a matter of concern, it makes sense to add in this layer for the sake of increased generated image quality and much more stable GAN network.

We also note that no discernible performance difference has been observed with using label smoothing as compared to an architecture without it. This might become more noticeable if the number of training epochs reaches over a couple of thousand.

There is a huge scope for future improvements in the area of using GANs to generate images. One suggestion would be to further explore the possibility of generating never-before-seen creatures by exploiting the vector arithmetic property of the generators. This generator property could also be of excellent use for clothing companies to design different variations of a clothing article to suit a wide variety of audiences. Another would be to use a combination of loss functions such as Wasserstein distance with L1 distance and with empirical study determine a middle ground that works well.

Bibliography

[1] Xi Chen et al. "InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets". In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. 2016, pp. 2180–2188.

[2] François Chollet et al. *Keras*. 2015. URL: `https://keras.io` (visited on 03/15/2019).

[3] Utkarsh Desai. 2018. URL: `https://github.com/utkd/gans` (visited on 04/04/2019).

[4] John Glover. *An introduction to Generative Adversarial Networks*. 2016. URL: `http://blog.aylien.com/introduction-generative-adversarial-networks-code-tensorflow` (visited on 04/15/2019).

[5] Ian Goodfellow. "NIPS 2016 Tutorial: Generative Adversarial Networks". In: *Computing Research Repository* abs/1701.00160 (2017).

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[7] Ian Goodfellow et al. "Generative Adversarial Nets". In: *Advances in Neural Information Processing Systems 27*. 2014, pp. 2672–2680.

[8] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proceedings of the 32nd International Conference on Machine Learning*. 2015, pp. 448–456.

[9] Alex Krizhevsky. *Learning Multiple Layers of Features from Tiny Images (CIFAR-10)*. Tech. rep. 2009. Chap. 3. URL: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Neural Information Processing Systems*. 2012, pp. 1106–1114.

[11] Christian Ledig et al. "Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network". In: *IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 105–114.

[12] Jiwei Li et al. "Adversarial Learning for Neural Dialogue Generation". In: *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. 2017, pp. 2157–2169.

[13] Danqing Liu. *A Practical Guide to ReLU*. 2017. URL: https://medium.com/tinymind/a-practical-guide-to-relu-b83ca804f1f7 (visited on 04/29/2019).

[14] *Loss Functions*. 2017. URL: https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html (visited on 04/07/2019).

[15] David Mack. *A simple explanation of the Inception Score*. 2019. URL: https://medium.com/octavian-ai/a-simple-explanation-of-the-inception-score-372dff6a8c7a (visited on 05/01/2019).

[16] Martın Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[17] Mehdi Mirza and Simon Osindero. "Conditional Generative Adversarial Nets". In: *arXiv e-prints* (2014), arXiv:1411.1784.

[18] Vinod Nair and Geoffrey E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. 2010, pp. 807–814.

[19] Erik Norén. 2018. URL: https://github.com/eriklindernoren/Keras-GAN (visited on 04/03/2019).

[20] Augustus Odena, Christopher Olah, and Jonathon Shlens. "Conditional Image Synthesis with Auxiliary Classifier GANs". In: *Proceedings of the 34th International Conference on Machine Learning*. 2017, pp. 2642–2651.

[21] Federico Peccia. *Batch normalization: theory and how to use it with Tensorflow*. 2018. URL: https://towardsdatascience.com/batch-normalization-theory-and-how-to-use-it-with-tensorflow-1892ca0173ad (visited on 04/28/2019).

[22] Alec Radford, Luke Metz, and Soumith Chintala. "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks". In: *International Conference on Learning Representations*. 2016.

[23] Scott Reed et al. "Training Deep Neural Networks on Noisy Labels with Bootstrapping". In: *arXiv e-prints* (2014), arXiv:1412.6596.

[24] Tim Salimans et al. "Improved Techniques for Training GANs". In: *Advances in Neural Information Processing Systems 29*. 2016, pp. 2234–2242.

[25] Thalles Silva. *An intuitive introduction to Generative Adversarial Networks (GANs)*. 2018. URL: https://medium.freecodecamp.org/an-intuitive-introduction-to-generative-adversarial-networks-gans-7a2264a81394 (visited on 03/08/2019).

[26] Christian Szegedy et al. "Rethinking the Inception Architecture for Computer Vision". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition)* (2016), pp. 2818–2826.

[27] Keras Team. 2015. URL: https://keras.io/examples/mnist_acgan/ (visited on 04/03/2019).

[28] Lantao Yu et al. "SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient". In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*. 2017, pp. 2852–2858.