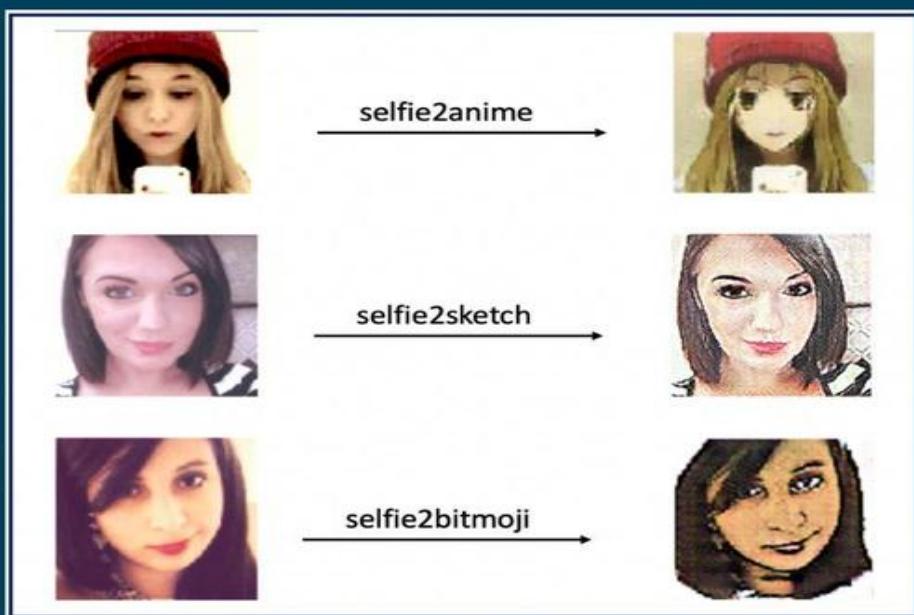


Hands on

THE GAN BOOK

>>> Train stable Generative Adversarial Networks using TensorFlow2, Keras and Python.



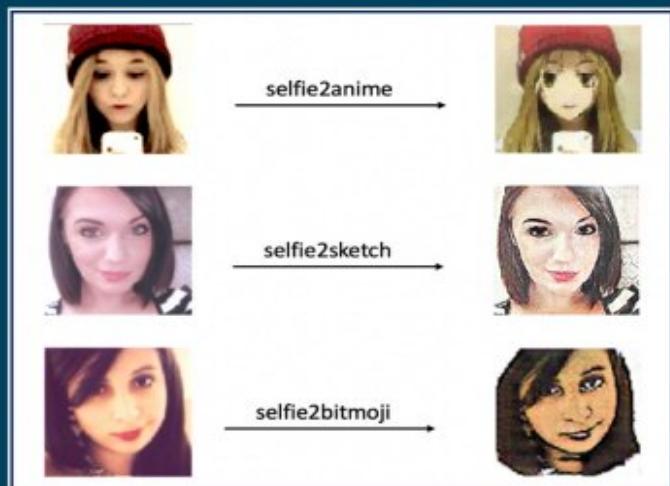
FUN EXPERIMENTS

KARTIK CHAUDHARY

Hands on

THE GAN BOOK

>>> Train stable Generative Adversarial Networks
using TensorFlow2, Keras and Python.



FUN EXPERIMENTS

KARTIK CHAUDHARY

The GAN Book

Train stable Generative Adversarial Networks
using TensorFlow2, Keras and Python.

Kartik Chaudhary

To my mother, Smt. Sarita Devi, and my father, Mr. Inderpal Singh, for their sacrifices,

constant love, and never-ending support.

Thank you for teaching me to believe in myself, in God, and in my dreams.

To my little brother, Chakit Gill, for continuous encouragement, support and love.

Thanks for being my best friend; I am really proud of you.

To my friends and colleagues for their inspiration, motivation, and always being there for me.

And, most importantly, to all the readers - I hope this book helps you with your goals,

because that's the real motivation behind writing this book and every single technical article that I share publicly on my blog.

– Kartik Chaudhary

About the author

Kartik Chaudhary is an AI enthusiast, educator, and a ML professional with 6+ years of industry experience. He is currently working as a Senior AI Engineer with Google to design and architect ML solutions for Google's strategic enterprise level customers, leveraging core Google products, frameworks and AI tools. He has previously worked with UHG, as a Data Scientist and helped in making the healthcare system work better for everyone. Kartik has filed 9 patents at the intersection of ML, AI and Healthcare.

Kartik loves sharing knowledge and runs his own blog on AI, titled [Drops of AI](#).

Away from work, he loves watching anime and movies and capturing the beauty of sunsets.

*I would like to thank my parents, my brother, And
my friends for their constant love and support.*

TABLE OF CONTENTS

Preface

Skill 1: Generative Learning

What are Generative Models?

Generative vs Discriminative Learning

How does Generative Learning work?

What are Deep Generative Models?

What are Autoregressive Generative Models?

What are Variational Autoencoders?

What are Generative Adversarial Networks?

What are the qualities of a good Generative Model?

Experiment: VAE for Digit generation

Experiment: Pixel CNN for Handwritten Digits

Skill 2: Generative Adversarial Networks

What are Generative Adversarial Networks?

What is GAN objective?

How to train a GAN based model?

What are the pros and cons of GAN?

Experiment: GAN for Handwritten Digits Generation

Skill 3: GAN Failure Modes

Why GAN training is unstable?

What are GAN failure modes?

What is mode collapse in GAN training?

What is convergence failure in GAN training?

Experiment: Mode Collapse in GAN training

Experiment: Convergence Failure in GANs

Skill 4: Deep Convolutional GANs

What is a Deep Convolutional GAN or DCGAN?

Guidelines for stable training of DCGANs

Impressive Image Generation results of DCGANs

Unsupervised Representations Learning

Experiment: DCGAN for Fashion MNIST

Experiment: DCGAN for Anime Face Generation

Experiment: DCGAN for Human Face Generation

Skill 4(II): Into the Latent Space

Making sense of the Latent Space

Experiment: Into the Latent Space: Anime Model

Experiment: Into the Latent Space: Human Face Model

Skill 5: Towards stable GANs

Best Practices for Training Stable GANs

Skill 6: Conditional GANs

What are conditional GANs?

Introduction to CGAN

Introduction to SGAN

Introduction to Info-GAN

Introduction to ACGAN

Experiment: CGAN for MNIST

Experiment: SSGAN or SGAN for Fashion MNIST

[Experiment: Info GAN for MNIST Handwritten Digits](#)

[Experiment: ACGAN on Fashion MNIST](#)

Skill 7: Better Loss functions

[Why other loss functions?](#)

[What is a Wasserstein GAN \(WGAN\)?](#)

[Experiment: WGAN on MNIST Digits Dataset](#)

[What is least squares GAN \(LSGAN\)?](#)

[Experiment: LSGAN for MNIST Handwritten Digits](#)

[Improving WGAN using gradient penalty \(WGAN-GP\)](#)

[Experiment: WGAN-GP for Fashion MNIST Dataset](#)

Skill 8: Image-to-Image Translation

[Image-to-Image Translation with GANs](#)

[Supervised Image-to-Image Translation](#)

[Unsupervised Image-to-Image Translation](#)

[Experiment: Pix2Pix for Black-n-White to Color Images](#)

[Experiment: Pix2Pix for Google Maps Experiment](#)

[Experiment: Cycle GAN for Apples to Oranges translation](#)

[Experiment: Cycle GAN for Horses to Zebras translation](#)

Skill 9: Other GANs and experiments

[Super Resolution GAN or SRGAN](#)

[Experiment: Super Resolution GAN or SRGAN](#)

[Disco GAN](#)

[Experiment: Disco GAN for Male to Female Experiment](#)

[Cartoon-GAN](#)

Experiment: Customized Cartoon GAN for 3 Experiments

Context Encoder

Experiment: Context Encoder Experiment

Skill 9(II): Advanced Scaling of GANs

BigGAN for Generating High-Resolution Images

Progressive Growing of Generative Adversarial Network

StyleGAN for Generating Photorealistic Human Faces

Skill 10: How to evaluate GANs?

Challenges of Evaluating GANs

Manual Evaluation of GANs

Qualitative Evaluation of GANs

Quantitative Evaluation of GANs

Skill 11: Adversarial Examples

What is Adversarial Machine Learning?

What are common types of Adversarial Attacks?

How to defend against Adversarial Attacks?

Why it is hard to defend against Adversarial Attacks?

Skill 12: Impressive Applications of GANs

10 Impressive Applications of GANs

Skill 13: Top Research Papers

Top 20 Research Papers on GANs

Bibliography

PREFACE

Hello there!

The GAN book is a comprehensive guide that highlights the common challenges of training GANs and also provides guidelines for developing GANs in such a way that they result in stable training and high-quality results. This book also explains the generative learning approach of training ML models and its key differences from the discriminative learning approach. After covering the different generative learning approaches, this book deeps dive more into the Generative Adversarial Network and their key variants.

This book takes a hands-on approach and implements multiple generative models such as *Pixel CNN*, *VAE*, *GAN*, *DCGAN*, *CGAN*, *SGAN*, *InfoGAN*, *ACGAN*, *WGAN*, *LSGAN*, *WGAN-GP*, *Pix2Pix*, *CycleGAN*, *SRGAN*, *DiscoGAN*, *CartoonGAN*, *Context Encoder* and so on. It also provides a detailed explanation of some advanced GAN variants such as *BigGAN*, *PGGAN*, *StyleGAN* and so on. This book will make you a GAN champion in no time.

Who this book is for

If you are a ML practitioner who wants to learn about generative learning approaches and get expertise in Generative Adversarial Networks for generating high-quality and realistic content, this book is for you. Starting from a gentle introduction to the generative learning approaches, this book takes you through different variants of GANs, explaining some key technical and intuitive aspects about them. This book provides hands-on examples of multiple GAN variants and also, explains different ways to evaluate them. It covers key applications of GANs and also, explains the adversarial examples.

What this book covers

Skill 1, Generative Learning, provides and introduction to the generative learning approach of training ML models and its key differences from the discriminative approach. It also covers different techniques of training or learning the generative models.

Skill 2, Generative Adversarial Networks, covers the basics of Generative Adversarial Networks and its objective function.

Skill 3, GAN Failure Modes, explains two common training failure scenarios for GANs, using experiments for recreating them. It also highlights the possible reasons for a training failure.

Skill 4, Deep Convolutional GANs, covers the CNN based DCGAN model. It also covers some best-practices and experiments for developing DCGAN for stable training and better results.

Skill 4(II), Into the Latent Space, explores the latent space of the trained generator networks of GANs. It shows some interesting findings about the latent space of a trained GAN based model.

Skill 5, Towards stable GANs, covers some of the common best practices for developing and training stable GAN based models.

Skill 6, Conditional GANs, covers different variants of conditional GANs such as CGAN, SSGAN, Info GAN and ACGAN. It also covers experiments related to these variants of conditional GANs.

Skill 7, Better Loss functions, explores different loss functions for developing and training stable GANs. This skill also covers the hands on experiments related to WGAN, WGAN-GP and LSGAN variants.

Skill 8, Image-to-Image Translation, explains the image-to-image translation application of GANs.

Skill 9, Other GANs and experiments, covers some other popular GAN variants and their applications. It also covers some hands on experiments related to those variants.

Skill 9(II), Advanced Scaling of GANs, covers some best practices for scaling GANs. It shows how to develop GANs for generating high-quality and

high-resolution images. This skill covers the following GAN variants: BigGAN, PGGAN and StyleGAN.

Skill 10, How to evaluate GANs?, covers some of the common evaluation techniques for GANs.

Skill 11, Adversarial Examples, explains the adversarial examples and different ways to defend the ML models against them.

Skill 12, Impressive Applications of GANs, covers some of the common application areas of GAN based generative models.

Skill 13, Top Research Papers, lists down top 20 research papers related to GANs that will help you in becoming a GAN expert.

To get the most out of this book

You will need to have a basic understanding of machine learning (ML) and deep learning (DL) techniques. You should also have beginner level experience with Python programming language.

Example code files

The code samples within this book are given just for the understanding purposes. If you want to try out some experiments, I would recommend you to download the code files from the Github repository of this book at: <https://github.com/kartikgill/The-GAN-Book>. If there is an update to the code, it will be updated in the Github repository.

Get in touch

Your valuable feedback is always welcome!

If you want a **free PDF version of this book**, feel free to drop an email.

You can reach out to me via email (kartikgill96@gmail.com) for any queries about the book. Feel free to connect over *LinkedIn* and stay tuned about my upcoming projects.

Homepage Link: <https://kartikgill.github.io/>

Personal Blog: <https://dropsofai.com/>

LinkedIn: <https://www.linkedin.com/in/chaudharykartik/>

Disclaimer

The information contained within this eBook is strictly for educational purposes. If you wish to apply ideas contained in this eBook, you are taking full responsibility for your actions. The author has made every effort to ensure the accuracy of the information within this book was correct at time of publication. The author does not assume and hereby disclaims any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from accident, negligence, or any other cause. No part of this eBook may be reproduced or transmitted in any form or by any means, electronic or mechanical, recording or by any information storage and retrieval system, without written permission from the author.

Copyright

The GAN Book

© Copyright 2024 Kartik Chaudhary. All rights reserved.

Let's get started!

Skill 1

Generative Learning

The term '**Artificial Intelligence**', or AI for short, refers to a branch of computer science concerned with making intelligent machines that are capable of doing amazing things as if there is a brain inside them. It doesn't mean that AI systems really have a brain and they are capable of understanding the world just like humans. It actually means that the modern AI systems can be designed or trained to solve some specific tasks smartly (as if there is some intelligence involved). An AI system could be very simple made-up of a few hardcoded if-else statements; also, it could also be a very complex system capable of solving a complex problem. For example, AI based language translation models that are capable of translating any given language to the language of our choice, are quite complex.

Machine Learning (ML) is a subfield of AI that gives the machines, an ability to learn things with experience. AI systems designed using ML techniques often start very dumb and become smart with experience and this experience is usually gained from the historical data. The field of ML has become quite popular over past few decades, as it has solved many complex real-world problems that seemed impossible to solve using deterministic algorithms (or hardcoded rules). There are several ML algorithms (or approaches) out there, each with its own pros and cons, but almost all of these methods have one thing in common – they require large amount of quality data for the learning purpose.

Deep Learning, or **DL** for short, is a particular type of ML technique, that is inspired from the function of a biological brain (human brain). Just like our brain uses biological neurons and activations to pass information around, DL systems such as **Artificial Neural Networks (ANNs)** are designed to learn in a similar way. However, there are significant differences between the learning objective of an ANN and how our brain works. Recent breakthroughs in the field of DL have led to the development of tons of neural network-based solutions, capable of solving many complex real-world

problems as accurately as humans and many times going even beyond the human level. Examples of these breakthroughs include state-of-the-art face-recognition systems, speech-to-text models, optical-character-recognition models, language-translation systems, text-to-speech, virtual-assistants and so on.

Researchers and Data Scientists, working in the field of ML and AI, are continuously developing new ways of making AI systems better at understanding the real-world. For people like us, understanding the world means: making use of our eyes, ears, hands, nose etc. to continuously assess the surroundings and taking decisions that are sensible. Decisions such as '**not going in front of a speeding car**', '**not jumping from the top of a tall building**' and '**helping grand-parents in finding their things**' and so on. Giving this kind of understanding of the world to the dumb machines (dumb because there is no inherent brain inside them), is a highly complex task and based on the progress made until today, we are not even close. The development of self-driving cars that are self-aware, is a big step towards understanding the real-world but still, driving a car is just a fraction of the things that humans do in their daily life.

In the field of AI and ML, research is progressing at a very high pace and we will continue to see big breakthroughs in the future as well. AI and ML are going to create wonders by solving problems that might seem impossible as of now. Recent breakthroughs such as Generative Adversarial Networks, Diffusion models and Large Language Models, are solving many complex problems today that seemed impossible just a few years back.

In this skill, we will discuss the generative learning approach of developing ML models and its key differences from the discriminative learning approach. We will look into different ways of developing generative models including – Autoregressive Models, Variational Autoencoders and Generative Adversarial Networks. Finally, we will also develop a Variational Autoencoder and a Pixel CNN based model in python for generating handwritten digits. Specifically, this skill covers the following topics:

- What are Generative Models?
- Generative vs Discriminative Learning
- How does Generative Learning work?

- What are Deep Generative Models?
- What are Autoregressive Generative Models?
- What are Variational Autoencoders?
- What are Generative Adversarial Networks?
- What are the qualities of a good Generative Model?
- **Experiment:** Variational Autoencoder for Digit generation

Let's get started.

1. What are Generative Models?

Generative Learning refers to a special class of statistical models that are capable of generating content that is very hard to distinguish from the reality (or fake content that looks real). The generated content could be poems, images, music, songs, videos, 3D objects, or some content from a new domain we could imagine. A domain is nothing but a fancy word for a bunch of examples that follow some common pattern. Interesting part is that, sometimes, the generated content is not just realistic, but it's completely new as well (or unseen in the training examples). Everyone must have seen or heard about the modern technologies that can generate very realistic looking faces of the people that do not even exist in the world. Projects such as Face aging apps, Virtual try-on, converting photos to paintings, and a lot more advancements with similar technologies are examples of the modern generative models.

Now the question comes – Is every ML model generative in nature? **Well, No!**

ML models can be broadly classified into the following two categories:

- ***Discriminative Models***
- ***Generative Models***

Let's understand these two categories in more details.

1. Discriminative Models

As the name suggests, the discriminative models are used for discriminative tasks such as predicting whether there is a **Dog** present in an image or a **Cat**. In ML applications, the discriminative models are quite popular and are heavily used for classification tasks such as Sentiment Classification, Classifying emails into spam vs not spam, Image Classification and so on. In the next paragraph, we will understand how does the learning process work for the discriminative models.

The discriminative models are presented with a large number of training pairs of type (x,y) where x represents the observation and y represents the corresponding outcome (also known as the label). The objective of the ML model is to learn a mapping function from x to y , such that when presented with some new observations in the future, it should be able to automatically calculate (or predict) the most likely outcome (or label). A sufficiently deep Neural Network (NN), provided with sufficient number of labelled observations, can learn the mapping function between the observations and the labels efficiently through backpropagation by utilizing any stochastic gradient descent-based optimization algorithm (also termed as: Optimizer).

In order to learn this mapping function, the discriminative models rely upon labelled datasets. In many real-world applications, it can be difficult to gather sufficient amount of labelled data every time. Generative models, however, do not always require labelled datasets as they have a completely different type of objective function to optimize. Let's get a quick understanding of the generative models next.

1. Generative Models

As discussed earlier, the generative models are special type of ML models that are capable of generating realistic content. A ML model or any technology in particular or even a human mind can only generate realistic content when it is aware of almost every important detail about the target content, which can also be termed as the domain understanding. To achieve this goal, A generative learning approach aims at learning the underlying distribution of the target domain (where, the target domain is the domain of

the content that we want to generate). Once our model knows the true distribution of data, we can keep sampling from it and generate infinite volume of content that follows the same data distribution.

It may sound easy but learning a distribution is not a trivial task. We will soon talk about the challenges of learning a data distribution but before that it's important to properly understand the differences between a generative and a discriminative learning approach of ML. Understanding the key differences between the two aforementioned approaches is important; and it will help us in following the forthcoming content of this book which is mostly related to the generative models. Let's look at both the approaches to get a better understanding.

1. Generative vs Discriminative Learning

The Generative approach of the statistical modelling (or ML) aims at learning the joint probability distribution $p(x,y)$ over the given pairs of observations and corresponding labels (x,y) or just $p(x)$ when labels are not present (as discussed earlier, the generative models don't always require labelled data). Because $p(x)$ represents the data distribution of the input samples x , sampling from $p(x)$ would generate a new sample every time.

Apart from generating data, the generative models can also be utilized for estimating the conditional probability $p(y|x)$ using the Bayes rules (with the help of learned joint distribution $p(x,y)$) to make their predictions by choosing the most likely label y for a given input observation x . Here is how the conditional probability $p(y|x)$ can be estimated:

$$p(x) = \sum p(x,y)$$

$$p(y|x) = \frac{p(x,y)}{p(x)}$$

A discriminative approach on the other hand, as discussed earlier as well, estimates the conditional probability (or posterior) $p(y|x)$ directly from the observations x and the corresponding labels y without worrying about the underlying data distribution (basically they learn just the mapping function from observations to labels). It makes the task of a discriminative approach pretty straight forward as the objective is just to learn a mapping function (also known as a classifier or a regressor) between x and y .

In simpler words, A generative model learns the distribution first and then decides the most likely output while a discriminative model learns the direct mappings between the inputs and the class labels (based on similarities or dissimilarities).

The discriminative approach is usually preferred when the task is about solving a classification problem, or an easy problem. A generative model, on the other hand, picks up the complex task of learning a data distribution, the harder problem. Most of the times, learning a data distribution may not be important and thus having a discriminative approach makes sense to keep the things simpler. Let's look at the following examples.

Example: In case of binary classification, all we need to do is to learn a decision boundary that separates two classes with minimum error. With this boundary, the model can decide whether a new data point belongs to class A or class B without worrying about the data distributions (see Figure 1.1).

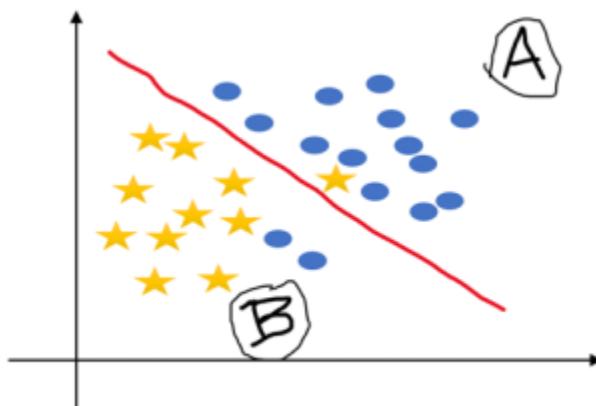


Figure 1.1: Decision Boundary (A discriminative approach)

Example 2: Both approaches have their own ways of solving problems. Let's look at one more example to understand the difference between generative and discriminative learning approaches. In this example, we will start by giving a task and then see how each of the approaches goes about solving it.

Task: Identify the animal in a given photograph?

Generative Approach: Study all the animals (and their characteristics) in the world and then determine which animal is present in the given picture. This approach looks at the low-level attributes such as eyes, face, legs, tail, color, height and so on, to decide the final outcome.

Discriminative Approach: No need to learn about any of the animals, simply look at the structural (or shape) differences or similarities and decide the animal. This approach usually looks at the high-level features such as structure and shape to draw a decision boundary between different animals.

Note: Based on the above definitions, one might think that ML models are always probabilistic in nature (as we discussed about estimating the prior and posterior distributions, in terms of probability), but a generative or discriminative model does not always need to output probabilities to be considered as a valid model. For example: A decision tree-based classifier, directly gives the output class without estimating any probability value and is still a valid discriminative approach. Because the predicted labels follow the distribution of the real labels provided as training data.

Now that we have a good background about the generative approach of solving ML problems, let's look at some common generative approaches that have been frequently used. Check out the following list of Generative Approaches (source: Wikipedia).

Gaussian mixture model

Hidden Markov model

Probabilistic context-free grammar

Bayesian network (e.g. Naive bayes, Autoregressive model)

Averaged one-dependence estimators

Latent Dirichlet allocation

- Boltzmann machine
- Flow-based generative model
- Energy based model
- Variational autoencoder
- Generative adversarial network

Discriminative approaches, on the other hand, are very frequently used for solving real-world business problems due to their simplistic nature. Following is a list of commonly applied discriminative approaches in past few decades (source: Wikipedia).

- k-nearest neighbours algorithm
- Logistic regression
- Support Vector Machines
- Decision Trees
- Random Forest
- Maximum-entropy Markov models
- Conditional random fields
- Neural networks

We now have a good enough understanding of the two ML approaches – Generative learning and Discriminative Learning. As this book is mainly focused on the generative learning approach, we will mostly talk about the generative models henceforth. Let's get into more details about the generative learning approach.

1. How does Generative Learning work?

To understand how exactly the generative learning works, let's first define an example problem and then we will solve it using a generative model. Let's assume that we have a dataset (D) of **1 million** cat images representing multiple breeds of cats across the world and the photographs have been taken

from almost all possible angles. Note that the number 1 million is significant here, as generative models generally require larger datasets to estimate the target distribution more accurately.

Because a generative learning approach estimates the data distribution to solve a problem, our focus is to define a generative model that is capable of learning the distribution (P_{cats}) that these cat images represent. Note that every dataset represents some data distribution that it is originally sampled from, and that data distribution is known as the true distribution of that particular dataset. Here P_{cats} is the distribution of all possible cat images in the universe and this dataset is sampled from it as a representative of the true data distribution.

If somehow, our model is able to learn the distribution P_{cats} , It will be able to answer all possible questions about cats present in this universe. For example –

- It will be able to tell whether a given image x represents a cat or not. If the likelihood value $p(x)$ is high, then x is definitely a cat or vice-versa.
- Secondly, if you go ahead and sample an image from $p(x)$, it will always be a cat image. In this way, it will be able to generate cat images infinitely.

This example gave us a much better understanding of the generative models. We now understand that a generative model first learns the underlying data distribution so that later it could answer any questions about that data. But in reality, learning a data distribution is not trivial. To understand how complex, it can be to learn a joint distribution, we first need to understand what does a joint distribution actually mean? The following subsection explains the joint distributions.

3.1 Joint Distribution

As we all are aware, the digital images are made up of pixels. Each pixel inside an image, represents a color and a group of such color pixels, may

represent the objects inside that image. In digital computers and smartphones, each pixel is represented using three discrete random variables R, G and B representing the intensity of three colors – Red, Green and Blue.

In a given digital image, each color pixel represented by these three random discrete variables, can choose any random discrete integer value from the range **[0, 255]** for each variable R, G and B. We can represent the joint distribution of a single-colored pixel by $p(R,G,B)$ such that sampling from this distribution $(r,g,b) \sim p(R,G,B)$ always generates a colorful pixel. In this case, the total number of parameters required to specify the joint distribution $p(R,G,B)$ would be:

$$= 256 \times 256 \times 256 - 1 = 256^3 - 1$$

Here, as each random variable has 256 possible values (intensity of color), so total parameters required to specify this true distribution would be one less than the total possible combinations, as shown in the calculation above.

This was just a single pixel, now think about an image with 100×100 dimensions (though it's a pretty low-resolution image in modern era) that is made up of 10,000 such colorful pixels. Now, can you imagine the number of parameters required to represent a true joint distribution of all such possible 100×100 dimensional-color images? Pretty huge right. Let's calculate it. We just need to multiply the number of possible combinations of one colored pixel ten thousand time. Check out the following calculation.

$$= (256^3 - 1) \times (256^3 - 1) \times \dots \text{ 10,000 times}$$

$$= 256^{30,000} \text{ (approximately)}$$

This number is pretty huge. Now if I ask you, can you prepare a dataset that can efficiently represent the above-described distribution of the color images with 100×100 resolution? The answer is pretty obvious –

“Never”.

It is impossible to practically represent the true data distribution in this case, no matter how big dataset you have, it's never enough. Any given dataset, representing a distribution P_{Dataset} , is a “**not very efficient**” representative of the true data distribution. Now, one question that pops up in

our mind is: Do we really need to model the true joint distribution? Can we settle for less (something like P_{Dataset})?

Actually, modelling the true distributions is pointless as they are deterministic in nature. In other words, if we already have the required information about the true distribution, we don't really need to model it. For example: consider the distribution of all the possible colorful images of dimensions 100 x 100, we don't actually need to model it. Because we already know that any random color image of 100 x 100 dimensions will always belong to the aforementioned distribution with a 100% confidence. Thus, there is no point in learning such distributions.

The aforementioned data distribution is deterministic in nature, because we assumed the pixels to be independent from each other. But what if the pixels are somehow related? This relation between pixels can restrict the given true distribution to represent only a particular class of color images, such as the dataset of **1 million cats** where pixels are not independent.

The dataset of 1 million cat images can be considered as a restricted distribution due to the pixel relationships. Learning this kind of restricted distribution, instead of the true joint distribution described above, can be helpful. To understand this, let's get into more details about the restricted distributions next.

3.2 Restricted Distribution

Now let's get back to our dataset of 1 million cat images. Let's assume that our dataset has the distribution P_{cats} which is supposed to be very close to the real distribution of all the possible cats in this universe. Now suppose, we are able to learn a generative model P_{Model} (model distribution) such that P_{Model} is very close to P_{cats} (from our dataset).

Using this model distribution (P_{Model}), we should be able to perform the following tasks such as –

- **Generation:** Sampling from the model ($\mathbf{x}_{\text{new}} \sim P_{\text{Model}}$) will always generate a cat image and it will give us the flexibility of generating

infinite number of cat images if required. (See example in Figure 1.2).

- **Prediction:** It will be able to tell whether a given image x , represents a cat or not. If the likelihood value $P_{\text{Model}}(x)$ is high, x is a cat or vice-versa.
- **Representation Learning:** The model will be capable of learning the unsupervised features related to cats such as breed, color, eyes, tail and so on without explicitly providing labels for these attributes.

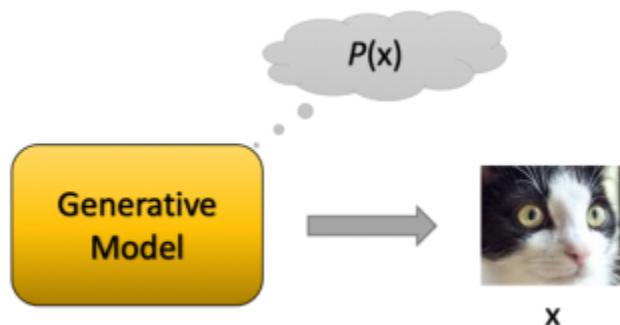


Figure 1.2: Generated sample x , sampled from probability distribution $p(x)$

Given the above notion, a conditional generative model is also possible. Suppose that we want to generate a set of variables: Y , given some other set of variables: X , we can directly train a generative model to learn the conditional distribution $P(Y|X)$ without worrying about the joint distribution. This is very similar to the sequence generation tasks where the next candidate of the sequence is predicted given some already existing candidates. Another popular example of conditional generative models is: Latent variable based generative models. Let's discuss how latent variable based generative models actually work.

3.3 Latent variable based Generative Models

To understand the latent variable based generative models, let's get back to our dataset of 1 million cat images. This dataset is not annotated and it

means that there is no information about the type of cat, that is present in a given picture. Now suppose that we want to train a generative model on this dataset, so that later we can use it to generate a few cat images. But this time we would like our model to generate the images of desired type of cats, instead of generating the random cat images. This time, we are asking the model to learn the unsupervised features as well, along with the data distribution so that it is able to answer questions like: Generate an orange long-haired cat image! The term ‘unsupervised features’ makes sense here because we are not providing the model with any labelled information for learning these features.

Latent Variable: A Latent variable is a variable that is hidden or that is not directly observed but is actually inferred from other variables that are observed.

The idea is to learn these unsupervised features, such as colors, hair-length, poses and so on, with the help of a latent vector (\mathbf{z}). Here, the latent vector \mathbf{z} is expected to represent these high-level features of cat images. In this case, a cat image of desired type can be sampled from the conditional distribution $p(x|z)$, if we are able to provide the correct value of \mathbf{z} here. Now, our objective has changed and our new goal is to learn the conditional distribution $p(x|z)$, instead of the joint distribution $P(x)$ which was more complex. Figure 1.3 shows the high-level idea of sampling from this new model, this time sampling is conditioned on the latent input.

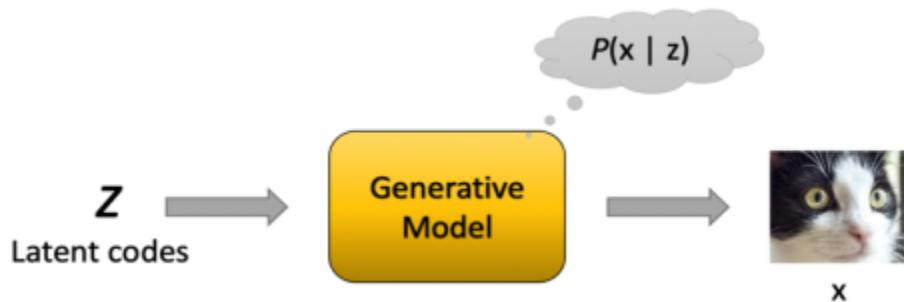


Figure 1.3: Generated Sample x , sampled from conditional distribution $p(x | z)$

Now the real question is: How do we know what value of \mathbf{z} generates which type of cat image? Because the training is also completely unsupervised (due to unlabeled dataset), we can't really have control over the latent variables. But here the trick, we will let our model learn the conditional distribution $p(\mathbf{x}|\mathbf{z})$ and then, we can simply reverse the equation and learn the reverse conditional distribution $p(\mathbf{z}|\mathbf{x})$. Using this simple trick, we will be able to map the generated cats back to their input latent vector \mathbf{z} , and hence, we will be able to find out what value of \mathbf{z} generates which type of cat image.

Our plan seems simple, but again, learning this conditional distribution $p(\mathbf{x}|\mathbf{z})$ is a challenging task and we cannot be 100% sure that it will learn the desired features corresponding to different values of the latent vector \mathbf{z} (so that we could map them back by reversing the process), as the training is completely unsupervised (or uncontrollable).

Another important thing to consider is that we are talking about training models to learn the distributions here. To accomplish that –

“We need a way to compare distributions”!

As we have discussed earlier as well, the real job of a generative model is to learn the distribution P_{Model} which is really close to the true data distribution P_{TRUE} (here, P_{Dataset} can be a good proxy for the true distribution). During the learning phase, a generative model may request to know how close the model distribution (P_{Model}) is to the expected distribution (P_{Dataset}). For this reason, we need a function that is capable of efficiently comparing and quantifying the closeness of two distributions (e.g., P_{Model} and P_{Dataset}). We shall talk about comparing distributions shortly in this skill but before that let's discuss a little bit more about the deep generative models.

1. What are Deep Generative Models?

Recent advances in the field of deep learning have led to the development of complex generative models that are capable of generating high quality content in the form of text, audio, pictures, videos and so on. Generative

models that make use of deep learning architectures to tackle the task of learning distributions, are known as deep generative models. Due to the flexibility and scalability of neural networks, deep generative models have become the most exciting and swiftly evolving field of ML and AI. Deep generative modeling techniques have helped in developing modern AI agents that are constantly generating and processing vast amounts of data.

Generative learning with deep neural networks is creating wonders today. There are many popular deep generative learning frameworks that are quite an active area of research. In this skill, we will cover three such popular frameworks. Later in this section, we will discuss the following three deep generative learning frameworks –

- **Autoregressive Generative Models**
- **Variational Autoencoders**
- **Generative Adversarial Networks**

Before jumping right into the generative modelling frameworks, it's important to understand few things about the probabilistic distributions. This knowledge will help us in understanding and also discovering new ways of comparing distributions. We will now talk about the following concepts related to the probabilistic distributions –

- **Maximum Likelihood Estimation (MLE)**
- **Kullback-Leibler or KL Divergence**
- **Jenson-Shannon or JS Divergence**

Let's learn about each of these concepts now.

4.1 Maximum Likelihood Estimation (MLE)

Maximum Likelihood Estimation, or MLE for short, is a method of estimating (or learning) a probability distribution. As per our understanding until now, we want our generative model to learn a distribution P_{Model} that best describes the given dataset. It means that any given sample x from the

dataset must be very likely as per the model. In other words, the likelihood value $P_{\text{Model}}(x)$ is high for all the samples in the dataset.

As the name suggests, the method ‘Maximum Likelihood Estimation’ takes this notion very seriously and provides an objective function that makes the given dataset (or observed data) most likely under the model distribution. The state or point of model parameters, that makes the observed dataset most likely, is known as the maximum likelihood estimate.

In statistical terms, the likelihood function is defined over the model parameter space and the objective is to find a subset of the Euclidean space (a space defined by possible ranges of the model parameters) that maximizes the value of likelihood function (for the observed data obviously). If the likelihood function is differentiable, we can take advantage of the gradient descent method to learn the optimal parameters. Let’s now understand the likelihood function in mathematical terms as well.

Suppose, we have a likelihood function f (or model) defined over the parameter space (Θ). Here, the likelihood objective (\hat{l}) for the given data samples (x_1, x_2, \dots, x_n) can be defined by the following equation –

$$\hat{l}(\theta; x) = \frac{1}{n} \sum_{i=1}^n \ln f(x_i | \theta)$$

This equation assumes the data samples to be identically distributed and independent. Question: why do we have log in this equation?

Answer: It’s often convenient to work with log-likelihoods as the optimal solution ($\hat{\theta}$) still remains same.

We now have defined a simple likelihood function. We can now maximize this function for a given dataset to get the maximum likelihood estimate or a trained model that understands the dataset distribution. We now have a good idea about how MLE works, let’s now look at the second concept – Kullback-Leibler divergence.

4.2 Kullback-Leibler or KL Divergence

Kullback-Leibler Divergence, or KL divergence, or D_{KL} for short, is also called relative entropy. KL divergence is a statistical way of calculating or quantifying the distance between two probability distributions. A relative entropy of zero, confirms that two distributions are identical.

Given any two discrete probability distributions \mathbf{P} and \mathbf{Q} , the relative entropy over the same probability space (S) can be calculated using the following equation:

$$D_{KL}(P \parallel Q) = \sum_{x \in S} P(x) \log \frac{P(x)}{Q(x)}$$

Or,

$$D_{KL}(P \parallel Q) = - \sum_{x \in S} P(x) \log \frac{Q(x)}{P(x)}$$

The relative entropy, $D_{KL}(P \parallel Q)$ can be understood as the relative entropy of distribution \mathbf{P} with respect to \mathbf{Q} (or the divergence of \mathbf{P} from \mathbf{Q}). Because KL divergence is an asymmetric measure, it does not obey the triangular inequality. And thus, it does not qualify as a true statistical metric of spread.

In simpler terms,

$$D_{KL}(P \parallel Q) \neq D_{KL}(Q \parallel P)$$

Similar to KL divergence, Jenson-Shannon divergence can also be used for comparing distributions. Let's see how it works.

4.3 Jenson-Shannon Divergence

Jenson-Shannon Divergence, or JSD for short, is again a method of measuring the similarity between two probability distributions, very similar to KL divergence. JSD is actually a clever modification of KL divergence that makes it symmetric and smooth, and hence a true statistical metric. The square root of JSD is also a metric, known as Jenson-Shannon distance.

JSD between two probability distributions \mathbf{P} and \mathbf{Q} can be defined as –

$$JSD(P \parallel Q) = \frac{1}{2} D_{KL}(P \parallel M) + \frac{1}{2} D_{KL}(Q \parallel M)$$

Where, M is the average of the two distributions and it can be written as –

$$M = \frac{1}{2} (P + Q)$$

As we can see from the above equations, JSD very cleverly makes use of KL divergence to arrive at a new symmetric metric. We now have a good understanding of three popular probabilistic methods – MLE, KL divergence and JSD. With this knowledge, we are ready to jump right into deep generative modelling frameworks. Let's start with the first framework – Autoregressive Generative Models.

1. Autoregressive Generative Models

The term '*autoregressive*' is taken from the field of time-series forecasting frameworks, where the model considers all the past observations in an ordered or timely manner, to make predictions about the future.

Autoregressive generative models are also quite similar in nature, as they also take help from all of their past predictions for deciding the subsequent prediction.

For example: an autoregressive model may generate an image by predicting one pixel at a time. Here each new pixel value is estimated based on the previously predicted or observed pixel values. In statistical terms, an autoregressive model is trained to learn the conditional distribution of each individual pixel given the surrounding pixels that are already known (or have already been estimated by the same model in earlier iterations).

Remember, this kind of sequence prediction paradigm is not new and has been successfully applied in the past in the fields of natural language processing to solve sequence learning tasks such as – predicting the next word in a sentence, given some already known words. But when it comes to applying these models on digital images, the job becomes way more complex, as the length of the sequences becomes quite huge.

For example: the task of generating a digital image of 100x100 resolution, can be considered as a sequence prediction task where the sequence length is 10,000 (this sequence can be obtained by arranging the image pixels using raster scan, or, reading the image pixels in left-to-right and top-to-bottom manner). Generating a sequence this long where each pixel is generated iteratively by conditioning over all previous pixels, is quite complex and a very time-consuming task.

In mathematical terms, the objective of an autoregressive generative model is to learn a data distribution using the maximum likelihood estimation (MLE) method. Imagine, we are given a ‘n-dimensional’ dataset to learn the distribution from. We can write the joint-distribution formula by chain rule as –

$$\begin{aligned} p(x_1, x_2, x_3, \dots, x_n) &= p(x_1) * p(x_2|x_1) * p(x_3|x_2, x_1) * \dots * p(x_n|x_n, \dots, x_2, x_1) \\ &= \prod_{i=1}^n p(x_i | x_{i-1}, x_{i-2}, \dots, x_2, x_1) \end{aligned}$$

The chain rule is a general way of calculating joint probability assuming no conditional independence between the random variables. And that is exactly how the autoregressive generative models work (bayesian networks with no conditional independence assumption).

To learn an autoregressive generative model, we first need to arrange all the random variables in a fixed order (such as x_1, x_2, \dots, x_n), and then learn a mapping function for each subsequent random variable in the following stepwise manner –

- x_1 needs to be randomly sampled from all possible values
- x_2 can be estimated using x_1
- x_3 can be estimated using x_1 and x_2

.....

.....

- x_n can be estimated using x_1, x_2, \dots, x_{n-1}

Here, each of the above steps, is a model that learns the linear, or non-linear in case of neural networks, combination of all previously estimated random variables to estimate the next random variable.

Each model has $O(n)$ parameters and we have ‘ n ’ such models. Thus, the overall setup has $O(n^2)$ parameters. Main drawback of this approach is that the data generation process is very slow. As we can only estimate variables in a sequential way (similar to the execution of a for loop in computer programming).

Pixel CNN, Pixel RNN, Character CNN, Character RNN and Wave-Net are some popular examples of the Deep Autoregressive Generative models. Some of these models have been successfully applied in the field of anomaly detection and adversarial attacks detection. We will learn about the adversarial attacks later in this book.

Now that we have gotten a very good understanding of autoregressive approach for training generative models. Let’s look at some of the pros and cons of this methodology –

Following is a list of some pros of autoregressive generative models:

- Easy to understand and calculate likelihoods
- Training process is supervised and very straight-forward

Let’s now look at some of the cons of autoregressive approach:

- Requires an ordering of random variables
- Generation process is sequential, hence very slow
- High likelihood does not guarantee better looking samples in practice
- Does not learn unsupervised features (or representations)

We just learned about the first approach of training generative models. We should now be able to develop autoregressive generative models on our own. Pixel-CNN is a popular model that follows the autoregressive approach. A python example of developing pixel-CNN for MNIST dataset is available towards the end of this skill, and it can also be found in the Github repository

of this book. With this, let's move to the second framework for developing deep generative models – **Variational Autoencoders**.

1. Variational Autoencoders

Variational Autoencoders, or **VAEs** for short, are latent variable based generative models. The name ‘Variational Autoencoder’ is meaningful here because the underlying architecture looks very similar to autoencoders, and the learning process for the latent representations, takes a variational approach.

The basic architecture of a VAE is quite similar to that of a classic autoencoder but the learning process is quite different. Unlike vanilla autoencoders, VAEs are generative models. VAEs have many practical applications. Let's get into the technical details about this framework.

As discussed earlier in this skill, the latent variable based generative models have the following properties –

- Latent variable based generative models optimize for learning the conditional distribution $p(x|z)$, instead of the complex $p(x)$ over the dataset.
- Learning unsupervised representations (or features) is also possible, as the input latent vector controls the type of generated content.
- Mapping between the unsupervised features and latent variables can be established by learning the posterior $p(z|x)$, by just reversing the objective.
- No way to control the features that a given latent vector is going to learn.
- More difficult to train as compared to the auto-regressive models

To summarize, the goal of a latent variable-based generative model is to learn the conditional distribution $p(x|z)$, so that it could later be used for generating the desired type of content, by passing relevant latent vector (z). Here, the latent vector z is a multi-dimensional vector of latent variables sampled from some probability density function $p(z)$ (where, the term $p(z)$ represents a probability distribution defined over z). Each latent variable, in the latent vector, is expected to learn a meaningful feature from the training data (as discussed, in an unsupervised way). Learning a reverse mapping function $p(z|x)$ is also possible, and it can help us in finding out the corresponding latent vector z , for a given input sample x from the dataset.

We now have built a good intuition about the science behind the latent variable based generative approaches. Now, let's get into more technical details about VAEs, their basic model architecture, and their learning process.

6.1 Understanding VAE Architecture

As the name suggests, the basic architecture of a VAE is quite similar to the classical autoencoder. VAE architecture is also made up of two parts: the encoder network and the decoder network.

In a classical autoencoder setup, the encoder and the decoder are usually implemented as two standalone neural networks. The encoder network accepts the training data as input and encodes it into a dense representation. The decoder network is expected to use this dense representation as input and generate the desired data as output. Now let's see how a VAE makes use of this setup for training a latent variable based generative model.

The encoder part of a VAE learns the sample to latent space mapping $p(z|x)$ (for each sample x in the dataset), while the decoder part learns to generate the data sample as output for a given latent vector z (vector z comes from the encoder network). In simpler words, the decoder basically learns $p(x|z)$. One obvious challenge with this setup is that the encoder network only learns a one-to-one mapping between the input samples and the latent vectors. So, the question comes: how should ensure if the model is learning correct $z \sim p(z)$ or not?

To overcome this challenge, VAE makes a small assumption about the latent space: It assumes that all the latent variables follow a unit gaussian distribution (or a standard normal distribution). This small assumption makes the learning process of a VAE meaningful. One good thing about the standard normal distributions is that they can be represented with just two simple statistics: the mean and the variance.

We will take advantage of this property of standard normal distributions, during the training of VAE. So basically, during the training process, we will expect the encoder part of our VAE to learn the inherent distribution of latent vectors (that we assumed to be a unit gaussian distribution), instead of directly estimating the latent vector for a given input data sample. Specifically, we will ask the encoder to generate the two statistics: the mean and the variance, corresponding to each latent variable of the latent vector. Note that a latent vector is composed of multiple latent variables (as discussed earlier in this skill as well). Now that the encoder model is giving a normal distribution as output, we can optimize it by bringing this output closer to the true distribution of the latent space. Remember, we assumed the latent space to follow a unit gaussian distribution per latent variable.

To quantify the closeness of the estimated distribution (the output of VAE encoder) with the true distribution (unit gaussian), we can use KL divergence here. As discussed earlier in this skill, KL divergence is a metric of similarity or distance between two probabilistic distributions. Our encoder network is now all set to learn the conditional distribution of the latent space with respect to the input data samples. Now let's see how can we use this information for actually generating data, as our main goal is to develop a generative model.

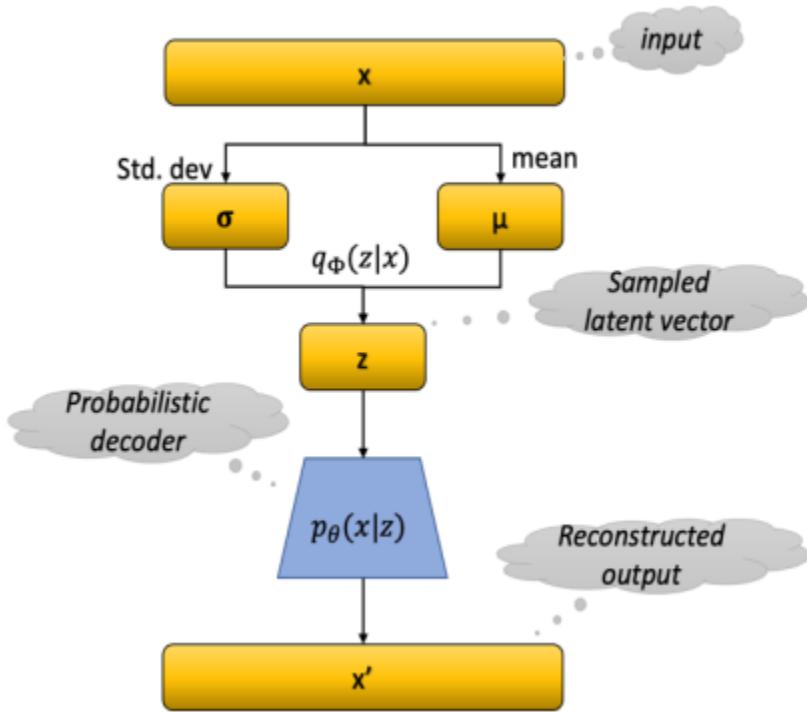


Figure 1.4 : Variational Auto-encoder architecture diagram

Once we have the latent distribution (from the output of encoder network), generating data samples is pretty straight-forward. We can always sample a latent vector $z \sim p(z)$ and the decoder will generate a data sample x using $p_\theta(x|z)$ (where $p(z)$ is the distribution of latent space that our encoder has already learned).

Because we want to train the entire setup end-to-end using backpropagation, the sampling part would also be part of the setup. In this case, the sampling technique should be a differentiable function, and thus we cannot rely on random sampling. But as we know that the distribution of the latent space is gaussian, we can sample a latent variable using the mathematical formula: (**mean + variance^{1/2} * epsilon**), here **epsilon** is a term to introduce variations and it is also sampled from the unit gaussian distribution. This sampled vector z can be fed as input to the decoder network and the decoder network can be trained to reconstruct the original input sample x using the reconstruction loss. (Note that the reconstruction loss can

be calculated by pixelwise comparisons of two images: the generated image and the input image).

Now, the overall setup is end-to-end trainable using the backpropagation technique, just like an ordinary autoencoder. Now, the overall objective of the VAE setup can be obtained by combining the loss function of both networks. We can write the overall objects as:

$$\text{Objective(VAE)} = \text{KL-loss(Encoder)} + \text{Reconstruction-loss(Decoder)}$$

VAE based generative models, though they are quite powerful and successfully applied in multiple application areas, are criticized for generating blurry images. And thus, they are not preferred for generating the high-quality images. Following are some pros and cons of the VAE based generative approach:

Pros:

- Flexible models, easy to build and train using backpropagation.
- Suitable for unsupervised feature learning, as labelled data is not required.

Cons:

- Hard to evaluate likelihoods.
- VAE approach makes assumptions when estimating $p(z|x)$.
- Generated samples are often blurry.

Now that we have a very good understanding of the VAE and its learning process, we are all set to implement it. VAE implementation example for image generation is explained towards the end of this skill. To keep the flow, let's first learn about the Generative Adversarial Networks.

1. Generative Adversarial Networks

Generative adversarial Networks (or **GANs**) are significantly different from the other generative learning frameworks (such as Autoregressive models, VAEs and normalizing flow models), because we do not train them using maximum likelihood. GANs are likelihood free so they don't have the issues that come from maximum likelihood estimations. Let's get into more details.

7.1 Towards Likelihood free Learning

An optimal model estimated using the maximum likelihood estimation (MLE) technique is expected to generate best quality samples and also assign high likelihood values to the real data (from the test set). However, this is not always the case, a model that generates good quality samples may not give high likelihoods for the test data and similarly, another model that gives high likelihoods to the test data, may generate poor samples. This kind of behavior from likelihood-based models encourages us to think about learning generative models in a likelihood-free manner. We need to consider a different way of comparing distributions that does not directly depend upon the likelihood function. Let's see how can we learn distributions without depending upon the likelihood function.

7.2 Two-Sample Test

A simple and likelihood free way of comparing two distributions could be: a two-sample test. Two-sample test is a statistical test that checks whether two given finite set of samples belong to the same distribution (or single distribution) or not. Suppose that S_1 is a finite set of samples from distribution P ($S_1 \sim P$) and S_2 is a finite set of samples from distribution Q ($S_2 \sim Q$), we can calculate some test statistic f , based on the differences between sets S_1 and S_2 . If the value of f is below a certain threshold, we can consider the distributions P and Q as same and different otherwise.

In a generative learning setup, we can take advantage of two-sample test to compare the samples from the dataset ($S_1 \sim p_{\text{data}}$) and the generated samples from the model ($S_2 \sim p_{\text{Model}}$). In this approach, we can optimize our model such that the value of test statistic- f is as low as possible. This will

bring the model distribution very close to the data distribution which is our ultimate goal as well. Now let's see how a GAN based approach work.

7.2 The GAN Framework

A Generative Adversarial Network, or **GAN** for short, is a generative learning framework that takes advantage of two-sample test method for learning the data distribution. A typical GAN based setup is composed of two component models: the generator model and the discriminator model. The learning process of a GAN can be understood as a two-player mini-max game between the generator and the discriminator network. In this game, the generator network tries to fool the discriminator by generating samples that look very similar to the real samples (from dataset), while the discriminator network tries its best to identify the fake samples (coming from the generator network output).

During the learning process, both networks keep getting better at their jobs and the training can be terminated when the discriminator is no longer able to distinguish the generated samples from the real ones. Formally, the objective of a GAN can be written as follows:

$$\min_{\theta} \max_{\phi} V(G_{\theta}, D_{\phi}) = E_{x \sim p_{\text{data}}} [\log D_{\phi}(x)] + E_{z \sim p_z} [\log(1 - D_{\phi}(G_{\theta}(z)))]$$

Don't worry, if you do not get this equation now, we will discuss the GAN objective in more details in the next skill.

We now have a very basic idea about the learning process of GANs. We will learn the GAN framework in details in the upcoming skills of this book. Following are some of the common pros and cons of GAN based generative learning approach:

Pros:

- Data generation process is faster than autoregressive models, also the generated content is high-quality, unlike the blurry results of VAEs.
- Likelihood-free, thus no issues in working with the high-dimensional data.
- GANs provide a good way of training the classifiers in semi-supervised manner (more details about this in the coming skills).

Cons:

- Training is unstable sometimes (However, there are some good practices to follow while developing GANs, we will learn them in coming skills).
- Hard to generate discrete data, such as text.

GANs currently generate the most plausible results (realistic results) and over the last decade, a lot of research has been done around GAN based generative models. In the subsequent skills, we will mainly focus on different variants, applications and implementation details of GANs in a very detailed manner.

We now know quite a few ways of developing generative models. Let's learn about the qualities of a good generative models.

1. Qualities of a Good Generative Model

Generative models are capable of generating realistic content from variety of domains such as Images, Audio, Text and so on. Depending upon the target domain and application, there are different ways possible for defining the quality measures for generative models. As this book is mostly focused on the task of Image generation, we will define our own quality measures that are slightly biased towards the task of Image data generation, instead of going very general. Another reason behind biasing the qualities towards the image data is that the recent advancements of GANs are also slightly biased towards image data generation. GANs have mostly been applied for Image generation tasks. However, it does not limit the potential of **GANs** as they are highly generic and don't favor any particular domain. It is certain that the researchers will keep finding new application domains for applying GANs.

As discussed earlier, we are mostly interested in the qualities of generative models for the task of image generation. Following are some basic qualities that can be assessed to measure the goodness of a generative model:

- Quality
- Quantity
- Control
- Complexity
- Stability

Let's get into more details about these qualities.

1. Quality

A good generative model is expected to generate very high-quality content. The generated content should look realistic and it should be very difficult to identify it as fake. There are multiple ways of quantifying the quality of generated images, we will learn about them at a later point in this book. Visually looking at the outputs is also a commonly used way of assessing the quality of image generation.

1. Quantity

A good generative model should be able to produce unlimited quantity of the content. Fundamentally, a well-trained generative model is a good representative of the target domain of data. And thus, it should be able to generate infinite quantity of content with varieties depending upon the domain. There are quantifiable measures for assessing the variety of generated content as well, and we will learn about them later in this book.

1. Control

There should be a way to control the variations of the generated content. For example: If a generative model is capable of generating pictures of animals, there should be a way to control what type of animal would be generated. It is important to get the desired content as output.

1. Complexity

The generative framework should be easy to understand and implement. It makes it more usable. If the framework is very complex and takes several months to train, it may not be very valuable for small business and researchers.

1. Stability

The training process of the framework should be stable, such that different engineers are able to replicate the same quality of results when trained the same model in similar conditions. If it works for me, it should work for everyone.

We have discussed the top five qualities that should be measured to assess the quality of image generation models. Depending upon the problem we are solving, we may have to add or remove any quality measure into this list but this is a good starting point to think about it. Now that we are towards the end of this chapter, let's go back to our experimentations of implementing a VAE and Pixel-CNN.

1. Experiments

We have already discussed some of the common generative learning approaches and their learning process. Now, it's time that we try implementing some of these techniques on our own. In this section, we will see two examples implementing two different generative learning models: **a VAE and a Pixel-CNN**. Specifically, we will perform the following two experiments:

- VAE for Digit Generation
- Pixel-CNN for Handwritten Digits

The code samples shown in this skill can be downloaded from the following Github location:

<https://github.com/kartikgill/The-GAN-Book/tree/main/Skill-01>

Let's get started.

VAE FOR DIGIT GENERATION

Objective

The main objective of this experiment is to develop a VAE based generative model that is capable of generating handwritten digits in different styles.

This experiment can be divided into the following small steps:

- Importing libraries
- Data Preparation
- Defining Encoder
- Defining Decoder
- Defining VAE
- Custom loss function
- Model Training
- Reconstruction Results
- Latent Distribution graph
- Data Generation

Let's get started.

Step 1: Importing Libraries

The very first step is to import useful python libraries into a cell of jupyter notebook. Here, we will be utilizing *matplotlib* and *seaborn* packages for visualization purposes and TensorFlow for developing the deep learning model. Check out the following snippet:

```
import numpy as np  
import matplotlib.pyplot as plt  
import pandas as pd  
import seaborn as sns  
import warnings  
warnings.filterwarnings('ignore')  
%matplotlib inline
```

```
import tensorflow
tensorflow.compat.v1.disable_eager_execution()
```

Now, let's move to the data preparation step.

Step 2: Data Preparation

In this experiment, we will utilize *MNIST handwritten digits* dataset. MNIST handwritten digits dataset is a collection images of handwritten digits (0-9) where each digit image has size 28 x 28. This dataset already has partitions with 60K train and 10K test samples along with the labels. It can be downloaded directly from the *tensorflow.keras.datasets* as shown in the following snippet. We will also plot some digit images to verify if data is correct.

```
from tensorflow.keras.datasets import mnist
(trainX, trainy), (testX, testy) = mnist.load_data()
print('Training data shapes: X=%s, y=%s' % \
      (trainX.shape, trainy.shape))
print('Testing data shapes: X=%s, y=%s' % \
      (testX.shape, testy.shape))
for k in range(9):
    plt.figure(figsize=(9,9))
    for j in range(9):
        i = np.random.randint(0, 10000)
        plt.subplot(990 + 1 + j)
        plt.imshow(trainX[i], cmap='gray_r')
        plt.axis('off')
    plt.show()
```

This python code generates the output as shown in Figure 1.5.

```
Training data shapes: X=(60000, 28, 28), y=(60000, )
Testing data shapes: X=(10000, 28, 28), y=(10000, )
```



Figure 1.5: MNIST handwritten Digits dataset overview

Now that we have downloaded and visualized the data, let's normalize it.

Data Normalization: It is advised to normalize image pixel intensities before passing into a neural network as it results in a faster and smoother convergence. Here, we scale the pixel intensity values from range [0, 255] to range [0, 1] by dividing each pixel intensity value by 255. Also, we add a new dimension to the input data to represent image channels (as supported by Convolutional Neural Networks, look at the following snippet: -

```
train_data = trainX.astype('float32')/255
test_data = testX.astype('float32')/255
train_data = np.reshape(train_data, (60000, 28, 28, 1))
test_data = np.reshape(test_data, (10000, 28, 28, 1))
print (train_data.shape, test_data.shape)
Out[3]: (60000, 28, 28, 1) (10000, 28, 28, 1)
```

Now, our data is ready. Let's work on the model architecture.

Step 3: Define Encoder

Now that our training and evaluation data is now ready, let's define the encoder part of the VAE. The encoder network of our VAE, should accept handwritten digit images ($28 \times 28 \times 1$) as input, and learn the latent distribution $p(z)$. Additionally, it should also return a latent vector z , sampled from the learned latent distribution.

The encoder network compresses the information present in an input image into a small dense vector, by passing it through a stack of convolutional layers, activations and pooling layers and finally a fully connected or dense layer. This compressed vector is expected to represent the important features about the input image. The following python code generates the aforementioned dense vector of size 16.

```
input_data = tensorflow.keras.layers.Input(shape=(28, 28, 1))
encoder = tensorflow.keras.layers.Conv2D(64, (5,5), activation='relu')(input_data)
encoder = tensorflow.keras.layers.MaxPooling2D((2,2))(encoder)
encoder = tensorflow.keras.layers.Conv2D(64, (3,3), activation='relu')(encoder)
encoder = tensorflow.keras.layers.MaxPooling2D((2,2))(encoder)
encoder = tensorflow.keras.layers.Conv2D(32, (3,3), activation='relu')(encoder)
encoder = tensorflow.keras.layers.MaxPooling2D((2,2))(encoder)
encoder = tensorflow.keras.layers.Flatten()(encoder)
encoder = tensorflow.keras.layers.Dense(16)(encoder)
```

As the final goal of our encoder network is to learn a latent distribution and return a sampled latent vector from it, we will utilize two fully connected layers to estimate distribution mean (of latent distribution) and log-variance from this compressed vector. (As discussed earlier, a standard normal distribution can be represented using mean and variance). See the following python code:

```
distribution_mean = tensorflow.keras.layers.Dense(2, name='mean')(encoder)
```

```
distribution_variance = tensorflow.keras.layers.Dense(2, name='log_variance')(encoder)
```

Finally, we will define a function that returns a sampled latent vector from the learned latent distribution. Check out the following python code:

```
def sample_latent_features(distribution):
    distribution_mean, distribution_variance = distribution
    batch_size = tensorflow.shape(distribution_variance)[0]
    random = tensorflow.keras.backend.random_normal(shape=
(batch_size, tensorflow.shape(distribution_variance)[1]))
    return distribution_mean + tensorflow.exp(0.5 * distribution_variance) * r
andom

sampled_latent_encoding = tensorflow.keras.layers.Lambda(sample_laten
t_features)([distribution_mean, distribution_variance])
```

Now we have the overall flow of our encoder network, that takes an image as input and returns a latent vector. We can now define the TensorFlow model object and print the model summary. See the following python code:

```
encoder_model = tensorflow.keras.Model(input_data, sampled_latent_enc
oding)
```

```
encoder_model.summary()
```

Following is the summary of encoder network:

Out[3]: Model: "model"

Layer (type) Output Shape Param # Connected to

```
=====
=====
```

```
input_1 (InputLayer) [(None, 28, 28, 1)] 0
```

```
conv2d (Conv2D) (None, 24, 24, 64) 1664 input_1[0][0]
```

max_pooling2d (MaxPooling2D) (None, 12, 12, 64) 0 conv2d[0][0]

conv2d_1 (Conv2D) (None, 10, 10, 64) 36928 max_pooling2d[0][0]

max_pooling2d_1 (MaxPooling2D) (None, 5, 5, 64) 0 conv2d_1[0][0]

conv2d_2 (Conv2D) (None, 3, 3, 32) 18464 max_pooling2d_1[0][0]

max_pooling2d_2 (MaxPooling2D) (None, 1, 1, 32) 0 conv2d_2[0][0]

flatten (Flatten) (None, 32) 0 max_pooling2d_2[0][0]

dense (Dense) (None, 16) 528 flatten[0][0]

mean (Dense) (None, 2) 34 dense[0][0]

log_variance (Dense) (None, 2) 34 dense[0][0]

```
lambda (Lambda) (None, 2) 0 mean[0][0]
```

```
log_variance[0][0]
```

```
Total params: 57,652
```

```
Trainable params: 57,652
```

```
Non-trainable params: 0
```

Our encoder is now ready. Let's define the decoder model now.

Step 4: Define Decoder

The decoder part of our VAE is should accept a latent vector as input and generate a handwritten digit image of size ‘28 x 28 x 1’ corresponding to that latent vector. As our latent vector is linear, we can inverse the convolutional operations to convert it into an image output. The following python code takes the latent vector of size 2, as input and generates an image of size 28 x 28 x 1 as output, by passing it thorough multiple inverse convolutions, activations and up-sampling layers.

```
decoder_input = tensorflow.keras.layers.Input(shape=(2))
decoder = tensorflow.keras.layers.Dense(64)(decoder_input)
decoder = tensorflow.keras.layers.Reshape((1, 1, 64))(decoder)
decoder = tensorflow.keras.layers.Conv2DTranspose(64, (3,3), activation
='relu')(decoder)
decoder = tensorflow.keras.layers.Conv2DTranspose(64, (3,3), activation
='relu')(decoder)
decoder = tensorflow.keras.layers.UpSampling2D((2,2))(decoder)
```

```
decoder = tensorflow.keras.layers.Conv2DTranspose(64, (3,3), activation='relu')(decoder)
decoder = tensorflow.keras.layers.UpSampling2D((2,2))(decoder)
decoder_output = tensorflow.keras.layers.Conv2DTranspose(1, (5,5), activation='relu')(decoder)
decoder_model = tensorflow.keras.Model(decoder_input, decoder_output)
decoder_model.summary()
```

Below is a summary of the Decoder network:

Out[4]: Model: "model_1"

Layer (type) Output Shape Param #

=====

input_2 (InputLayer) [(None, 2)] 0

dense_1 (Dense) (None, 64) 192

reshape (Reshape) (None, 1, 1, 64) 0

conv2d_transpose (Conv2DTran (None, 3, 3, 64) 36928

conv2d_transpose_1 (Conv2DTr (None, 5, 5, 64) 36928

up_sampling2d (UpSampling2D) (None, 10, 10, 64) 0

conv2d_transpose_2 (Conv2DTr (None, 12, 12, 64) 36928

up_sampling2d_1 (UpSampling2 (None, 24, 24, 64) 0

conv2d_transpose_3 (Conv2DTr (None, 28, 28, 1) 1601

=====

Total params: 112,577

Trainable params: 112,577

Non-trainable params: 0

Let's now define the final VAE network.

Step 5: Defining VAE

Now that our encoder and decoder networks are ready, let's combine them to obtain the final VAE architecture. It's important to combine them into a single network because we want to train this end-to-end system using backpropagation. To combine them, we can simply pass the output layer of encoder network as input to the decoder network. See the following code:

```
sampled_latent_encoding = encoder_model(input_data)
decoded = decoder_model(sampled_latent_encoding)
autoencoder = tensorflow.keras.models.Model(input_data, decoded)
```

Our VAE network is now ready. Next, let's define the loss function for training VAE.

Step 6: Custom Loss function

Basically, Our VAE model has two main goals – learning the latent distribution and generating handwritten digit images. Our loss function should also consider both of these scenarios such that our VAE is able to meet its goals during training. We will use ‘KL-loss’ for learning distribution and ‘pixel-wise reconstruction loss’ for comparing the generated images with the real ones. We can combine these two loss values to get the total loss of network and utilize that for training the VAE network end-to-end.

The following python code defines the custom loss function for our VAE:

```
def get_loss(distribution_mean, distribution_variance):
    def get_reconstruction_loss(y_true, y_pred):
        reconstruction_loss = tensorflow.keras.losses.mse(y_true, y_pred)
        reconstruction_loss_batch = tensorflow.reduce_mean(reconstruction_loss)
    )
    return reconstruction_loss_batch*28*28

def get_kl_loss(distribution_mean, distribution_variance):
    kl_loss = 1 + distribution_variance - tensorflow.square(distribution_mean)
    kl_loss -= tensorflow.exp(distribution_variance)
    kl_loss_batch = tensorflow.reduce_mean(kl_loss)
```

```

return kl_loss_batch*(-0.5)
def total_loss(y_true, y_pred):
    reconstruction_loss_batch = get_reconstruction_loss(y_true, y_pred)
    kl_loss_batch = get_kl_loss(distribution_mean, distribution_variance)
    return reconstruction_loss_batch + kl_loss_batch
return total_loss

```

As our custom loss function is now ready, we can go ahead and start training our network.

Step 6: Training VAE

Our final architecture and final objective is ready. We can now compile our model, to make it ready for training. We will utilize ‘Adam’ optimizer with default values for training VAE.

Compiling the VAE:

```

autoencoder.compile(loss=get_loss(distribution_mean, distribution_varia
nce), optimizer='adam')
autoencoder.summary()

```

Here is the final summary of our combined VAE network. It has roughly 170k trainable parameters:

Out[6]: Model: "model_2"

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 28, 28, 1)]	0
model (Functional)	(None, 2)	57652
model_1 (Functional)	(None, 28, 28, 1)	112577
Total params:		170,229

Trainable params: 170,229

Non-trainable params: 0

We can now go ahead and start training our VAE. We will train this network for 20 epochs with a batch size of 64.

```
autoencoder.fit(train_data, train_data, epochs=20, batch_size=64, validation_data=(test_data, test_data))
```

Below are the training logs, we can see that validation loss decreases as the training is progressing. This means that network is actually learning.

Out[6]: Train on 60000 samples, validate on 10000 samples

Epoch 1/20

```
60000/60000 [=====] - 19s 322us/sample - loss: 43.0782 - val_loss: 38.6226
```

Epoch 2/20

```
60000/60000 [=====] - 7s 109us/sample - loss: 38.2413 - val_loss: 37.3725
```

Epoch 3/20

```
60000/60000 [=====] - 7s 110us/sample - loss: 37.0779 - val_loss: 36.7547
```

Epoch 4/20

```
60000/60000 [=====] - 7s 109us/sample - loss: 36.4296 - val_loss: 36.1563
```

Epoch 5/20

```
60000/60000 [=====] - 7s 111us/sample - loss: 35.9672 - val_loss: 35.5301
```

.....

.....
.....

Epoch 20/20

60000/60000 [=====] - 7s 110us/sample - loss: 33.5621 - val_loss:
34.0148

Now that the training of VAE network is complete. We can start checking results now.

Step 7: Reconstruction Results

As a first step, we will check out the image reconstruction results of our VAE. Let's pass some test set images to the VAE model and see how well it is able to reconstruct the digits. We are not eyeing for an exactly similar reconstruction, as it is not an ordinary autoencoder. Instead, we would like to see the variations our VAE can introduce while keeping the original class of input data same.

The following python code chooses around 25 test images and plots them.

```
i = 10
print ("Real Test Images")
# Real Images
for k in range(5):
    plt.figure(figsize=(5,5))
    for j in range(5):
        plt.subplot(550 + 1 + j)
        plt.imshow(test_data[k+j+i,:,:,-1], cmap='gray_r')
        plt.axis('off')
    plt.show()
```

Below are the chosen 25 real test images:

Real Test Images

0	6	9	0	1
6	9	0	1	5
9	0	1	5	9
0	1	5	9	7
1	5	9	7	3

Figure 1.6: Real test images from Handwritten Digit Images dataset.

Next few lines of code, will use these 25 test images as input and generate VAE outputs. It will also plot the final reconstructed version of each of these images respectively.

```
# Reconstructed Images
print ("Reconstructed Images with Variational Autoencoder")
for k in range(5):
    plt.figure(figsize=(5,5))
    for j in range(5):
        plt.subplot(550 + 1 + j)
        output = autoencoder.predict(np.array([test_data[k+j+i]]))
        op_image = np.reshape(output[0]*255, (28, 28))
        plt.imshow(op_image, cmap='gray_r')
        plt.axis('off')
    plt.show()
```

Image reconstruction output from VAE:

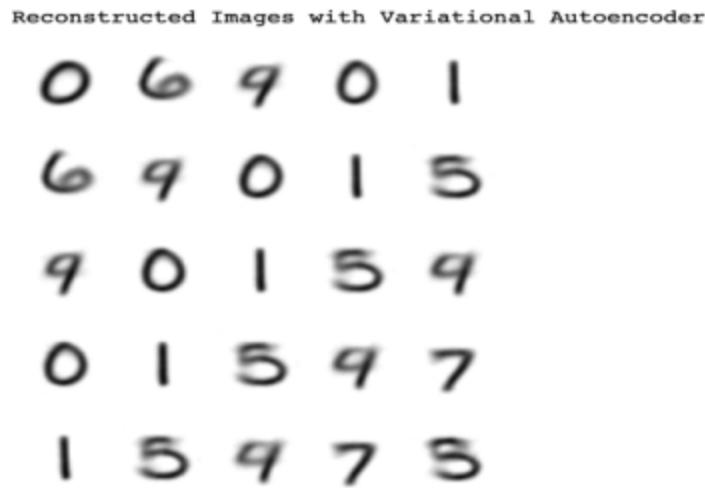


Figure 1.7: Reconstructed Handwritten Digit images by Variation Autoencoder (VAE)

Figure 1.7 shows the reconstructed version of test images from Figure 1.6. We can see that the VAE is able to introduce some variations in the images while keeping the original class of the samples intact. Another important thing to notice is that the output images are a little blurry which is expected (As discussed earlier about VAE).

Step 8: Latent Distribution Graph

Another goal of our VAE was to learn the latent distribution. As we assumed (and optimized for) the latent distribution to be gaussian, we can verify the fact by drawing a graph of the latent points.

The following python code, generates the latent vectors for around 10k test images and generates a point plot. It also color codes the latent points with their digit class.

```

x = []
y = []
k = []
for i in range(10000):
    k.append(testy[i])
    op = encoder_model.predict(np.array([test_data[i]]))
    x.append(op[0][0])

```

```

y.append(op[0][1])
df = pd.DataFrame()
df['x'] = x
df['y'] = y
df['k'] = ["digit-"+str(i) for i in k]
plt.figure(figsize=(8, 6))
sns.scatterplot(x='x', y='y', hue='k', data=df)
plt.show()

```

The following plot shows the distribution of latent vectors generated using the encoder network of our VAE model.

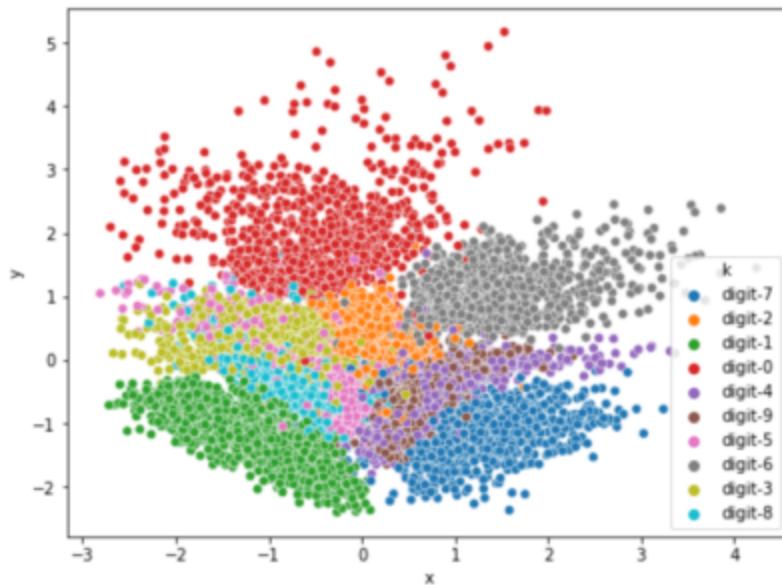


Figure 1.8: Latent distribution of Handwritten Digit Images generated using VAE Encoder network.

Figure 1.8 shows that the latent distribution is gaussian as it is centered at zero. Another important thing to notice is that each digit class has its own subset of space (see the colored areas), which again proves that the latent variables have learned unsupervised features about the classes. We can take advantage of this property of VAEs to generate data for the desired class.

Step 9: Data Generation

We have already verified that our model has successfully learned a latent distribution that is centered at zero and our decoder network can make use of this distribution for generating data. It means that we do not need input images anymore. The decoder part of our VAE can generate infinite amount of data given proper latent vectors as input. Let's try to generate a lot of digits using only the latent space and the decoder network.

The following python snippet will generate a large number of latent vectors. It will then pass those latent vectors as inputs to the decoder network of our trained VAE. The decoder is expected to generate a handwritten digit image for each of these vectors.

```
generator_model = decoder_model
x_values = np.linspace(-2, 2, 30)
y_values = np.linspace(-2, 2, 30)
figure = np.zeros((28 * 30, 28 * 30))
for ix, x in enumerate(x_values):
    for iy, y in enumerate(y_values):
        latent_point = np.array([[x, y]])
        generated_image = generator_model.predict(latent_point)[0]
        figure[ix*28:(ix+1)*28, iy*28:(iy+1)*28,] = generated_image[:, :, -1]
plt.figure(figsize=(15, 15))
plt.imshow(figure, cmap='gray_r', extent=[2, -2, 2, -2])
plt.show()
```

Below are the generated handwritten digit images:

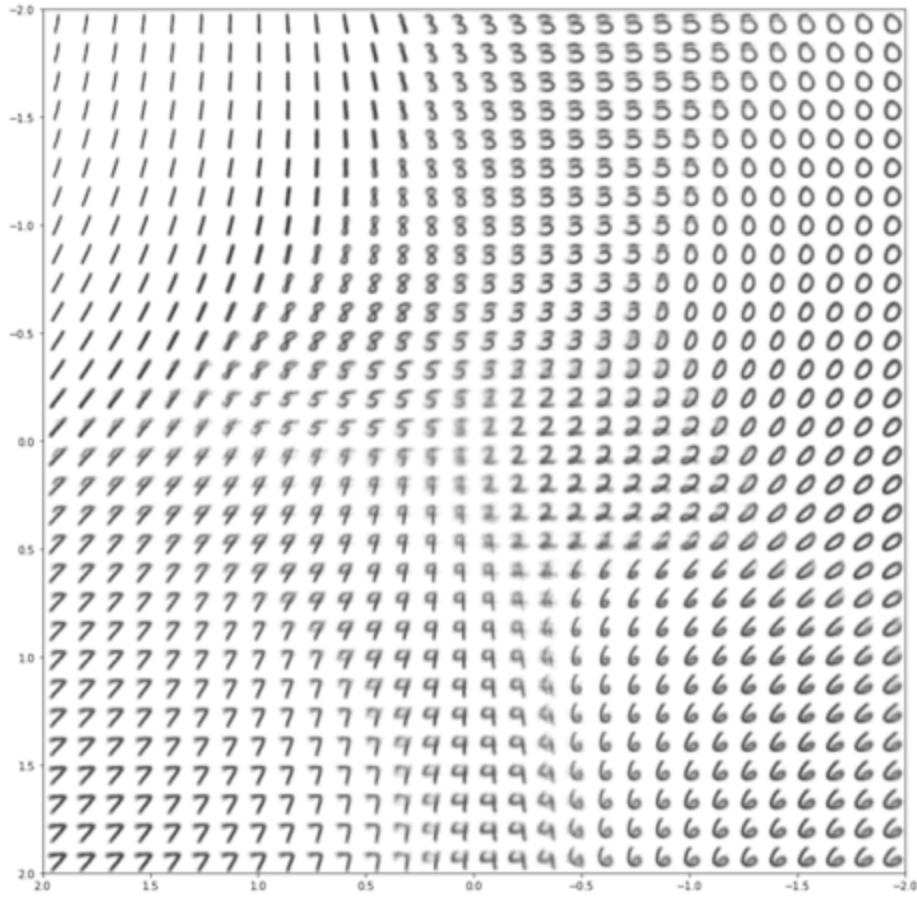


Figure 1.9: Handwritten Digit images generated by VAE decoder by just using latent vectors as input

Figure 1.9 shows that the selected subset of latent space is able to generate all the digits (0 to 9) with variations in styles as well as orientations. In this way, we can generate infinite volume of handwritten digits using just the decoder model. Hence, we have successfully learned a generative model for handwritten digits.

We have just implemented and trained a VAE network for handwritten digit images. Using the same concepts, we can scale it up for more complicated use-cases and datasets. Next, we will implement and train Pixel CNN.

PIXEL-CNN FOR HANDWRITTEN DIGITS

Objective

In this example, we will implement and train an autoregressive generative model called ‘Pixel CNN’ on MNIST handwritten digits dataset and verify its generative capabilities.

This experiment has the following steps:

- Importing Libraries
- Download and show data
- Data Preparation
- Define Masked CNN layers
- Define Pixel-CNN
- Training Pixel-CNN Model
- Results

Let’s get started.

Step 1: Importing Libraries

As a first step, we will import some useful python libraries in a Jupyter Notebook cell. See the following snippet:

```
import numpy as np  
import matplotlib.pyplot as plt  
import pandas as pd  
import seaborn as sns  
import warnings  
warnings.filterwarnings('ignore')  
%matplotlib inline  
import tensorflow  
print(tensorflow.__version__)
```

We will be using TensorFlow library with version 2.4.1 for our experiments.

Out[1]: 2.4.1

Now, let's download the dataset.

Step 2: Download and show data

In this step, we will download the MNIST Handwritten Digits dataset. This is a very common dataset and thus it is also available in keras datasets. The following python code downloads the data and plot's some handwritten digit images just to verify the dataset.

```
from tensorflow.keras.datasets import mnist  
(trainX, trainy), (testX, testy) = mnist.load_data()  
print('Training data shapes: X=%s, y=%s' % (trainX.shape, trainy.shape))  
  
print('Testing data shapes: X=%s, y=%s' % (testX.shape, testy.shape))
```

for k in range(9):

```
    plt.figure(figsize=(9,6))  
    for j in range(9):  
        i = np.random.randint(0, 10000)  
        plt.subplot(990 + 1 + j)  
        plt.imshow(trainX[i], cmap='gray_r')  
        plt.axis('off')  
        #plt.title(trainy[i])  
    plt.show()
```

Here is the output:



Figure 1.10: A glimpse of MNIST Handwritten Digits dataset.

Figure 1.10 gives a basic idea about the MNIST handwritten digits dataset. It has around 60k training images and 10k test images each with dimensions: 28 x 28.

Now, let's prepare data for the model.

Step 3: Data Preparation

In this step, we will prepare our dataset for the model training purpose. Firstly, we will convert these images into binary format (as expected by our basic Pixel CNN). Secondly, we will add a channel dimension to all the images, as expected by the Convolutional layers in our Pixel CNN.

```
trainX = np.where(trainX < (0.33 * 256), 0, 1)
train_data = trainX.astype(np.float32)
testX = np.where(testX < (0.33 * 256), 0, 1)
test_data = testX.astype(np.float32)
train_data = np.reshape(train_data, (60000, 28, 28, 1))
test_data = np.reshape(test_data, (10000, 28, 28, 1))
```

```
print (train_data.shape, test_data.shape)
```

Following is the final shape of our dataset after pre-processing:

```
Out[3]: (60000, 28, 28, 1) (10000, 28, 28, 1)
```

Our data is now ready. We can now define Pixel CNN.

Step 4: Define Masked CNN layers

In this step, we will define some building blocks of our Pixel CNN framework. First, we will define a *PixelConvLayer* class. A Pixel CNN layer is simply built on top of 2D Convolutional layers. Only difference is that it includes masking in order to accommodate the autoregressive training of the network. In simple words, autoregressive models generate output one pixel at a time, so, masking is a clever way to train Pixel CNN with regular CNN layers.

See the python code below for Pixel CNN layer implementation:

```
class PixelConvLayer(tensorflow.keras.layers.Layer):  
    def __init__(self, mask_type, **kwargs):  
        super(PixelConvLayer, self).__init__()  
        self.mask_type = mask_type  
        self.conv = tensorflow.keras.layers.Conv2D(**kwargs)  
    def build(self, input_shape):  
        # Build the conv2d layer to initialize kernel variables  
        self.conv.build(input_shape)  
        # Use the initialized kernel to create the mask  
        kernel_shape = self.conv.kernel.get_shape()  
        self.mask = np.zeros(shape=kernel_shape)  
        self.mask[: kernel_shape[0] // 2, ...] = 1.0  
        self.mask[kernel_shape[0] // 2, : kernel_shape[1] // 2, ...] = 1.0  
        if self.mask_type == "B":  
            self.mask[kernel_shape[0] // 2, kernel_shape[1] // 2, ...] = 1.0  
    def call(self, inputs):  
        self.conv.kernel.assign(self.conv.kernel * self.mask)
```

```
return self.conv(inputs)
```

Secondly, we will define a residual block layer. It is also a normal residual block, which is based on the Pixel CNN layer. See the following code:

```
class ResidualBlock(tensorflow.keras.layers.Layer):  
    def __init__(self, filters, **kwargs):  
        super(ResidualBlock, self).__init__(**kwargs)  
        self.conv1 = tensorflow.keras.layers.Conv2D(  
            filters=filters, kernel_size=1, activation="relu"  
        )  
        self.pixel_conv = PixelConvLayer(  
            mask_type="B",  
            filters=filters // 2,  
            kernel_size=3,  
            activation="relu",  
            padding="same",  
        )  
        self.conv2 = tensorflow.keras.layers.Conv2D(  
            filters=filters, kernel_size=1, activation="relu"  
        )  
    def call(self, inputs):  
        x = self.conv1(inputs)  
        x = self.pixel_conv(x)  
        x = self.conv2(x)  
        return tensorflow.keras.layers.add([inputs, x])
```

We now have the building blocks ready for Pixel CNN. We can now go ahead and define the final Pixel CNN architecture.

Step 5: Define Pixel CNN

In this step, we will define the Pixel CNN model architecture. This architecture is very similar to the one shown in original Pixel CNN paper. It combines Pixel CNN layers with residual blocks as shown in the following python code:

```
inputs = tensorflow.keras.Input(shape=(28,28,1))
x = PixelConvLayer(
    mask_type="A", filters=128, kernel_size=7, activation="relu", padding="same"
)(inputs)
for _ in range(5):
    x = ResidualBlock(filters=128)(x)
    for _ in range(2):
        x = PixelConvLayer(
            mask_type="B",
            filters=128,
            kernel_size=1,
            strides=1,
            activation="relu",
            padding="valid",
        )(x)
out = tensorflow.keras.layers.Conv2D(
    filters=1, kernel_size=1, strides=1, activation="sigmoid", padding="valid"
)
(x)
pixel_cnn = tensorflow.keras.Model(inputs, out)
pixel_cnn.summary()
```

Following is the final summary of our Pixel CNN model. It has roughly 500k trainable parameters.

Layer (type)	Output Shape	Param #
--------------	--------------	---------

input_1 (InputLayer)	[(None, 28, 28, 1)]	0
----------------------	---------------------	---

pixel_conv_layer (PixelConvL)	(None, 28, 28, 128)	6400
-------------------------------	---------------------	------

residual_block (ResidualBloc)	(None, 28, 28, 128)	98624
-------------------------------	---------------------	-------

residual_block_1 (ResidualBl)	(None, 28, 28, 128)	98624
-------------------------------	---------------------	-------

residual_block_2 (ResidualBl)	(None, 28, 28, 128)	98624
-------------------------------	---------------------	-------

residual_block_3 (ResidualBl)	(None, 28, 28, 128)	98624
-------------------------------	---------------------	-------

residual_block_4 (ResidualBl)	(None, 28, 28, 128)	98624
-------------------------------	---------------------	-------

pixel_conv_layer_6 (PixelCon)	(None, 28, 28, 128)	16512
-------------------------------	---------------------	-------

pixel_conv_layer_7 (PixelCon)	(None, 28, 28, 128)	16512
-------------------------------	---------------------	-------

conv2d_18 (Conv2D)	(None, 28, 28, 1)	129
--------------------	-------------------	-----

Total params: 532,673

Trainable params: 532,673

Non-trainable params: 0

Our Pixel CNN architecture is now ready. We will utilize *Adam* optimizer with a learning rate of 0.0005 for updating the weights of our model during training. Also, we will utilize ‘*binary_crossentropy*’ as a loss function for our model. Remember, Pixel CNN generates image one pixel at a time, and our

setup considers binary images with values 0 and 1. Thus, ‘*binary_crossentropy*’ loss makes sense here.

Compiling Pixel CNN:

```
adam = tensorflow.keras.optimizers.Adam(learning_rate=0.0005)
pixel_cnn.compile(optimizer=adam, loss="binary_crossentropy")
```

Our Pixel CNN is now ready for training.

Step 6: Training Pixel-CNN Model

In this step, we will train the Pixel CNN model on MNIST Handwritten Digits dataset. We will train our model for 50 epochs with a batch size of 128.

```
pixel_cnn.fit(
    x=train_data, y=train_data, batch_size=128, epochs=50, validation_data=
(test_data, test_data), verbose=1)
```

Following are the training logs. We can see that the validation loss is decreasing consistently, which proves that model is learning:

Out[3]:

Epoch 1/50

```
469/469 [=====] - 57s 116ms/step - loss: 0.1748 - val_loss: 0.0923
```

Epoch 2/50

```
469/469 [=====] - 53s 113ms/step - loss: 0.0918 - val_loss: 0.0892
```

Epoch 3/50

```
469/469 [=====] - 53s 113ms/step - loss: 0.0894 - val_loss: 0.0882
```

Epoch 4/50

```
469/469 [=====] - 53s 113ms/step - loss: 0.0881 - val_loss: 0.0873
```

.....

.....

. . . .
. . . .

Epoch 45/50

469/469 [=====] - 53s 113ms/step - loss: 0.0815 - val_loss: 0.0829

Epoch 46/50

469/469 [=====] - 53s 113ms/step - loss: 0.0815 - val_loss: 0.0827

Epoch 47/50

469/469 [=====] - 53s 113ms/step - loss: 0.0813 - val_loss: 0.0829

Epoch 48/50

469/469 [=====] - 53s 113ms/step - loss: 0.0812 - val_loss: 0.0829

Epoch 49/50

469/469 [=====] - 53s 113ms/step - loss: 0.0813 - val_loss: 0.0830

Epoch 50/50

469/469 [=====] - 53s 113ms/step - loss: 0.0812 - val_loss: 0.0828

As our model training is complete now, we can now check the results.

Step 7: Results

The following python code utilizes our Pixel CNN model for generating handwritten digit images. It also plots some of the generated results.

```
from IPython.display import Image, display
from tqdm import tqdm_notebook
# Create an empty array of pixels.
batch = 81
pixels = np.zeros(shape=(batch,) + (pixel_cnn.input_shape)[1:])
batch, rows, cols, channels = pixels.shape
```

```

# Iterate over the pixels because generation has to be done sequentially pi
xel by pixel.

for row in tqdm_notebook(range(rows)):
    for col in range(cols):
        for channel in range(channels):
            # Feed the whole array and retrieving the pixel value probabilities for the
            next

            # pixel.

            probs = pixel_cnn.predict(pixels)[:, row, col, channel]

            # Use the probabilities to pick pixel values and append the values to the i
            mage

            # frame.

            pixels[:, row, col, channel] = tensorflow.math.ceil(
                probs - tensorflow.random.uniform(probs.shape)
            )

            counter = 0

            for i in range(9):
                plt.figure(figsize=(9,6))
                for j in range(9):
                    plt.subplot(990 + 1 + j)
                    plt.imshow(pixels[counter,:,:,:], cmap='gray_r')
                    counter += 1
                plt.axis('off')
            plt.show()

```

Figure 1.11 shows the output of Pixel CNN model. We can see that the output images look very close to the handwritten digits. They are not perfect though. To get better results, we should move to a better way of training generative models such as GANs.



Figure 1.11: Handwritten Digit Images generated using the Pixel CNN model

We have now successfully implemented and trained a Pixel CNN based autoregressive generative model. Python code for all these experiments can also be found in the Github repository of this book.

1. Summary

This skill was focused on giving the readers a good understanding of generative learning frameworks. We started with an intuitive definition of the generative approach, and also learned about its key differences from the discriminative approach of developing ML models. We understood: what does it mean to learn a generative model, what are the expectations from a trained generative model and finally, what are the different ways of learning the generative models. Towards the end, we talked about some of the quality measures of a good generative model for the task of image generation. Towards the end, we also implemented and trained a couple of generative models: VAE and Pixel CNN.

Following are the key takeaways from this skill:

- Comprehensive understanding of the generative learning approach and its differences from the discriminative approach
- The learning objectives and expectations from a generative learning framework.
- Basic understanding of popular deep generative learning frameworks such as Autoregressive Models, VAEs, GANs and so on.
- Different methods of comparing and learning distributions.
- Pros and Cons of Likelihood-based vs likelihood-free learning approaches of developing the generative models.
- Hands on examples for developing a VAE and Pixel CNN.

With this, Skill 1 ends here. In the next few skills, we will mostly learn about the most popular generative learning framework: GAN.

1. you can try next...

- Find new application areas and apply Autoregressive and Variational Autoencoder based generative models.
- Think about improving objective functions for Variational Autoencoders to make them even better.
- Make sure you can explain the difference between a generative model and a discriminative model to a non-technical person.

End of Skill-1

SKILL 2

Generative Adversarial Networks

Artificial Neural Networks (ANNs) are often compared with human brain and the reason behind that is that the information flow works pretty much same in both cases. Recent progress in the field of Artificial Intelligence (AI) through deep learning (ANNs) has been phenomenal, to make machines capable of performing complex tasks as efficient as a real human and sometimes even better. Today machines can translate any given language into desired languages, machines can also read for us, write for us, and even talk to us to some extent.

Today, we have the capability to develop AI systems to perform almost any task we want (given sufficient amount of training examples). Although, these AI systems are really good at solving problems that they are designed to solve, but still, they can't make decisions beyond a certain point (especially when presented with unseen observations). Humans on the other hand, understand the world and the surroundings in a different manner. They understand the physics of this world and can make correct decisions even in the unknown surroundings or situations. They pretty well understand that jumping from a high building is not a good idea, without even jumping once. An AI system may not understand this until it jumps at least once just to find out that it really was not a fruitful decision. Collecting examples of all such possible scenarios is close to impossible. It means that we need to think beyond the current learning algorithms, if we want a machine to understand this world like humans do.

Human brain is the most mysterious computer, the world has ever seen. Our brain learns the patterns of the world though multiple different ways: it understands how the world works, it understands the 3-Dimensional world way better than the ‘edges and corners’- like a neural network learns. When it comes to understand the world for an AI system (and not just classify the objects), the generative learning approaches are a big step towards it. Because they are capable of learning complex representations of a given dataset and also able to generate realistic data. Though there are multiple ways of training

generative models as discussed in last skill, they all have their own strengths and weaknesses.

Generative Adversarial Networks, or GANs for short, are known for producing most promising results currently (i.e., generating high quality and realistic data samples). In this skill, we will learn about how GANs can be trained to generate infinite amount of realistic data.

This skill covers the following topics:

- What are Generative Adversarial Networks?
- What is GAN objective?
- How to train a GAN based model?
- What are the pros and cons of GAN?
- Experiment: GAN for Handwritten Digits Generation

Let's get started.

1. Generative Adversarial Networks

Generative Adversarial Network, or GAN for short, is a framework for learning generative models. GANs were invented by Ian Goodfellow in 2014 and first described in his research paper titled '*Generative Adversarial Networks*'. They are made up of two distinct networks: a *generator* and a *discriminator*.

For a given dataset D, the generator network is responsible for learning the data distribution so that later it can generate data that looks very similar to the real data samples from D. The job of the discriminator network is to look at a sample and decide whether it came from the dataset or 'the generator network'. During the training process, the generator network keeps getting better at generating realistic samples and tries its best to fool the discriminator, whereas the discriminator network itself tries to become a better detective and aims at correctly recognizing the fake samples produced by the 'the generator'. The training is complete when the samples coming from the generator are so good that the discriminator is unable to take a

decision whether the sample is from the dataset or ‘the generator’ (considering that the discriminator is not very dumb).

The original GAN paper introduces Mini-Max loss for training the GAN framework, however, quite a few different loss functions have been successfully applied to GANs. We will learn about these new loss functions in upcoming skills. In this skill, we will mostly focus on the original GAN objective presented by the paper. The Mini-Max loss function can be written as follows:

$$E_x[\log D(x)] + E_z[\log(1 - D(G(z)))]$$

In this loss function:

- $D(x)$ is the discriminator’s probability estimate indicating that a given data sample is real. So higher value indicates realistic samples.
- E_x represents the expected value over all the data samples.
- $G(z)$ is a generated sample by the generator, given some noise input vector z (we will learn about z later).
- $D(G(z))$ is the discriminator’s probability estimate that the given generated sample is real (which is actually fake).
- E_z represents the expected value over all noisy inputs to the generator network.

Let’s now learn about the GAN objective in details.

1. GAN Objective

Assume, D is a dataset of n samples (x_1, x_2, \dots, x_n) and G_θ is a generator network defined over the parameter space Θ , such that it deterministically generates samples x from z . Whereas, D_ϕ is a discriminator network defined over the parameter space ϕ , and its main job is to distinguish the samples coming from the real dataset and the generator network. Here, z is a noise vector sampled from any arbitrary distribution $p(z)$.

During the training process, the generator and the discriminator networks play a two-player mini-max game, where the generator tries to fool the discriminator by generating samples that look indistinguishable from the real ones (by minimizing two-sample test objective) and the discriminator tries to become a better detective by maximizing the same objective.

As described in the Ian Goodfellow's paper, the GAN objective can be written formally as:

$$\min_{\theta} \max_{\phi} V(G_{\theta}, D_{\phi}) = E_{x \sim p_{\text{data}}} [\log D_{\phi}(x)] + E_{z \sim p_z} [\log(1 - D_{\phi}(G_{\theta}(z)))]$$

The training criterion further can be re-written as:

$$V(G, D) = \int_x p_{\text{data}}(x) \log(D(x)) dx + \int_z p_z(z) \log(1 - D(G(z))) dz$$

An optimal generator is when the model distribution $p_G = p_{\text{data}}$ (such that the distribution learned by the generator is similar to the real distribution of the dataset), and the generated samples $G(z) = x$.

The function now can be re-written as:

$$V(G, D) = \int_x p_{\text{data}}(x) \log(D(x)) dx + p_G(x) \log(1 - D(x)) dx$$

As the discriminator's job is to maximize the above function, the maximum of a function of type $a * \log(y) + b * \log(1 - y)$ is achieved at $y = \frac{a}{a+b}$ for given range $[0, 1]$ of y .

Thus, for a fixed generator G , the optimal discriminator D is possible when:

$$D_G^*(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_G(x)}$$

Now, the optimal cost value:

$$= E_{x \sim p_{\text{data}}} [\log D_G^*(x)] + E_{x \sim p_G} [\log(1 - D_G^*(x))]$$

$$\begin{aligned}
&= E_{x \sim p_{\text{data}}} \left[\log \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_G(x)} \right] + E_{x \sim p_G} \left[\log \frac{p_G(x)}{p_{\text{data}}(x) + p_G(x)} \right] \\
&= E_{x \sim p_{\text{data}}} \left[\log \frac{p_{\text{data}}(x)}{\frac{(p_{\text{data}}(x) + p_G(x))}{2}} \right] - E_{x \sim p_{\text{data}}} [\log 2] + E_{x \sim p_G} \\
&\quad \left[\log \frac{p_G(x)}{\frac{(p_{\text{data}}(x) + p_G(x))}{2}} \right] - E_{x \sim p_G} [\log 2]
\end{aligned}$$

Here, $-E_{x \sim p_{\text{data}}} [\log 2] - E_{x \sim p_G} [\log 2] = -\log(4)$. So now the function would be:

$$= -\log(4) + D_{\text{KL}} \left(p_{\text{data}} \parallel \frac{p_{\text{data}} + p_G}{2} \right) + D_{\text{KL}} \left(p_G \parallel \frac{p_G + p_{\text{data}}}{2} \right)$$

where D_{KL} represents the Kullback-Leibler (KL) divergence between two probability distributions (a background about KL divergence is present in skill 1).

$$= -\log(4) + 2 * JSD(p_{\text{data}} \parallel p_G)$$

Here, JSD represents the Jensen-Shannon divergence between two distributions, which is always non-negative and it's zero when the two distributions are exactly same. The only optimal solution is $p_{\text{data}} = p_G$, it means that the generator network perfectly replicating the data generating process and the global minimum of above cost function is $-\log(4)$.

1. Training a GAN

In the previous section we saw that the mini-max loss function, introduced by Ian Goodfellow, is meaningful as it is minimizing the distance (Jenson Shannon divergence) between the model distribution and the true data distribution. As discussed before, the mini-max function can be written as:

$$\text{mini - max Obj} = E_x[\log D(x)] + E_z[\log(1 - D(G(z)))]$$

Writing a training iteration for a GAN is simple. First, we update the weights of the discriminator network on a mini batch of data (real plus noisy samples). Then we update the generator by taking feedback from the outputs of discriminator network for only the noisy samples. So, the gradient for the generator network is passed by the discriminator network. Figure 2.1 shows an implementation of training iteration loop as suggested by the original GAN paper.

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k , is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

```

for number of training iterations do
  for  $k$  steps do
    • Sample minibatch of  $m$  noise samples  $\{z^{(1)}, \dots, z^{(m)}\}$  from noise prior  $p_g(z)$ .
    • Sample minibatch of  $m$  examples  $\{x^{(1)}, \dots, x^{(m)}\}$  from data generating distribution  $p_{\text{data}}(x)$ .
    • Update the discriminator by ascending its stochastic gradient:
      
$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[ \log D(x^{(i)}) + \log (1 - D(G(z^{(i)}))) \right].$$

  end for
  • Sample minibatch of  $m$  noise samples  $\{z^{(1)}, \dots, z^{(m)}\}$  from noise prior  $p_g(z)$ .
  • Update the generator by descending its stochastic gradient:
    
$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(z^{(i)}))).$$

end for
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

```

Figure 2.1: GAN training algorithm as per Ian Goodfellow's paper

Based on the requirements, we can choose to update the discriminator network more than the generator network, by increasing the value of k as illustrated in Figure 1.2. Because, the learning of the generator network is highly dependent upon the quality of gradients provided by the discriminator network. Later in this skill, we will implement a GAN and its training algorithm. Before that lets discuss some of the common pros of cons of GAN based generative models.

1. Pros and Cons of GANs

GAN based generative models have some known pros and cons over other known generative modeling techniques. The following is a detailed overview of the common advantages and dis-advantages of GAN framework based generative models:

4.1 Advantages of GANs

- Unlike the likelihood based generative models, GANs do not directly optimize on the training samples. The generator network weights are updated indirectly through the discriminator network and it makes them better at learning the distribution. Thus, they are capable of generating wide variety of content which may or may not be exactly similar to the training samples.
- GANs are computationally efficient and has the ability to generate high dimensional data real fast.
- The images generated by GANs are good quality and look very realistic.

4.2 Dis-advantages of GANs

- Training failure is quite common in case of GANs, as one needs to synchronize the generator and discriminator training well.
- GANs are good at generating continuous data like image-pixels, audio...etc. It's hard to generate discrete data with GANs e.g., text.

Now that we have a good background of GANs along with their advantages and dis-advantages, we can now jump directly into the experimentation part. In this skill, we will implement a simple GAN capable of generating handwritten digit images.

1. Experiments

In this skill, we will implement and train a simple GAN model to generate handwritten digit images.

The code samples shown in this skill can be downloaded from the following Github location:

<https://github.com/kartikgill/The-GAN-Book/tree/main/Skill-02>

Let's get started.

GAN FOR HANDWRITTEN DIGIT GENERATION

Objective

In this experiment, we will implement a GAN model from scratch and train it on MNIST Handwritten digits dataset. After learning the distribution, it should be capable of generating unlimited samples of realistic handwritten digit images. Entire python code of this experiment is also present in the Github repository of this book.

This experiment has the following steps:

- Importing Useful Libraries
- Download and Display data
- Dataset normalization
- Defining Generator
- Defining Discriminator
- Defining Combined model
- Utility functions
- Training GAN
- Results
- Loss Chart

Let's get started.

Step 1: Importing Useful Libraries

As a first step, we will open a new Jupyter Notebook with python kernel and import some useful libraries as shown in the snippet below.

```
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
from tqdm import tqdm_notebook  
%matplotlib inline  
import tensorflow
```

```
print(tensorflow.__version__)
```

Now, we let's download and verify the data.

Step 2: Download and Display data

In this step, we will download the MNIST Handwritten digits dataset and display some random images to verify if everything is good. Check out the following python snippet:

```
from tensorflow.keras.datasets import fashion_mnist, mnist  
(trainX, trainY), (testX, testY) = mnist.load_data()  
print('Training data shapes: X=%s, y=%s' % (trainX.shape, trainY.shape))  
  
print('Testing data shapes: X=%s, y=%s' % (testX.shape, testY.shape))
```

```
for k in range(9):
```

```
    plt.figure(figsize=(7, 7))
```

```
    for j in range(9):
```

```
        i = np.random.randint(0, 10000)
```

```
        plt.subplot(990 + 1 + j)
```

```
        plt.imshow(trainX[i], cmap='gray_r')
```

```
        plt.axis('off')
```

```
#plt.title(trainY[i])
```

```
plt.show()
```

Figure 2.2 shows some data samples. We can see that MNIST Handwritten digits dataset has roughly 60k training images and 10k test images, each with dimensions 28 x 28. The dataset also has labels for the class of digits from 0 to 9.

```

Training data shapes: X=(60000, 28, 28), y=(60000)
Testing data shapes: X=(10000, 28, 28), y=(10000,

5 7 1 7 2 4 / 0 8
4 5 0 1 4 0 6 0 7
5 0 9 8 2 0 4 4 1
9 6 4 9 2 6 3 6 2
2 1 2 3 0 4 8 9 1
2 9 9 6 0 6 5 1 2
0 2 9 4 0 2 7 7 1
4 0 9 1 1 0 7 6 0
9 6 0 9 4 0 3 9 1

```

Figure 12: MNIST Handwritten Digits dataset samples

We now have data with us. Let's prepare it for the model.

Step 3: Dataset normalization

In this step, we will first normalize the pixel values of each image from training and test sets. As pixel values range from 0 to 255 in a regular image, we will normalize them into a smaller range of [0, 1], as the normalized data helps in the convergence of neural networks. Secondly, we will also add an extra channel dimension to our images, as expected by convolutional layers.

See the following python code for data preparation:

```

trainX = [image/255.0 for image in trainX]
testX = [image/255.0 for image in testX]
trainX = np.reshape(trainX, (60000, 28, 28, 1))
testX = np.reshape(testX, (10000, 28, 28, 1))
print (trainX.shape, testX.shape, trainY.shape, testY.shape)
Out[3]: (60000, 28, 28, 1) (10000, 28, 28, 1) (60000,) (10000,)

```

Data preparation code also prints the final shape of prepared dataset. As shown in the output above. Our dataset is now ready for the model, let's go ahead and define the model architecture now.

Step 4: Defining Generator

In this step, we will define the generator network. The generator network of a GAN based model is expected to receive a noise vector as input and generate an image as output. In our setup, we will pass a noisy vector of size 50, as input to our generator network and then pass it through multiple repetitions of fully connected layers, activation function and batch normalization layers. In the final layer, we will reshape our output into three dimensions as the expected output is an image of size 28 x 28 x 1. Because we have restricted the pixel range from 0 to 1, we can make use of the *sigmoid* activation function in our final layer to generate an image with pixel range [0, 1].

The following python snippet implements the generator network:

```
random_input = tensorflow.keras.layers.Input(shape = 50)
x = tensorflow.keras.layers.Dense(1200, activation='relu')
(random_input)

x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Dense(1000, activation='relu')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Dense(28*28)(x)
x = tensorflow.keras.layers.Reshape((28, 28, 1))(x)
generated_image = tensorflow.keras.layers.Activation('sigmoid')(x)
generator_network = tensorflow.keras.models.Model(inputs=random_input,
outputs=generated_image)

generator_network.summary()
```

Following is the summary of our generator network. It has roughly 2 Million (or 2M), trainable parameters.

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[(None, 50)]	0
dense_3 (Dense)	(None, 1200)	61200
batch_normalization_2 (Batch Normalization)	(None, 1200)	4800
dense_4 (Dense)	(None, 1000)	1201000
batch_normalization_3 (Batch Normalization)	(None, 1000)	4000
dense_5 (Dense)	(None, 784)	784784
reshape_1 (Reshape)	(None, 28, 28, 1)	0
activation_1 (Activation)	(None, 28, 28, 1)	0

Total params: 2,055,784
Trainable params: 2,051,384
Non-trainable params: 4,400

The generator network is now ready. Let's define the discriminator network now.

Step 5: Defining Discriminator

The discriminator network architecture is very similar to an image classifier. As its main job is to distinguish the real samples from the fake

ones. It should take an image of size 28 x 28 x 1 as input and provide a probability value as output, indicating the confidence of the model to classify a sample as real. Our discriminator network is going to be very simple. We will first flatten the image input into a one-dimensional input and then pass it through the layers of fully connected, activation, and dropout. Final layer would have only a single neuron which will then be passed into a sigmoid activation layer to get the final probability output within range [0, 1].

Following python code implements the discriminator network:

```
image_input = tensorflow.keras.layers.Input(shape=(28, 28, 1))
x = tensorflow.keras.layers.Flatten()(image_input)
x = tensorflow.keras.layers.Dense(256, activation='relu')(x)
x = tensorflow.keras.layers.Dropout(0.3)(x)
x = tensorflow.keras.layers.Dense(128, activation='relu')(x)
x = tensorflow.keras.layers.Dropout(0.5)(x)
x = tensorflow.keras.layers.Dense(1)(x)
real_vs_fake_output = tensorflow.keras.layers.Activation('sigmoid')(x)
discriminator_network = tensorflow.keras.models.Model(inputs=image_i
nput, outputs=real_vs_fake_output)
discriminator_network.summary()
```

The following is the summary of the discriminator network. It roughly has 200k trainable parameters.

Layer (type) Output Shape Param #

=====

input_4 (InputLayer) [(None, 28, 28, 1)] 0

flatten_1 (Flatten) (None, 784) 0

dense_9 (Dense) (None, 256) 200960

```
dropout_2 (Dropout) (None, 256) 0
```

```
dense_10 (Dense) (None, 128) 32896
```

```
dropout_3 (Dropout) (None, 128) 0
```

```
dense_11 (Dense) (None, 1) 129
```

```
activation_3 (Activation) (None, 1) 0
```

```
=====
```

```
Total params: 233,985
```

```
Trainable params: 233,985
```

```
Non-trainable params: 0
```

As the discriminator network is very similar to a binary classifier, we can compile it using the '*binary_crossentropy*' loss. We will utilize the Adam optimizer with a learning rate of 0.00005 and beta_1 value of 0.5 for updating the weights of the discriminator network.

```
adam_optimizer = tensorflow.keras.optimizers.Adam(learning_rate=0.00005, beta_1=0.5)
```

```
discriminator_network.compile(loss='binary_crossentropy', optimizer=adam_optimizer, metrics=['accuracy'])
```

The discriminator network is now all set. We can now define the final GAN model.

Step 6: Defining Combined model

Our generator and discriminator networks are ready now. We can now combine them to define the final GAN model. As discussed earlier, the gradient for updating the generator model comes from a discriminator. It is important to fix the discriminator while updating the weights of the generator network. So, our final combined model will have a frozen discriminator network combined with the generator network. We can freeze the

discriminator network by just setting the flag ‘*trainable=False*’, in the model object. To combine the networks into a single network, we will pass the generated output of the generator network in the discriminator network. So, our final combined model will have noise vector as input and probability as output.

The following python code implements the combined GAN model:

```
discriminator_network.trainable=False  
g_output = generator_network(random_input)  
d_output = discriminator_network(g_output)  
gan_model = tensorflow.keras.models.Model(random_input, d_output)  
gan_model.summary()
```

Following is the summary of our final GAN model. It has around 2M trainable parameters. Note that the discriminator network is frozen so trainable parameters only include the parameters of the generator network.

Layer (type)	Output Shape	Param #
--------------	--------------	---------

input_2 (InputLayer)	[(None, 50)]	0
----------------------	--------------	---

model_1 (Functional)	(None, 28, 28, 1)	2055784
----------------------	-------------------	---------

model_3 (Functional)	(None, 1)	233985
----------------------	-----------	--------

Total params: 2,289,769

Trainable params: 2,051,384

Non-trainable params: 238,385

Finally, we can compile our final GAN model with ‘*binary_crossentropy*’ as loss function and Adam optimizer (similar to the discriminator network), for updating weights.

```
gan_model.compile(loss='binary_crossentropy', optimizer=adam_optimizer)
```

Our GAN architecture is now ready. Let's write some utility functions to create batches of training data.

Step 7: Utility functions

In this step, we will define some utility functions to get the batches of data for training. We will define separate functions for: generating random noise, getting fake samples from the discriminator and getting real samples from the dataset. We will also define a function, that shows the images generated by the generator network using random noise as input. This will help us in verifying the quality of generated results while the training progresses.

The following python code implements the utility functions:

```
indices = [i for i in range(len(trainX))]

def get_random_noise(batch_size, noise_size):
    random_values = np.random.randn(batch_size*noise_size)
    random_noise_batch = np.reshape(random_values, (batch_size, noise_size))
    return random_noise_batch

def get_fake_samples(generator_network, batch_size, noise_size):
    random_noise_batch = get_random_noise(batch_size, noise_size)
    fake_samples = generator_network.predict_on_batch(random_noise_batch)
    return fake_samples

def get_real_samples(batch_size):
    random_indices = np.random.choice(indices, size=batch_size)
    real_images = trainX[np.array(random_indices),:]
    return real_images

def show_generator_results(generator_network):
    for k in range(9):
```

```

plt.figure(figsize=(7, 7))
fake_samples = get_fake_samples(generator_network, 9, noise_size)
for j in range(9):
    plt.subplot(990 + 1 + j)
    plt.imshow(fake_samples[j,:,:,-1], cmap='gray_r')
    plt.axis('off')
# plt.title(trainY[i])
plt.show()
return

```

Our utility functions are now ready. We can now start training our GAN.

Step 8: Training GAN

In this step, we will define the training iteration loop for our GAN model. We plan to train our model for 200 epochs with a batch size of 100. Each epoch has 500 steps and the size of the noise vector is kept 50.

During each training step, we will first update the weights of the discriminator network on a batch on mixed (real and fake) samples. Subsequently, we will update the generator network based on the loss value of fake samples provided by a frozen discriminator network. The gradient for the generator network here flows through the frozen discriminator network first and then updates the weights of the generator network. Note that, we will flip the labels of fake samples from 0 to 1 while calculating the loss of end-to-end GAN network to make the discriminator believe that these are real samples and thus, update the generator weights accordingly.

The following python code implements the training loop for our GAN setup:

```

epochs = 200
batch_size = 100
steps = 500
noise_size = 50
losses_d = []

```

```

losses_g = []
for i in range(0, epochs):
    if (i%10 == 0):
        show_generator_results(generator_network)
    for j in range(steps):
        fake_samples = get_fake_samples(generator_network, batch_size//2, noise_size)
        real_samples = get_real_samples(batch_size=batch_size//2)
        fake_y = np.zeros((batch_size//2, 1))
        real_y = np.ones((batch_size//2, 1))
        input_batch = np.vstack((fake_samples, real_samples))
        output_labels = np.vstack((fake_y, real_y))
        # Updating Discriminator weights
        discriminator_network.trainable=True
        loss_d = discriminator_network.train_on_batch(input_batch, output_labels)
        gan_input = get_random_noise(batch_size, noise_size)
        # Make the Discriminator believe that these are real samples and calculate
        # loss to train the generator
        gan_output = np.ones((batch_size))
        # Updating Generator weights
        discriminator_network.trainable=False
        loss_g = gan_model.train_on_batch(gan_input, gan_output)
        losses_d.append(loss_d[0])
        losses_g.append(loss_g)
        if j%50 == 0:
            print ("Epoch:%.0f, Step:%.0f, D-Loss:%.3f, D-Acc:%.3f, G-
Loss:%.3f"%(i,j,loss_d[0],loss_d[1]*100,loss_g))

```

As the training progresses, both networks keep getting better at their job. After a certain number of epochs, we will find out that the loss of the generator model becomes very small. At that stage, we can stop the training and check the samples generated by the generator network. If the generated samples are good we can stop training.

The following training logs show the training progress:

Out[8]:Epoch:0, Step:0, D-Loss:0.798, D-Acc:54.000, G-Loss:0.519

Epoch:0, Step:50, D-Loss:0.127, D-Acc:100.000, G-Loss:3.190
Epoch:0, Step:100, D-Loss:0.039, D-Acc:100.000, G-Loss:5.013
Epoch:0, Step:150, D-Loss:0.034, D-Acc:99.000, G-Loss:6.396
Epoch:0, Step:200, D-Loss:0.017, D-Acc:100.000, G-Loss:7.363
Epoch:0, Step:250, D-Loss:0.022, D-Acc:100.000, G-Loss:9.171
Epoch:0, Step:300, D-Loss:0.010, D-Acc:100.000, G-Loss:10.002
Epoch:0, Step:350, D-Loss:0.026, D-Acc:99.000, G-Loss:12.115
Epoch:0, Step:400, D-Loss:0.058, D-Acc:98.000, G-Loss:13.406
Epoch:0, Step:450, D-Loss:0.098, D-Acc:97.000, G-Loss:13.396
Epoch:1, Step:0, D-Loss:0.094, D-Acc:98.000, G-Loss:11.440
Epoch:1, Step:50, D-Loss:0.128, D-Acc:95.000, G-Loss:11.614
Epoch:1, Step:100, D-Loss:0.163, D-Acc:94.000, G-Loss:11.277
Epoch:1, Step:150, D-Loss:0.091, D-Acc:97.000, G-Loss:10.763
Epoch:1, Step:200, D-Loss:0.102, D-Acc:98.000, G-Loss:11.872
Epoch:1, Step:250, D-Loss:0.089, D-Acc:97.000, G-Loss:11.978
Epoch:1, Step:300, D-Loss:0.139, D-Acc:93.000, G-Loss:12.862
Epoch:1, Step:350, D-Loss:0.172, D-Acc:92.000, G-Loss:12.465
Epoch:1, Step:400, D-Loss:0.141, D-Acc:94.000, G-Loss:12.817
Epoch:1, Step:450, D-Loss:0.106, D-Acc:95.000, G-Loss:14.795
Epoch:2, Step:0, D-Loss:0.150, D-Acc:94.000, G-Loss:12.162
Epoch:2, Step:50, D-Loss:0.176, D-Acc:94.000, G-Loss:12.748
Epoch:2, Step:100, D-Loss:0.223, D-Acc:91.000, G-Loss:13.636
Epoch:2, Step:150, D-Loss:0.237, D-Acc:90.000, G-Loss:14.058

Epoch:2, Step:200, D-Loss:0.195, D-Acc:92.000, G-Loss:14.113

...

...

...

Epoch:199, Step:50, D-Loss:0.614, D-Acc:67.000, G-Loss:0.867

Epoch:199, Step:100, D-Loss:0.552, D-Acc:71.000, G-Loss:0.914

Epoch:199, Step:150, D-Loss:0.618, D-Acc:65.000, G-Loss:0.866

Epoch:199, Step:200, D-Loss:0.638, D-Acc:62.000, G-Loss:0.868

Epoch:199, Step:250, D-Loss:0.597, D-Acc:65.000, G-Loss:1.043

Epoch:199, Step:300, D-Loss:0.601, D-Acc:70.000, G-Loss:0.856

Epoch:199, Step:350, D-Loss:0.620, D-Acc:62.000, G-Loss:0.918

Epoch:199, Step:400, D-Loss:0.674, D-Acc:58.000, G-Loss:0.949

Epoch:199, Step:450, D-Loss:0.596, D-Acc:73.000, G-Loss:0.953

Now that our GAN is successfully trained, we can check out the results.

Step 9: Results

During the training process of GANs, it is a good practice to keep saving the generator outputs and model weights, after every few epochs. It can later help us in understanding the optimal number of epochs that give the best results. During the training of our GAN model, we checked the generator outputs multiple times. We saw that the generated images were very noisy before starting the training (as expected). And, then after few epochs the outputs started taking a meaningful shape. Around epoch 50, even though the generated samples were not very clean, we could still identify digits. Finally, after 200 epochs of training, we saw that the generator model was capable of generating very realistic looking handwritten digit images. Figure 2.3 shows the generator outputs at different number of epochs during training.

It is quite fascinating to see that using GAN framework, we can convert a noisy vector into a meaningful and realistic image. Once the training is complete, we can discard the discriminator network and make use of the generator network for generating unlimited number of samples. All we need to do is: pass a noisy vector as input.

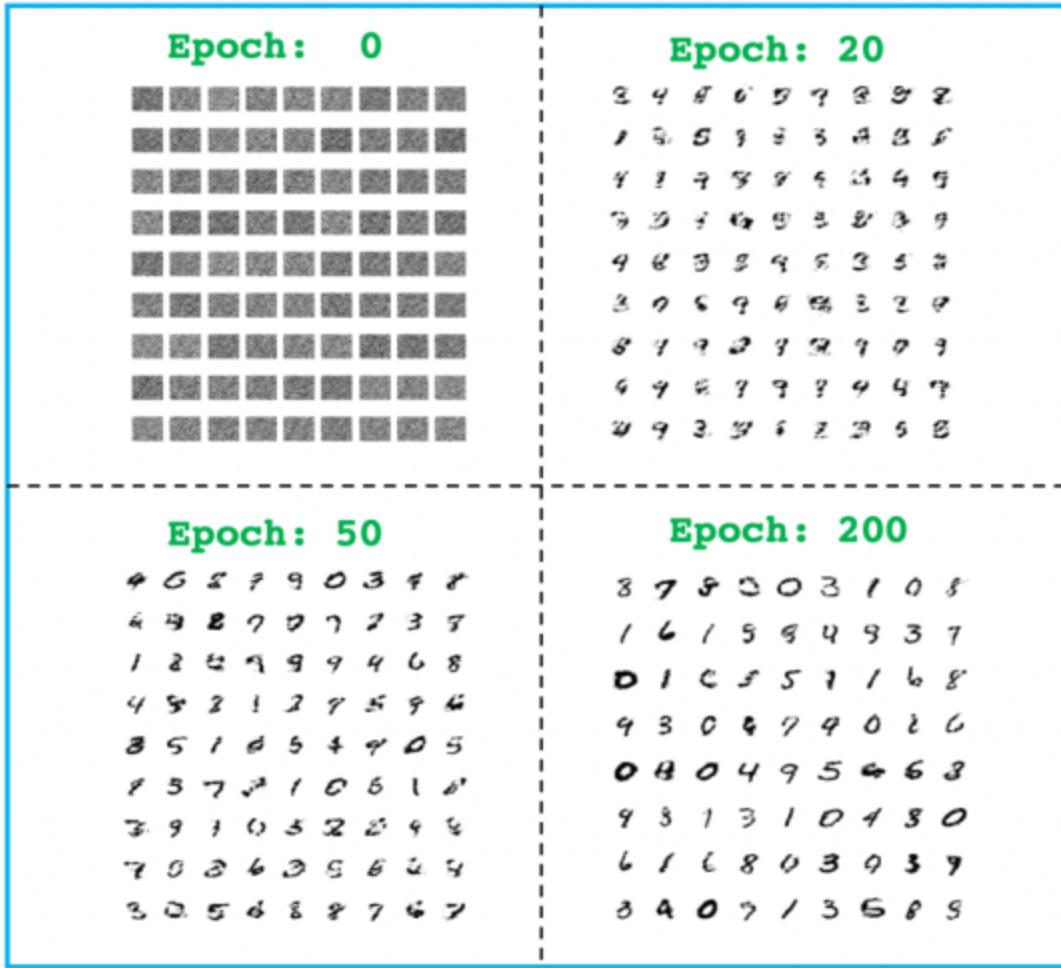


Figure 13: Handwritten Digit Images generated by GAN, at different epochs of training

Looking at the results present in Figure 2.3; we can say that our GAN training was successful. Let's now look at the loss charts of networks for training.

Step 10: Loss Chart

In this step, we will plot the loss values of both the generator model and the discriminator model with respect to the training steps. As we have been storing the loss values during training, we can simply plot them using matplotlib. See the python code below:

```
steps = [i for i in range(len(losses_d))]
plt.figure(figsize=(10, 6))
plt.plot(losses_d[:5000])
```

```

plt.plot(losses_g[:5000])
plt.xlabel('Steps')
plt.ylabel('Loss Value')
plt.title("GAN: Loss Trends")
plt.legend(['Discriminator Loss', 'Generator Loss'])
plt.show()

```

Figure 2.4 shows the loss charts. We can see that the generator loss is very high during the start of the training. It is expected, as the generator is quite dumb initially and only generates meaningless data. The discriminator on the other hand is a simple classification model, and thus it learns to distinguish between the real and fake samples pretty quickly. After some time, the generator loss starts decreasing which mean that the generator network is finally starting to generate some meaningful content. We can stop training when the generator loss is quite low and doesn't fluctuate much. Additionally, we can also keep checking the generated outputs to find the best point of stopping the training.

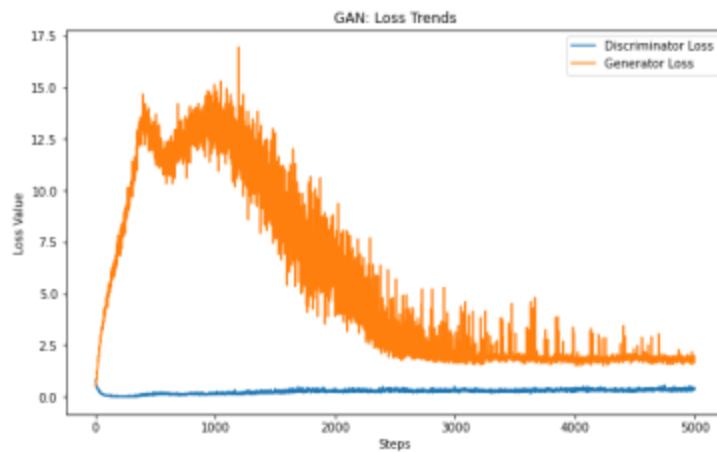


Figure 14: Plotting the loss of the generator and the discriminator network during training

We have now successfully understood, implemented, trained and tested a GAN model. With this, we are now ready to dive into more complicated GANs.

1. Summary

This skill was focused on the basics of generative adversarial nets or GANs, a generative learning framework introduced by Ian Goodfellow in 2014. After reading this skill, we should be able to answer the following questions about GANs:

- What are generative adversarial networks and how they are different from the other known generative learning frameworks?
- What is the main objective of the generator and the discriminator component of a GAN based model architecture?
- How GAN objective is related to the Jenson-Shannon divergence?
- How can we implement the GAN training algorithm?
- What are the common advantages or dis-advantages of GAN-framework as compared to other generative learning frameworks?

With this skill 2 ends here. Next, we will learn about some failure modes while training GANs in practice.

You can try next...

- Understand the GAN objective and think about improving it.
- Think about other ways of learning a likelihood-free generative model.
- Think about the possible applications of GANs.

End of Skill-2

SKILL 3

GAN Failure Modes

The idea of two competing neural networks is no doubt interesting; where, at each step one of them attempts to defeat the other one and in the process, both networks keep getting better at their job. But building such a dynamic training system is not always feasible. Generative Adversarial Networks or GANs are really difficult to train in practice. GANs usually suffer from two common types of failure modes during training. It is really important to understand these failure modes before training a GAN based model otherwise, we might not be able to develop and train a stable GAN based model. This skill will mainly focus on explaining those two common failure modes in details, along with python examples. Once we are able to understand these failure modes, it will be easier for us to start developing and training our own stable GAN based models to generate the data of our choosing.

This skill covers the following topics:

- Why GAN training is unstable?
- What are GAN failure modes?
- What is mode collapse in GAN training?
- What is convergence failure in GAN training?
- Summary

Let's get started.

1. Why GAN training is unstable?

Generative Adversarial Networks, or GANs for short, are quite difficult to train in practice. GAN training can be visualized as a two-player zero sum game. The game between the generator and the discriminator is non-cooperative which means that improvements in one model come at the expense of the other model. This kind of setup is only meaningful (or stable)

when both players are capacity-wise balanced. For example: If one of the players is very strong, it will dominate the game and will not let the weaker player gain anything. So, basically the weaker model might not be able to learn anything useful. And thus, the balance between two competing models is very important.

Finding this kind of balance between the generator and the discriminator model can be related to the problem of finding Nash Equilibrium (a game theory concept that determines the optimal solution in a non-cooperative game). A Nash Equilibrium occurs when each player has the minimal cost. Both the players in a GAN setup are neural networks (functions of high-dimensional parameter space), parameters are continuous variables, and the cost functions are non-convex. Finding the Nash Equilibrium in such a setup is not trivial. Also, there is no known algorithm for the same.

The key idea of keeping each player at minimum cost (Nash Equilibrium) motivates the use of traditional gradient descent-based learning methods. These learning methods can be leveraged for minimizing the cost of both players simultaneously. But in our GAN setup, modification in one player's cost using gradient descent, might influence the other player in a wrong way as well. This is the key reason that leads to the convergence failure of GANs very frequently.

As we can see that GANs are very unstable and difficult to train. Thus, it is important to study these common failure modes before developing GANs. Let's learn about the common GAN failure modes.

1. GAN failure modes

The best way to check whether the GAN training is successful or not, is to monitor the generated samples very carefully and frequently (along with model loss). After looking at the generated content for a few iterations of training, an experienced AI practitioner will know whether the training is going in the right direction or not. If the training is not going in the right direction, one should stop the training and make required changes to the

model and start training again. This skill will mainly focus on the failure modes of GAN training, while the next skill shows some of the best practices for developing stable GAN architectures.

Before starting to develop or train GANs, it is important to understand the common failure modes of GAN training. Although, there can be lot many reasons behind the training failure of a GAN like setup, and sometimes they may even be related to the training dataset as well. It is really difficult to identify the exact reason of failure every time. However, the researchers have observed the following two failure modes to be occurring very frequently in case of GANs.

- Mode Collapse
- Convergence Failure

Let's learn more about these failure modes.

1. Mode Collapse in GANs

Whenever we train a GAN based generative model, we expect it to generate wide variety of realistic samples. But it's not always the case. Sometimes, due to an unbalanced training setup, the generator network fails to learn the correct representations and thus, fails to generate large variety of content. This problem with the training of GANs is known as 'mode collapse'. Mode collapse is quite easy to detect but very difficult to solve. Basically, mode collapse means that the generator model collapses to generate only a limited variety of samples. This failure can be easily detected by inspecting the generated samples during the training phase. If there is very little diversity in the generated samples and some of the generated samples are exactly similar, then there are high chances of mode collapse.

Sometimes, all of the generated images are exactly same. Such scenario is known as a complete collapse. And the case when the generator collapses to a small variety of samples is termed as a partial collapse of GAN. Partial collapse is quite common whereas we don't get to see a complete collapse very often. Let's learn about the reasons behind a mode collapse.

3.1 Reasons behind Mode Collapse

Mode Collapse happens when the generator network fails to generate a wide variety of content. If the generator network is having a mode collapse and as a result it is producing similar kind of samples again and again, Then, the duty of an ideal discriminator network is to learn to reject those repeated samples as fake. And thus, it should make the generator put some more work into generating samples. But it is not always the case. Sometimes, the discriminator network gets stuck in a local minimum and fails to get out of it in future training steps, the generator takes advantage of this situation and learns to generate only a few variations of images which are good enough to fool the discriminator. In other words, both the players keep optimizing to exploit the short-term weakness of the opponent.

A generator network collapsing for a small variety of samples, even when the input noise vector (z) is always different, indicates that the generator output is independent of z (in simple words, the gradient associated with z approaches to zero). Now that we have built an intuition behind the most probable reasons of model collapse, let's see how can we solve it.

3.2 Solving Mode Collapse

Mode collapse, a very common problem leading to the failure of GANs, has led the researchers to find out multiple different ways to solve it. However, we can't confidently claim that a particular hack would work in a given situation. Following is a list of some of the common hacks (or best practices) to solve mode collapse and make the GANs output wide variety of samples:

- Wasserstein GAN solves this problem with the help of a new loss function that gives the flexibility of training the discriminator to optimality without worrying about the vanishing gradients problem. We will learn about Wasserstein GANs later in this book.
- Unrolled GANs use a different loss function for the generator that doesn't let it over-optimize for a particular discriminator state. This setup takes feedback from multiple discriminators while updating the generator parameters.

- Increasing the dimensions of the input vector (z) can introduce more variety to the generated content.
- Making the generator more complex (or deeper), such that it is capable of learning the complex representations/features.
- Impairing the discriminator by randomly giving some false labels.

The hacks listed above have proved to be useful in solving many mode collapse cases as per the study conducted by researchers. We may not be able to tell which hack might solve a given mode collapse problem, but with these hacks, our we should be able to solve most of our mode collapse related issues. Next, let's learn about another common failure mode of GANs: Convergence Failure.

1. Convergence Failure in GANs

Convergence failure is a scenario when GAN training fails to converge without even producing any meaningful content. GAN training is a two-player zero sum game where each player learns at the cost of the other one (at each iteration), such a setup needs both players to be chosen carefully in order to establish an equilibrium. If one of the players dominates, the whole GAN setup fails to converge. This kind of problem with the training of GANs is known as *Convergence Failure*. And in case of a convergence failure, GANs don't produce any meaningful output.

Convergence failure is quite frequently observed during the training of GANs. Following are the ways to identify convergence failure:

- The loss of the discriminator network rapidly decreases to a value very close to zero and stays there during the remaining training iterations.
- The samples generated by the generator network are very bad quality and it's a trivial job for the discriminator to flag them as fake.

- The loss of the generator network either reaches zero or keeps continuously increasing during the entire training phase.

If our GAN training fails and shows the behavior related to any of the above-described scenarios, our model is most probably having a convergence failure situation. Next, let's learn about the main reasons behind the convergence failure of GAN training.

4.1 Reasons behind Convergence Failure

Convergence failure happens when there is no balance between the generator and the discriminator networks. There can be two possible reasons behind this imbalance between the two networks. Following are those two possible scenarios:

- The first scenario is observed when the discriminator is weak, and the generator dominates. In this case, the discriminator is incapable of distinguishing real samples from the fake ones and thus it is not able to provide useful feedbacks to the generator. The generator loss reaches close to zero despite the bad quality of samples generated by it.
- Another possible scenario is when the discriminator is too good at its job. It means that the samples generated by the generator are always rejected and the generator is never able to fool the discriminator during the entire training process. The generator loss in such cases, keeps increasing during training and it produces garbage outputs.

We now understand the key reasons behind the convergence failure of a GAN training setup. Let's learn about some ways of solving it.

4.2 Solving Convergence Failure

There are quite a few tricks to solve the convergence failure problem of GANs. All we have to do is: make the generator and discriminator networks balanced, such that neither of them dominates during the training. Let's learn about some ways of balancing the two networks.

Solving for the first scenario where the Generator dominates:

- Make the discriminator deeper, thus better at rejecting the fake samples and making the generator put more work.
- Impairing the generator by adding dropout layers, thus it learns slower and in sync with the discriminator network.
- Making the generator weaker by removing some layers.

Solving for the second scenario where the Discriminator dominates:

- Impairing the discriminator by randomly giving some false labels to real images.
- Impairing the discriminator by adding dropouts (or other regularization techniques).
- Making the generator deeper, thus better at learning representations/features and producing realistic samples.
- Making the discriminator weaker by removing few layers, thus it learns slowly and in sync with the generator.

We have just learned about multiple tricks to balance the models for each scenario of convergence failure. All we have to do here is to balance the learning process of both models so that they learn in sync and keep improving each other. Now let's jump into some hands-on experiments.

1. Experiments

In this skill, we have learned about two common failure modes of GAN training: mode collapse and convergence failure. In this section, we will try to reproduce both of these failure scenarios. Basically, we will perform the following two experiments:

- Reproducing the mode collapse failure of GANs.
- Reproducing the convergence failure of GANs.

The code samples shown in this skill can be downloaded from the following Github location:

<https://github.com/kartikgill/The-GAN-Book/tree/main/Skill-03>

Let's get started.

MODE COLLAPSE IN GAN TRAINING

Objective

In this experiment, we will try to reproduce the mode collapse failure scenario of a GAN training. Let's get started.

This experiment has the following key steps:

- Importing useful libraries
- Download Dataset and show samples
- Data Normalization
- Define Generator
- Define Discriminator
- Define GAN
- Utility Functions
- Training GAN
- Results (Mode Collapsed)
- Loss chart

Let's go step-by-step.

Step 1: Importing useful libraries

First step is to open a Jupyter Notebook with python kernel and import useful libraries in a cell. We will be using TensorFlow library for developing the GAN architecture.

Checkout the following code for importing libraries:

```
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
from tqdm import tqdm_notebook  
%matplotlib inline  
import tensorflow  
print(tensorflow.__version__)
```

Now, let's download the dataset.

Step 2: Download Dataset and show samples

In this step, we will download the MNIST Handwritten Digits dataset and plot some samples to verify if everything is correct. The following python code downloads the MNIST data from Keras datasets and plots a few images.

```
from tensorflow.keras.datasets import mnist
(trainX, trainY), (testX, testY) = mnist.load_data()
print('Training data shapes: X=%s, y=%s' % (trainX.shape, trainY.shape))

print('Testing data shapes: X=%s, y=%s' % (testX.shape, testY.shape))
for k in range(9):
    plt.figure(figsize=(9, 6))
    for j in range(9):
        i = np.random.randint(0, 10000)
        plt.subplot(990 + 1 + j)
        plt.imshow(trainX[i], cmap='gray_r')
        #plt.title(trainY[i])
        plt.axis('off')
    plt.show()
```

Figure 3.1 shows some samples from the dataset. This dataset has roughly 60k training images and 10k test images of size 28 x 28. Each image also has a label associated with it: indicating the digit class.



Figure 3.1: A glimpse of MNIST Handwritten Digits Dataset

Now we have data with us. Let's prepare it for the model training.

Step 3: Data Normalization

In this step, we will first normalize the pixel values of all the images by dividing them with 255. This will restrict each image array to small range [0, 1]. Secondly, we will add a channel dimension to each of the training and testing images. As these images are grayscale, they will just have a single channel.

See the following python code for data preparation:

```
trainX = [image/255.0 for image in trainX]
testX = [image/255.0 for image in testX]
trainX = np.reshape(trainX, (60000, 28, 28, 1))
testX = np.reshape(testX, (10000, 28, 28, 1))
print (trainX.shape, testX.shape, trainY.shape, testY.shape)
```

Final data set has the following dimensions:

Out[3]: (60000, 28, 28, 1) (10000, 28, 28, 1) (60000,) (10000,)

Our dataset is now ready. Let's define the model architecture now.

Step 4: Define Generator

In this step, we will define the generator network. We will keep the generator network simple which will accept a noise vector of size 2 as input and generate an image of size 28 x 28 x 1 as output. In this example, our generator network passes the noise input into two fully connected layers and a reshaping layer to get a 3-dimensional image output. Final layer uses ‘sigmoid’ activation function to restrict the pixel range within [0, 1], as we have in our training data.

The following python code defines the generator network:

```
random_input = tensorflow.keras.layers.Input(shape = 2)
x = tensorflow.keras.layers.Dense(64, activation='relu')(random_input)
x = tensorflow.keras.layers.Dense(28*28)(x)
x = tensorflow.keras.layers.Reshape((28, 28, 1))(x)
generated_image = tensorflow.keras.layers.Activation('sigmoid')(x)
generator_network = tensorflow.keras.models.Model(inputs=random_input, outputs=generated_image)
generator_network.summary()
```

Following is a summary of the generator network. The network is quite simple and has around 51k trainable parameters.

Out[4]: Model: "model"

Layer (type)	Output Shape	Param #
<hr/>		
input_1 (InputLayer)	[(None, 2)]	0
dense (Dense)	(None, 64)	192
dense_1 (Dense)	(None, 784)	50960
reshape (Reshape)	(None, 28, 28, 1)	0
activation (Activation)	(None, 28, 28, 1)	0

```
=====
Total params: 51,152
Trainable params: 51,152
Non-trainable params: 0
```

The generator network is ready, let's work on the discriminator network now.

Step 5: Define Discriminator

In this step, we will define the discriminator network. The architecture of the discriminator network would be very similar to a binary image classification model. The discriminator network should accept an image as input and return back a probability value indicating whether the image is real or fake. In our setup, the discriminator network first flattens the input image into a one-dimensional vector and then passes it through a fully connected and activation layer. The final layer has a single neuron and a ‘sigmoid’ activation function to generate a single probability value as output.

The following python snippet defines the discriminator network:

```
image_input = tensorflow.keras.layers.Input(shape=(28, 28, 1))
x = tensorflow.keras.layers.Flatten()(image_input)
x = tensorflow.keras.layers.Dense(64, activation='relu')(x)
x = tensorflow.keras.layers.Dense(1)(x)
real_vs_fake_output = tensorflow.keras.layers.Activation('sigmoid')(x)
discriminator_network = tensorflow.keras.models.Model(inputs=image_i
nput, outputs=real_vs_fake_output)
discriminator_network.summary()
```

Following is the summary of the discriminator network. It roughly has 50k trainable parameters.

Out[5]: Model: "model_1"

Layer (type)	Output Shape	Param #
--------------	--------------	---------

```
=====
input_2 (InputLayer) [(None, 28, 28, 1)] 0
```

```
flatten (Flatten) (None, 784) 0
```

```
dense_2 (Dense) (None, 64) 50240
```

```
dense_3 (Dense) (None, 1) 65
```

```
activation_1 (Activation) (None, 1) 0
```

```
=====
Total params: 50,305
```

```
Trainable params: 50,305
```

```
Non-trainable params: 0
```

We will compile our discriminator network with *binary_crossentropy* loss and use Adam optimizer with a learning rate of 0.0001 for updating the network weights. The following python snippet compiles the discriminator network:

```
adam_optimizer = tensorflow.keras.optimizers.Adam(learning_rate=0.00  
01, beta_1=0.5)
```

```
discriminator_network.compile(loss='binary_crossentropy', optimizer=ad  
am_optimizer, metrics=['accuracy'])
```

The discriminator network is all set now. Let's define the combined model now.

Step 6: Define GAN

In this step, we will define the combined GAN model. This combined model will be required while updating the weights of the generator network.

For this combined model, it is important to freeze the weights of the discriminator network. As during this pass, we only want the generator network weights to be updated based on the feedback from the discriminator network. The gradient for the generator network will flow through the frozen layers of the discriminator network first.

The following python code defines the combined model:

```
discriminator_network.trainable=False  
g_output = generator_network(random_input)  
d_output = discriminator_network(g_output)  
gan_model = tensorflow.keras.models.Model(random_input, d_output)  
gan_model.summary()
```

Following is the summary of the combined GAN model. We can see that it only has around 51k trainable parameters. That is because the discriminator weights have been frozen by setting the '*model_object.trainable=False*' flag.

Out[6]: Model: "model_2"

Layer (type) Output Shape Param #

input_1 (InputLayer) [(None, 2)] 0

model (Functional) (None, 28, 28, 1) 51152

model_1 (Functional) (None, 1) 50305

Total params: 101,457

Trainable params: 51,152

Non-trainable params: 50,305

We can compile the combined model also with same loss function and optimizer that we used for the discriminator network earlier.

```
gan_model.compile(loss='binary_crossentropy', optimizer=adam_optimizer)
```

Our overall GAN setup is now ready for training. Let's define some utility functions for preparing data batches for the training.

Step 7: Utility Functions

In this step, we will define some utility function that will help us in creating data batches for model training. Specifically, we will define utility function to get batches of: random noise, fake samples from the generator and real samples from the dataset. In addition to these, we will define one more function that will plot the fake samples generated by the generator network.

See the code below for utility functions:

```
indices = [i for i in range(60000)]  
  
def get_random_noise(batch_size, noise_size):  
    random_values = np.random.randn(batch_size*noise_size)  
    random_noise_batch = np.reshape(random_values, (batch_size, noise_size))  
    return random_noise_batch  
  
def get_fake_samples(generator_network, batch_size, noise_size):  
    random_noise_batch = get_random_noise(batch_size, noise_size)  
    fake_samples = generator_network.predict_on_batch(random_noise_batch)  
    return fake_samples  
  
def get_real_samples(batch_size):  
    random_indices = np.random.choice(indices, size=batch_size)  
    real_images = trainX[np.array(random_indices),:]  
    return real_images  
  
def show_generator_results(generator_network):  
    fake_samples = get_fake_samples(generator_network, 81, noise_size)
```

```

counter = 0
for k in range(9):
    plt.figure(figsize=(9, 6))
    for j in range(9):
        plt.subplot(990 + 1 + j)
        plt.imshow(fake_samples[counter, :, :, 0], cmap='gray_r')
    counter += 1
    plt.axis('off')
plt.show()
return

```

Now that everything from the model side and the data side is ready. Let's go ahead and train the model.

Step 8: Training GAN

In this step, we will train our GAN. We will train our model for more than 100 epochs, with a batch size of 60 and each epoch will have 500 steps of gradient updates. The training iteration loop is quite simple, very similar to the one we implemented in last skill. As a first step in each iteration, we update the weights of the discriminator model based on a batch of mixed real and fake samples. Then, we freeze the discriminator network and update the weights of the generator network using the inverted loss value of the discriminator only on fake samples. To calculate inverted loss, we pass the labels of fake samples as 1, in order to make the discriminator believe that these are real samples. At every 50th step, we also log the loss values and discriminator accuracy.

The following python code implements the GAN training iteration loop as described in the last paragraph:

```

epochs = 500
batch_size = 60
steps = 500
noise_size = 2
losses_d = []

```

```

losses_g = []
for i in range(0, epochs):
    if (i%5 == 0):
        show_generator_results(generator_network)
    for j in range(steps):
        fake_samples = get_fake_samples(generator_network, batch_size//2, noise_size)
        real_samples = get_real_samples(batch_size=batch_size//2)
        fake_y = np.zeros((batch_size//2, 1))
        real_y = np.ones((batch_size//2, 1))
        input_batch = np.vstack((fake_samples, real_samples))
        output_labels = np.vstack((fake_y, real_y))

        # Updating Discriminator weights
        discriminator_network.trainable=True
        loss_d = discriminator_network.train_on_batch(input_batch, output_labels)

        gan_input = get_random_noise(batch_size, noise_size)
        # Make the Discriminator believe that these are real samples and calculate loss to train the generator

        gan_output = np.ones((batch_size))
        # Updating Generator weights
        discriminator_network.trainable=False
        loss_g = gan_model.train_on_batch(gan_input, gan_output)
        losses_d.append(loss_d[0])
        losses_g.append(loss_g)

        if j%50 == 0:
            print ("Epoch:%.0f, Step:%.0f, D-Loss:%.3f, D-Acc:%.3f, G-Loss:%.3f" % (i,j,loss_d[0],loss_d[1]*100,loss_g))

```

Following are the training logs showing the loss values of both models at every 50th step, along with the accuracy value of the discriminator network:

Out[8]:Epoch:0, Step:0, D-Loss:0.523, D-Acc:51.667, G-Loss:1.977

Epoch:0, Step:50, D-Loss:0.083, D-Acc:100.000, G-Loss:3.692

Epoch:0, Step:100, D-Loss:0.033, D-Acc:100.000, G-Loss:4.743

Epoch:0, Step:150, D-Loss:0.014, D-Acc:100.000, G-Loss:5.357

Epoch:0, Step:200, D-Loss:0.015, D-Acc:100.000, G-Loss:5.947

Epoch:0, Step:250, D-Loss:0.012, D-Acc:100.000, G-Loss:6.363

Epoch:0, Step:300, D-Loss:0.010, D-Acc:100.000, G-Loss:7.202

Epoch:0, Step:350, D-Loss:0.008, D-Acc:100.000, G-Loss:7.691

Epoch:0, Step:400, D-Loss:0.032, D-Acc:96.667, G-Loss:7.625

Epoch:0, Step:450, D-Loss:0.007, D-Acc:100.000, G-Loss:7.592

Epoch:1, Step:0, D-Loss:0.008, D-Acc:100.000, G-Loss:7.749

Epoch:1, Step:50, D-Loss:0.010, D-Acc:100.000, G-Loss:7.553

Epoch:1, Step:100, D-Loss:0.012, D-Acc:100.000, G-Loss:7.499

Epoch:1, Step:150, D-Loss:0.049, D-Acc:95.000, G-Loss:7.908

Epoch:1, Step:200, D-Loss:0.011, D-Acc:100.000, G-Loss:7.052

Epoch:1, Step:250, D-Loss:0.042, D-Acc:96.667, G-Loss:7.347

Epoch:1, Step:300, D-Loss:0.021, D-Acc:100.000, G-Loss:7.132

Epoch:1, Step:350, D-Loss:0.049, D-Acc:98.333, G-Loss:6.748

Epoch:1, Step:400, D-Loss:0.058, D-Acc:98.333, G-Loss:7.191

Epoch:1, Step:450, D-Loss:0.079, D-Acc:98.333, G-Loss:6.986

Epoch:2, Step:0, D-Loss:0.070, D-Acc:98.333, G-Loss:6.238

Epoch:2, Step:50, D-Loss:0.091, D-Acc:100.000, G-Loss:5.832

Epoch:2, Step:100, D-Loss:0.107, D-Acc:96.667, G-Loss:5.115

Epoch:2, Step:150, D-Loss:0.137, D-Acc:98.333, G-Loss:5.157

...

...

...

Epoch:149, Step:100, D-Loss:0.023, D-Acc:98.333, G-Loss:5.255

Epoch:149, Step:150, D-Loss:0.026, D-Acc:100.000, G-Loss:5.282

Epoch:149, Step:200, D-Loss:0.008, D-Acc:100.000, G-Loss:4.986

Epoch:149, Step:250, D-Loss:0.040, D-Acc:98.333, G-Loss:4.405

Epoch:149, Step:300, D-Loss:0.039, D-Acc:98.333, G-Loss:4.778

Epoch:149, Step:350, D-Loss:0.064, D-Acc:98.333, G-Loss:4.299

Epoch:149, Step:400, D-Loss:0.036, D-Acc:100.000, G-Loss:4.108

Epoch:149, Step:450, D-Loss:0.018, D-Acc:100.000, G-Loss:4.369

If we look closely at the loss trends, the discriminator model very soon becomes an expert in identifying real vs fake samples. During the most part of the training, it has close to 100% classification accuracy. The generator network on the other hand, does not converge and its loss values keeps fluctuating. This is a failure scenario of GAN training as the generator is not able to fool the discriminator. Let's look at the results to find out more about this failure mode.

Step 9: Results (Mode Collapsed)

We have been monitoring the outputs of the generator network at every few epochs. At epoch 50, we saw that most of the generator outputs look very similar in shape and it is difficult to recognize the class label. We kept training our model further but there was no improvement in the generated results even after 150 epochs of training (see Figure 3.2). As the generator is always generating same image, this is an example of model collapse and a complete collapse.

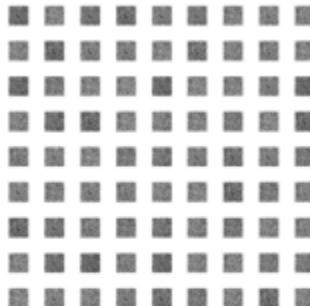
Epoch: 0	Epoch: 50
	4 3 4 9 9 9 9 8 8 8 8 8 8 3 8 9 8 8 2 3 3 9 6 2 9 8 8 2 4 2 3 6 2 4 3 8 2 8 8 8 8 2 8 3 3 8 2 9 4 4 9 3 3 3 8 8 2 9 4 2 3 2 8 2 3 8 2 4 2 2 2 8 2 9 3 3 3 3 2 2 2
Epoch: 100	Epoch: 150
8 8 8 8 8 9 3 9 8 8 8 8 8 8 8 4 4 9 8 4 2 5 5 4 8 8 8 8 8 8 8 8 8 5 4 8 8 8 8 8 8 8 8 8 8	9 9 1 9 9 1 9 9 1 1 9 9 1 9 9 9 1 9 9 1 1 9 1 9 9 9 9 1 1 1 9 1 9 9 1 9 9 1 1 9 1 9 9 9 9 1 1 1 9 9 9 9 1 9 1 9 9 1 9 9 1 1 9 9 9 1 9 1 9 9 1 9 1 1 1 9 9 1 9 9 1

Figure 3.2: Model Collapse failure model results of a GAN training

Figure 3.2 shows the outputs of a mode collapse scenario. We were able to replicate this failure method by making our generator model simpler and reducing the dimensions of input noise to 2. Let's look at the loss charts now.

Step 10: Loss chart

In this step, we are going to plot the loss charts of the discriminator and the generator networks training progress along with training steps.

The following python snippet generates the loss chart:

```
steps = [i for i in range(len(losses_d))]

plt.figure(figsize=(10, 6))

plt.plot(losses_d)
plt.plot(losses_g)

plt.xlabel('Steps')
plt.ylabel('Loss Value')
```

```

plt.title("Mode Collapse: Loss Trends")
plt.legend(['Discriminator Loss', 'Generator Loss'])
plt.show()

```

Figure 3.3 shows the resulting loss chart of this model collapse scenario. We can see that the discriminator loss is very small throughout the training but the generator loss keeps fluctuating and doesn't seem to converge.

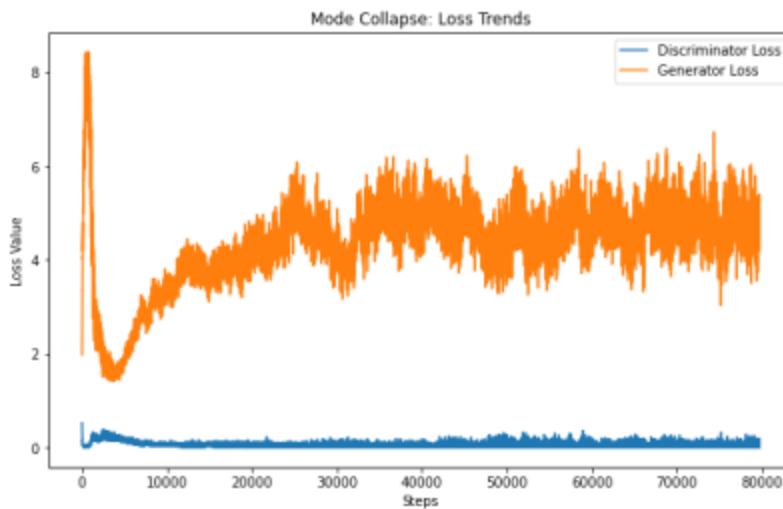


Figure 3.3: Loss chart for a mode collapse failure more scenario of GAN Training

We just replicated a training failure scenario known as mode collapse, for a simple custom GAN. If we encounter a loss chart, that is similar to the one in Figure 3.3, it can be a model collapse scenario. With this experiment, we should now have a good understanding of mode collapse and we should keep this in mind while developing GANs in future.

Next, we will move to the second experiment that is related to ‘Convergence Failure’.

CONVERGENCE FAILURE IN GANS

Objective

In this experiment, we will reproduce the convergence failure mode of GAN training intentionally, just for the understanding purpose.

This experiment has the following steps:

- Importing useful libraries
- Download Dataset and show samples
- Data Normalization
- Define Generator
- Define Discriminator
- Define GAN
- Utility Functions
- Training GAN
- Results (Convergence Failure)
- Loss chart

Let's get started.

Step 1: Importing useful libraries

The first step is to open a Jupyter Notebook and import useful libraries. Here '*pandas*' and '*numpy*' are for dataset manipulation, while '*matplotlib*' is for plotting images and loss charts. We will utilize TensorFlow for developing the GAN architecture.

```
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
from tqdm import tqdm_notebook  
%matplotlib inline  
import tensorflow  
print(tensorflow.__version__)
```

Next, let's download the dataset.

Step 2: Download Dataset and show samples

In this step, we will download the MNIST Handwritten Digits dataset and plot some samples to verify if everything is correct. The following python code downloads the MNIST data from Keras datasets and plots a few images.

```
from tensorflow.keras.datasets import mnist
(trainX, trainY), (testX, testY) = mnist.load_data()
print('Training data shapes: X=%s, y=%s' % (trainX.shape, trainY.shape))

print('Testing data shapes: X=%s, y=%s' % (testX.shape, testY.shape))
for k in range(9):
    plt.figure(figsize=(9, 6))
    for j in range(9):
        i = np.random.randint(0, 10000)
        plt.subplot(990 + 1 + j)
        plt.imshow(trainX[i], cmap='gray_r')
        #plt.title(trainY[i])
        plt.axis('off')
    plt.show()
```

Figure 3.4 shows some samples from the dataset. This dataset has roughly 60k training images and 10k test images of size 28 x 28. Each image also has a label associated with it: indicating the digit class.

```

Downloading data from https://storage.googleapis.com/tensorflow/
11493376/11490434 [=====] - 0s 0us/1
Training data shapes: X=(60000, 28, 28), y=(60000,)
Testing data shapes: X=(10000, 28, 28), y=(10000,)

  9  0  5  4  2  8  7  3  9
  7  2  0  0  9  7  2  1  7
  3  9  4  1  2  4  8  3  7
  4  0  2  4  5  5  7  0  3
  2  7  4  2  8  7  4  2  3
  5  1  6  2  5  0  9  2  8
  9  1  1  9  6  9  4  2  9
  4  4  5  3  2  1  0  6  6
  6  6  7  0  1  7  3  1  8

```

Figure 3.4: Few samples from the MNIST Handwritten Digits Dataset

We now have data with us. Let's prepare it for the model.

Step 3: Data Normalization

In this step, we will first normalize the pixel values of all the images by dividing them with 255. This will restrict each image array to small range [0, 1]. Secondly, we will add a channel dimension to each of the training and testing images. As these images are grayscale, they will just have a single channel.

See the following python code for data preparation:

```

trainX = [image/255.0 for image in trainX]
testX = [image/255.0 for image in testX]
trainX = np.reshape(trainX, (60000, 28, 28, 1))
testX = np.reshape(testX, (10000, 28, 28, 1))
print (trainX.shape, testX.shape, trainY.shape, testY.shape)

```

The following output shows the final shapes of training and test datasets.

Out[3]: (60000, 28, 28, 1) (10000, 28, 28, 1) (60000,) (10000,)

Our data is now ready. Let's define the model.

Step 4: Define Generator

In this step, we will define the generator network. We will keep the generator network simple which will accept a noise vector of size 50 as input and generate an image of size 28 x 28 x 1 as output. In this example, our generator network passes the noise input into a fully connected layer, ReLU activation and a reshaping layer to get a 3-dimensional image output. Final layer uses '*sigmoid*' activation function to restrict the pixel range within [0, 1], as we have restricted in our training data as well.

The following python code defines the generator network:

```
random_input = tensorflow.keras.layers.Input(shape = 50)
x = tensorflow.keras.layers.Dense(28*28, activation='relu')
(random_input)

x = tensorflow.keras.layers.Reshape((28, 28, 1))(x)
generated_image = tensorflow.keras.layers.Activation('sigmoid')(x)
generator_network = tensorflow.keras.models.Model(inputs=random_input, outputs=generated_image)

generator_network.summary()
```

Following is the summary of the generator network. It is extremely simple and has just approximately 40k trainable parameters.

Out[4]: Model: "model"

Layer (type) Output Shape Param #

=====

input_1 (InputLayer) [(None, 50)] 0

dense (Dense) (None, 784) 39984

reshape (Reshape) (None, 28, 28, 1) 0

```
activation (Activation) (None, 28, 28, 1) 0
```

```
=====
```

Total params: 39,984

Trainable params: 39,984

Non-trainable params: 0

Now that our generator network is ready. Let's work on the discriminator network.

Step 5: Define Discriminator

In this step, we will define the discriminator network. The architecture of the discriminator network would be quite similar to a binary classification model. The discriminator network should accept an image of size 28 x 28 x 1, as input and return back a probability value: indicating whether the image is real or fake, as output. In our setup, the discriminator network first flattens the input image into a one-dimensional vector and then passes it through three fully connected layers and ReLU activation layers. The final layer has a single neuron and a '*sigmoid*' activation function to generate a single probability value as output.

The following python snippet defines the discriminator network:

```
image_input = tensorflow.keras.layers.Input(shape=(28, 28, 1))
x = tensorflow.keras.layers.Flatten()(image_input)
x = tensorflow.keras.layers.Dense(512, activation='relu')(x)
x = tensorflow.keras.layers.Dense(256, activation='relu')(x)
x = tensorflow.keras.layers.Dense(128, activation='relu')(x)
x = tensorflow.keras.layers.Dense(1)(x)
real_vs_fake_output = tensorflow.keras.layers.Activation('sigmoid')(x)
discriminator_network = tensorflow.keras.models.Model(inputs=image_input, outputs=real_vs_fake_output)
```

```
discriminator_network.summary()
```

Following is the summary of the discriminator network. It has roughly 566k trainable parameters. Our discriminator network seems quite complex as compared to the generator network, but that's intentional as we plan to replicate the convergence failure mode by making one model dominate the other.

Out[5]: Model: "model_1"

```
Layer (type) Output Shape Param #
```

```
=====
```

```
input_2 (InputLayer) [(None, 28, 28, 1)] 0
```

```
flatten (Flatten) (None, 784) 0
```

```
dense_1 (Dense) (None, 512) 401920
```

```
dense_2 (Dense) (None, 256) 131328
```

```
dense_3 (Dense) (None, 128) 32896
```

```
dense_4 (Dense) (None, 1) 129
```

```
activation_1 (Activation) (None, 1) 0
```

```
=====
```

```
Total params: 566,273
```

```
Trainable params: 566,273
```

```
Non-trainable params: 0
```

As the discriminator model is simply a classifier, we can compile it using ‘*binary_crossentropy*’ loss. We will be using Adam optimizer with a learning rate of 0.0001 and beta_1 value of 0.5, for updating the model weights.

```
adam_optimizer = tensorflow.keras.optimizers.Adam(learning_rate=0.0001, beta_1=0.5)
```

```
discriminator_network.compile(loss='binary_crossentropy', optimizer=adam_optimizer, metrics=['accuracy'])
```

The discriminator network is all set now. Let’s work on defining the final GAN model.

Step 6: Define GAN

In this step, we will define the combined GAN model. This combined model will be useful while updating the weights of the generator network. In this combined model, it is important to freeze the weights of the discriminator network. As during this pass, we only want the generator network weights to be updated based on the feedback from the discriminator network. The gradient for the generator network will flow through the frozen layers of the discriminator network first.

The following python code defines the combined model:

```
discriminator_network.trainable=False  
g_output = generator_network(random_input)  
d_output = discriminator_network(g_output)  
gan_model = tensorflow.keras.models.Model(random_input, d_output)  
gan_model.summary()
```

Following is a summary of the combined model. It roughly has 40k trainable parameters. All these trainable parameters belong to the generator network as the weights of the discriminator network are frozen for this combined setup.

Out[6]: Model: "model_2"

Layer (type) Output Shape Param #

=====

```
input_1 (InputLayer) [(None, 50)] 0
```

```
model (Functional) (None, 28, 28, 1) 39984
```

```
model_1 (Functional) (None, 1) 566273
```

```
=====
```

```
Total params: 606,257
```

```
Trainable params: 39,984
```

```
Non-trainable params: 566,273
```

We can now compile the combined model. We will be using ‘*binary_crossentropy*’ as a loss function and Adam optimizer for updating the weights of the generator network via a frozen discriminator.

```
gan_model.compile(loss='binary_crossentropy', optimizer=adam_optimizer)
```

Our GAN setup is ready for training. Let’s define some utility functions for getting data batches during the training loop.

Step 7: Utility Functions

In this step, we will define some utility function that will help us in creating data batches for model training. Specifically, we will define utility function to get batches of: random noise, fake samples from the generator and real samples from the dataset. In addition to these, we will define one more function that will plot the fake samples generated by the generator network.

See the code below for utility functions:

```
indices = [i for i in range(60000)]  
def get_random_noise(batch_size, noise_size):  
    random_values = np.random.randn(batch_size*noise_size)
```

```

random_noise_batch = np.reshape(random_values, (batch_size, noise_size))
e)
return random_noise_batch

def get_fake_samples(generator_network, batch_size, noise_size):
    random_noise_batch = get_random_noise(batch_size, noise_size)
    fake_samples = generator_network.predict_on_batch(random_noise_batch)
    return fake_samples

def get_real_samples(batch_size):
    random_indices = np.random.choice(indices, size=batch_size)
    real_images = trainX[np.array(random_indices), :]
    return real_images

def show_generator_results(generator_network):
    fake_samples = get_fake_samples(generator_network, 81, noise_size)
    counter = 0
    for k in range(9):
        plt.figure(figsize=(9, 6))
        for j in range(9):
            plt.subplot(990 + 1 + j)
            plt.imshow(fake_samples[counter, :, :, 0], cmap='gray_r')
            counter += 1
        plt.axis('off')
        plt.show()
    return

```

We are now all set to start training the GAN.

Step 8: Training GAN

In this step, we will train our GAN. We will train our model for more than 100 epochs, with a batch size of 60 and each epoch will have 500 steps of gradient updates. The training iteration loop is quite simple, very similar to

the one we implemented in the last experiment. As a first step in each iteration, we update the weights of the discriminator model based on a batch of mixed real and fake samples. Then, we freeze the discriminator network and update the weights of the generator network using the inverted loss value of the discriminator only on fake samples. To calculate inverted loss, we pass the labels of fake samples as 1, in order to make the discriminator believe that these are real samples. At every 50th step, we also log the loss values and discriminator accuracy.

The following python code implements the GAN training iteration loop as described in the last paragraph:

```
epochs = 500
batch_size = 60
steps = 500
noise_size = 50
losses_d = []
losses_g = []
for i in range(0, epochs):
    if (i%5 == 0):
        show_generator_results(generator_network)
    for j in range(steps):
        fake_samples = get_fake_samples(generator_network, batch_size//2, noise_size)
        real_samples = get_real_samples(batch_size=batch_size//2)
        fake_y = np.zeros((batch_size//2, 1))
        real_y = np.ones((batch_size//2, 1))
        input_batch = np.vstack((fake_samples, real_samples))
        output_labels = np.vstack((fake_y, real_y))
        # Updating Discriminator weights
        discriminator_network.trainable=True
```

```

loss_d = discriminator_network.train_on_batch(input_batch, output_label
s)
gan_input = get_random_noise(batch_size, noise_size)
# Make the Discriminator believe that these are real samples and calculate loss to train the generator

gan_output = np.ones((batch_size))
# Updating Generator weights
discriminator_network.trainable=False
loss_g = gan_model.train_on_batch(gan_input, gan_output)
losses_d.append(loss_d[0])
losses_g.append(loss_g)
if j%50 == 0:
    print ("Epoch:%.0f, Step:%.0f, D-Loss:%.3f, D-Acc:%.3f, G-Loss:%.3f"% (i,j,loss_d[0],loss_d[1]*100,loss_g))

```

Following are the training logs for our setup. We can clearly see that the discriminator network starts dominating very soon and achieves 100% accuracy and a loss value close to zero. Whereas, the generator network is not able to learn anything and its loss value keeps increasing. This is a typical scenario of convergence failure, where the discriminator is dominating the generator network.

Out[8]: Epoch:0, Step:0, D-Loss:0.669, D-Acc:70.000, G-Loss:0.891

```

Epoch:0, Step:50, D-Loss:0.008, D-Acc:100.000, G-Loss:5.825
Epoch:0, Step:100, D-Loss:0.001, D-Acc:100.000, G-Loss:7.381
Epoch:0, Step:150, D-Loss:0.001, D-Acc:100.000, G-Loss:8.149
Epoch:0, Step:200, D-Loss:0.000, D-Acc:100.000, G-Loss:8.813
Epoch:0, Step:250, D-Loss:0.000, D-Acc:100.000, G-Loss:9.280
Epoch:0, Step:300, D-Loss:0.000, D-Acc:100.000, G-Loss:9.628
Epoch:0, Step:350, D-Loss:0.000, D-Acc:100.000, G-Loss:9.961
Epoch:0, Step:400, D-Loss:0.000, D-Acc:100.000, G-Loss:10.219
Epoch:0, Step:450, D-Loss:0.000, D-Acc:100.000, G-Loss:10.461

```

```
Epoch:1, Step:0, D-Loss:0.000, D-Acc:100.000, G-Loss:10.639
Epoch:1, Step:50, D-Loss:0.000, D-Acc:100.000, G-Loss:10.903
Epoch:1, Step:100, D-Loss:0.000, D-Acc:100.000, G-Loss:11.117
Epoch:1, Step:150, D-Loss:0.000, D-Acc:100.000, G-Loss:11.290
...
...
...
Epoch:80, Step:100, D-Loss:0.000, D-Acc:100.000, G-Loss:24.564
Epoch:80, Step:150, D-Loss:0.000, D-Acc:100.000, G-Loss:24.542
Epoch:80, Step:200, D-Loss:0.000, D-Acc:100.000, G-Loss:24.434
Epoch:80, Step:250, D-Loss:0.000, D-Acc:100.000, G-Loss:24.539
Epoch:80, Step:300, D-Loss:0.000, D-Acc:100.000, G-Loss:24.581
Epoch:80, Step:350, D-Loss:0.000, D-Acc:100.000, G-Loss:24.444
Epoch:80, Step:400, D-Loss:0.000, D-Acc:100.000, G-Loss:24.556
```

We got a good idea about the convergence failure after looking at the training logs. Let's now see some of the generated images as well.

Step 9: Results (Convergence Failure)

Figure 3.5 shows the results of the generator at different epochs. We can see that the generator being very simple as compared to the discriminator, is not able to compete. It is not able to learn anything meaningful. It just keeps producing garbage outputs throughout the training.

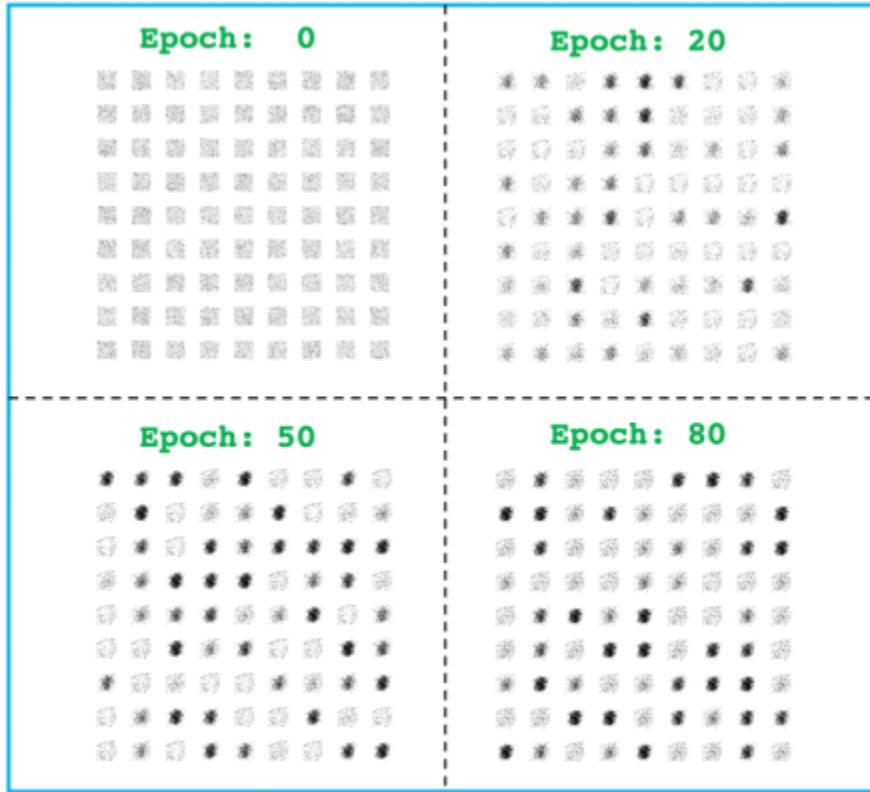


Figure 3.5: The results from a convergence failure mode of GAN training
 We can see that a GAN suffering from the convergence failure will never be able to generate meaningful content. Let's look at the loss charts of our setup now.

Step 10: Loss chart

In this step, we will plot the loss trends of the discriminator and the generator networks along with the training steps.

See the following python code:

```
steps = [i for i in range(len(losses_d))]
plt.figure(figsize=(10, 6))
plt.plot(losses_d)
plt.plot(losses_g)
plt.xlabel('Steps')
plt.ylabel('Loss Value')
plt.title("Convergence Failure: Loss Trends")
```

```
plt.legend(['Discriminator Loss', 'Generator Loss'])
```

```
plt.show()
```

The aforementioned code generates the following loss chart (see Figure 3.6).

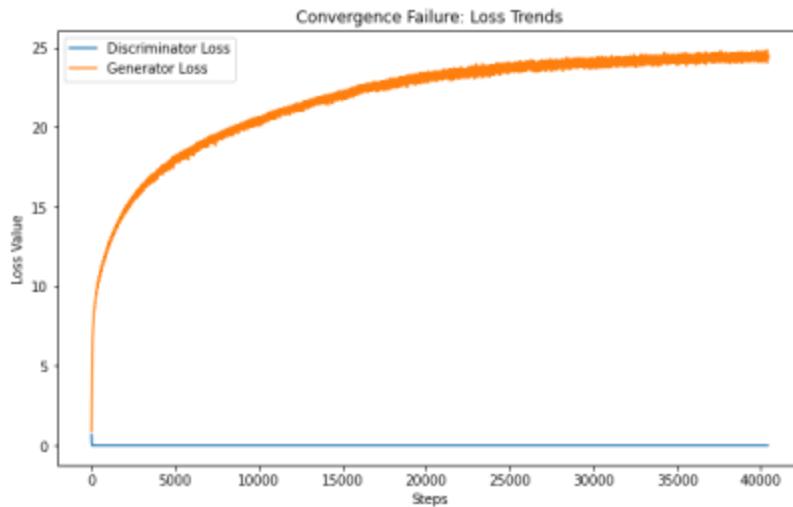


Figure 3.6: Loss chart for convergence failure model of GAN Training

In Figure 3.6, we can see that the loss value of the dominating discriminator network is close to zero during the entire training process. The loss value of the generator network keeps increasing on the other hand, leading to a convergence failure of the training. Such a setup, will never converge, no matter how long we keep training it. If we encounter any loss chart similar to this in future, it may be a case of convergence failure. With this, we will end this exercise here.

1. Summary

This skill highlights the fact that GANs are quite difficult to train. GANs usually suffer from two common training failure scenarios: model collapse and convergence failure. It is quite important to learn to identify and overcome these failure modes before starting to work on GANs. After reading this skill, the readers should be confident about answering the following questions about GAN failure modes:

- Why GANs are hard to train? What are the common problems they suffer from during training?
- How to identify the mode collapse failure during the GAN training? What are the different tricks to solve it?
- What are the common reasons behind the convergence failure of a GAN? How can we identify it during the training process? What are the tips and tricks to avoid and solve this issue?

1. you can try next...

- Think about new innovative ways of solving problems like: mode-collapse and convergence-failure.
- Maybe develop an intelligent agent (e.g., Reinforcement learning agent) that can automatically detect and solve such training failures.
- Discover better ways of training stable GAN based models.
- Find better objectives for GANs, enabling stable, powerful and failure-free GAN training.

End of Skill-3

SKILL 4

Deep Convolutional GANs

We now have a good understanding of Generative Adversarial Networks, or GANs, along with common training failure modes and key hacks to overcome them. It is now time to start developing GANs of our own and start inspecting the goodness of the generated content for our use cases. We have already verified the GANs ability to generate handwritten digit images on the MNIST dataset but that's not our final goal, as the MNIST images are very low-dimensional and gray-scale in nature. In fact, we would like to generate high-dimensional colorful images that are realistic. Images and scenes in the real world, are composed of wide variety of objects, backgrounds, colors, different lighting conditions and so on. Generating such content is a non-trivial task, unlike the case of MNIST handwritten digits generation where we only had 10 classes of digits and that too in grayscale.

Let's assume that we are interested in training a GAN based model capable of generating plausible images of human faces, given a few thousand real face images as training data. Learning such a generative model is a non-trivial task due to the large varieties of human faces, in terms of shape, hairstyle, gender, skin-color, beards, glasses and so on. In this skill, we will learn how to develop GAN based architectures that are capable of solving such complex tasks.

This skill covers the following key topics:

- What is a Deep Convolutional GAN or DCGAN?
- Guidelines for stable training of DCGANs
- Impressive Image Generation results of DCGANs
- Unsupervised Representations Learning
- Experiments
- Summary

Let's get started.

1. What is a Deep Convolutional GAN or DCGAN?

When it comes to generating image data the Convolutional Neural Networks, or CNNs for short, are a natural choice. Goodfellow et el., introduced Generative Adversarial Networks in 2014 with a demonstration of its image generation capabilities but those images were noisy and incomprehensible. As the main goal of that paper was to introduce a new way of training generative models. Shortly after the introduction of GANs, quite a few researchers tried to scale them up using CNNs but it was not a trivial task as the training was unstable. We have discussed some of these instability issues in Skill 3 as well.

In 2015, Alec Radford, Luke Metz and Soumith Chintala introduced a few settings leading towards a family of CNN based GAN architectures that resulted in stable training and they were able to generate plausible images on multiple datasets. This family of CNN based GAN architectures is also known as Deep Convolutional GANs or DCGANs.

Deep Convolutional Generative Adversarial Network, or DCGANs for short, were introduced by *Alec Radford et al.*, in their paper titled '*Unsupervised representation learning with deep convolutional generative adversarial networks*' in 2015.

Radford and team found out that it is important (*as quoted by Springenberg et al., 2014*) to use strided convolutions for the down-sampling or the up-sampling of data, in place of the deterministic pooling layers (e.g., maxpooling). Strided convolutions enable the networks to learn their own spatial mappings.

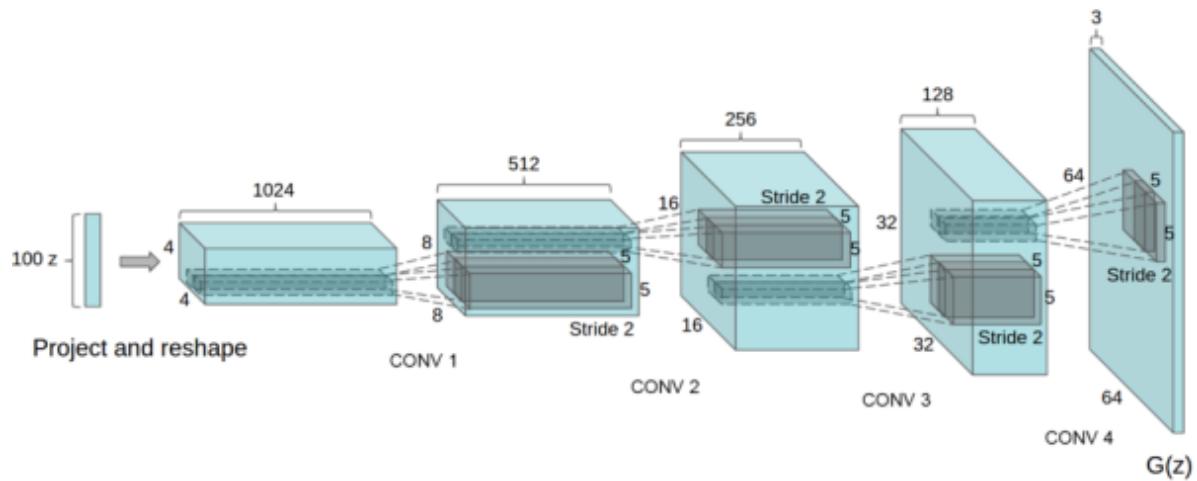


Figure 4.1: DCGAN generator architecture from the original paper, input is a 100 dimensional latent vector (noise, unit-gaussian) and output is a 64x64 image.

Another important thing is to avoid using fully connected layers in both the networks as they lead to the unstable training and convergence issues in DCGANs. Another challenge with deeper convolutional generators was the vanishing gradient problem, which was due to the poor initializations of the network parameters. This problem was addressed by using Batch Normalization layers (*Ioffe & Szegedy, 2015*), that enabled smooth gradient flow in the deeper networks. One other suggested change was to use *LeakyReLU* activations instead of the plain *ReLU*, within the discriminator networks.

Following is a list of some common architecture guidelines for training stable DCGANs as stated in the original paper:

- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
- Use Batch Normalization in both the generator and the discriminator networks.
- Remove all the fully connected hidden layers from the deeper convolutional GAN architectures.
- Use *ReLU* activation in the generator network for all layers except for the output layer, which uses *Tanh*.
- Use *LeakyReLU* activation in the discriminator network for all layers.

Keeping these things in mind, we should be able to develop more stable DCGAN models for our use cases. Next, we will learn about some guidelines for the stable training of DCGANs. (Figure 4.1 shows an example DCGAN architecture from the original paper).

1. Guidelines for stable training of DCGANs

In addition to the guidelines for designing stable architectures, the original DCGAN paper also shares some training guidelines as well. These guidelines may prove to be very helpful for the stable training of DCGANs.

Following is a list of important things to remember while training DCGAN based generative models:

- Scale the pixel values of the training images to range [-1, 1], the range of tanh activation.
- Train the generator and the discriminator networks with very small batches of data.
- It is advised to initialize the network weights with a gaussian distribution with a standard deviation of 0.02.
- Use *LeakyReLU* activations within the discriminator network with an alpha parameter value of 0.2 (the slope of the leak).
- It is recommended to train DCGANs with Adam optimizer with the following parameters: a small learning rate of about 0.0002, the momentum term beta_1 should be set to 0.5 for stable training.

After following these guidelines, we should now be able to develop and train stable DCGAN like models to generate good quality images. Let's check out some of the impressive results from DCGANs.

1. Impressive Image Generation Results from DCGANs

DCGAN paper was one of the first works to demonstrate the power of GAN framework for the task of image generation. After this paper was published, thousands of DCGAN based research papers followed, and out of which, mostly were related to its new applications while some of the papers were

focused on improving the DCGANs even further to get even more amazing outcomes. Further in this book, we will learn about some of those techniques that take DCGANs to another new level to generate photorealistic images. For now, let's focus on vanilla DCGAN results.

Let's now look at some of the impressive results of DCGAN, as shared in the original research paper.

1. MNIST handwritten digits

The images of handwritten digits generated by DCGAN, when trained with MNIST Handwritten Digits dataset, are slightly more plausible and realistic as compared to the vanilla GAN. The output images from DCGAN, look very similar to the groundtruth images as shown in the Figure 4.2.

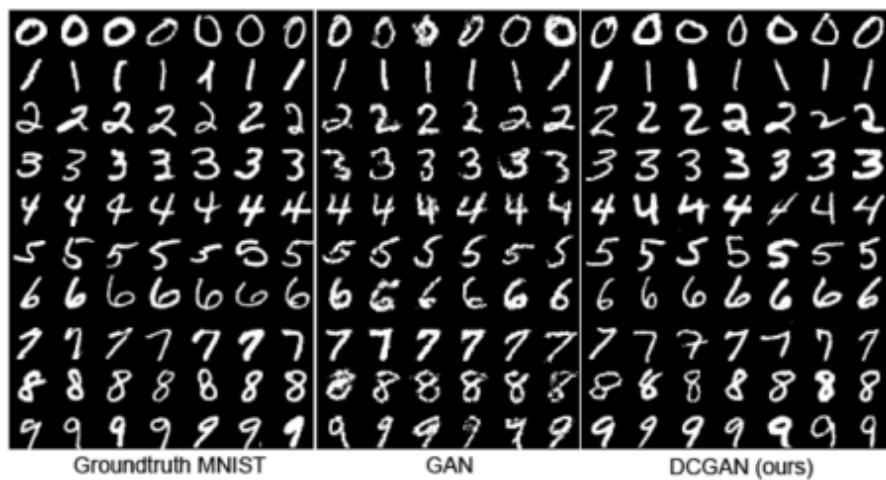


Figure 4.2: DCGAN results on MNIST Handwritten Digits dataset from the original paper

Let's now check out the results on more complicated datasets.

1. LSUN bedrooms dataset

DCGAN was also applied on LSUN bedrooms dataset. This dataset is composed of around 3M colorful bedroom images captured from different angles. DCGAN was able to generate quite realistic and diversified bedroom images in just 5 epochs of training.

Unlike the MNIST dataset, the LSUN dataset is composed of colorful images with wide variety of objects. It is a relatively very complex task to learn the representations of these images and then generate them. DCGAN was able to understand the important structures from the training dataset and thus, it was able to generate wide variety of similar images, as shown in the Figure 4.3.



Figure 4.3: DCGAN results on LSUN dataset taken from the original paper

As shown in the Figure 4.3, the generated images are really amazing. It shows the potential of DCGANs that they are capable of learning very complex representations. These results also prove the power of GANs as an effective generative learning framework for high dimensional data such as images.

Next, let's make the problem a little bit more complicated. This time, we will see if DCGANs are capable of generating realistic looking human faces.

1. Face generation

Another exciting application of DCGAN is to generate realistic human faces. Faces of the people that do not actually exist in the world. Yes! It is possible if a DCGAN is provided with sufficient amount of training data (face images). Figure 4.4 shows some human face images generated by the DCGAN model, as shown in the original paper. Though some of these faces were a little distorted, still it was a ground breaking result at that time (at the time

when DCGAN paper was published). It is very satisfying to see the advancements of deep learning, leading to such impressive results.



Figure 4.4: Human Faces generated by DCGAN, an image from the original paper

We just saw that DCGANs are capable of generating complex and realistic images, even human faces. This unveils the potential of DCGANs as a framework. Next, let's see what else can a GAN do apart from generating images.

1. Unsupervised Representations Learning

Another important thing about generative models is that they are capable of learning unsupervised representations (unsupervised features) from the training images. Unsupervised representation learning is a hot research topic as these features can be utilized to boost the accuracy of a supervised learning task where the labeled data is limited. GANs are very good at learning the underlying structure from the dataset and that's the reason they are able to generate plausible content. These features that are learned in an unsupervised manner, can be used to improve classification results in many supervised learning tasks.

Radford's paper also highlights the fact that DCGANs are also very good at learning unsupervised representations. They first trained a DCGAN model on Imagenet-1k and then used the learned representations to classify CIFAR-10 images, with just a linear classifier on the top of it. Interestingly, they were able to achieve state-of-the-art results among other unsupervised learning results, till that time.

We now have a good background about DCGANs and their capabilities. Let's now jump into some experiments and develop DCGANs of our own.

1. Experiments

In this skill, we are going to perform three exciting experiments with DCGAN architecture.

Following is the list of experiments:

- DCGAN for Fashion MNIST image generation
- DCGAN for Anime face generation
- DCGAN for Human face generation

The code samples shown in this skill can be downloaded from the following Github location:

<https://github.com/kartikgill/The-GAN-Book/tree/main/Skill-04>

Let's get started.

DCGAN FOR FASHION MNIST

Objective

In this experiment, we will implement and train a DCGAN like model on fashion MNIST dataset and verify its generative capabilities.

This experiment has the following steps:

- Importing Useful Libraries
- Download and Show dataset
- Data Normalization
- Define Generator
- Define Discriminator
- Define Combined model
- Define utility functions
- Training DCGAN on Fashion MNIST
- Results

Let's get started.

Step 1: Importing Useful Libraries

Open a Jupyter Notebook with python kernel and import useful python packages in a cell. In this experiment, we will ‘*numpy*’ for data manipulation, ‘*matplotlib*’ for plotting images and graphs, and *TensorFlow2* for implementing the model architecture.

Checkout the following snippet for importing libraries:

```
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
from tqdm import tqdm_notebook  
%matplotlib inline  
import tensorflow  
print(tensorflow.__version__)
```

Let's download the data now.

Step 2: Download and Show dataset

In this step, we will download the Fashion MNIST dataset and display a few samples to get a sense of the data. Fashion MNIST dataset can be loaded directly from the Keras datasets as shown in the following snippet:

```
from tensorflow.keras.datasets import fashion_mnist, mnist
(trainX, trainY), (testX, testY) = fashion_mnist.load_data()
print('Training data shapes: X=%s, y=%s' % (trainX.shape, trainY.shape))

print('Testing data shapes: X=%s, y=%s' % (testX.shape, testY.shape))
for k in range(9):
    plt.figure(figsize=(9, 6))
    for j in range(9):
        i = np.random.randint(0, 10000)
        plt.subplot(990 + 1 + j)
        plt.imshow(trainX[i], cmap='gray_r')
        plt.axis('off')
    #plt.title(trainY[i])
    plt.show()
```



Figure 4.5: Few examples from Fashion MNIST dataset

Figure 4.5 shows some examples from the Fashion MNIST dataset. The dataset has 60k images in the training set and 10k images in the test set, each with dimensions: 28 x 28. Each image also has a class label associated with it. It has 10 classes of fashion related objects.

We have data with us now. Let's prepare it for the model.

Step 3: Data Normalization

In this step, we will normalize the pixel values of all images by dividing them with 255. It will restrict the pixel values to [0, 1] range. We will also add a channel dimension to each image, to convert them into a 3D input as expected by CNN layers.

See the following code for data preparation:

```
trainX = [image/255 for image in trainX]
testX = [image/255 for image in testX]
trainX = np.reshape(trainX, (60000, 28, 28, 1))
testX = np.reshape(testX, (10000, 28, 28, 1))
print (trainX.shape, testX.shape, trainY.shape, testY.shape)
```

Our final dataset has the following shape:

Out [3]: (60000, 28, 28, 1) (10000, 28, 28, 1) (60000,) (10000,)

Our dataset is now ready. Let's work on the model architecture now.

Step 4: Define Generator

In this step, we will define the generator of our DCGAN. It will accept a noise vector of size 100 as input. We will first convert it into a 3-dimensional vector using a Reshape layer. Then, we will pass this 3D vector through multiple layers of *Conv2DTranspose*, *swish* activation and Batch Normalization. The final layer uses ‘sigmoid’ activation function to restrict the pixel values of the output image into [0, 1] range, similar to our training dataset.

The following python code implements the generator architecture:

```
random_input = tensorflow.keras.layers.Input(shape = 100)
x = tensorflow.keras.layers.Dense(128 * 5 * 5)(random_input)
x = tensorflow.keras.layers.Activation('swish')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Reshape((5, 5, 128))(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=128, kernel_size=(5,5))(x)
x = tensorflow.keras.layers.Activation('swish')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=128, kernel_size=(7,7))(x)
x = tensorflow.keras.layers.Activation('swish')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=128, kernel_size=(7,7))(x)
x = tensorflow.keras.layers.Activation('swish')(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=1, kernel_size=(8,8))(x)
generated_image = tensorflow.keras.layers.Activation('sigmoid')(x)
```

```
generator_network = tensorflow.keras.models.Model(inputs=random_input, outputs=generated_image)
generator_network.summary()
```

Following is the summary of the generator network. It has roughly 2.3M trainable parameters.

Out [4]: Model: "functional_1"

Layer (type)	Output Shape	Param #
--------------	--------------	---------

input_1 (InputLayer)	[(None, 100)]	0
----------------------	---------------	---

dense (Dense)	(None, 3200)	323200
---------------	--------------	--------

activation (Activation)	(None, 3200)	0
-------------------------	--------------	---

batch_normalization (BatchNo	(None, 3200)	12800
------------------------------	--------------	-------

reshape (Reshape)	(None, 5, 5, 128)	0
-------------------	-------------------	---

conv2d_transpose (Conv2DTran	(None, 9, 9, 128)	409728
------------------------------	-------------------	--------

activation_1 (Activation)	(None, 9, 9, 128)	0
---------------------------	-------------------	---

batch_normalization_1 (Batch	(None, 9, 9, 128)	512
------------------------------	-------------------	-----

conv2d_transpose_1 (Conv2DTr	(None, 15, 15, 128)	802944
------------------------------	---------------------	--------

activation_2 (Activation)	(None, 15, 15, 128)	0
---------------------------	---------------------	---

batch_normalization_2 (Batch	(None, 15, 15, 128)	512
------------------------------	---------------------	-----

```
conv2d_transpose_2 (Conv2DTr (None, 21, 21, 128) 802944
```

```
activation_3 (Activation) (None, 21, 21, 128) 0
```

```
conv2d_transpose_3 (Conv2DTr (None, 28, 28, 1) 8193
```

```
activation_4 (Activation) (None, 28, 28, 1) 0
```

```
=====
```

```
Total params: 2,360,833
```

```
Trainable params: 2,353,921
```

```
Non-trainable params: 6,912
```

Let's work on the discriminator model now.

Step 5: Define Discriminator

In this step, we will define the discriminator network. It accepts an image of size 28 x 28 x 1 as input and then passes it through multiple layers of *Conv2D*, *LeakyReLU* activation and Batch Normalization. At the end, we flatten the feature vector and pass it through a single neuron, and a '*sigmoid*' activation function to get the probabilistic output. This probability value indicates if a given input sample is real or fake.

The following python snippet implements the discriminator network:

```
image_input = tensorflow.keras.layers.Input(shape=(28, 28, 1))
x = tensorflow.keras.layers.Conv2D(filters=128, kernel_size=(3,3))
(image_input)
    x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
    x = tensorflow.keras.layers.Conv2D(filters=128, kernel_size=
(5,5), strides=2)(x)
    x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
    x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
```

```
x = tensorflow.keras.layers.Conv2D(filters=128, kernel_size=(5,5))(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2D(filters=64, kernel_size=(7,7))(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Flatten()(x)
x = tensorflow.keras.layers.Dense(1)(x)
real_vs_fake_output = tensorflow.keras.layers.Activation('sigmoid')(x)
discriminator_network = tensorflow.keras.models.Model(inputs=image_input, outputs=real_vs_fake_output)
discriminator_network.summary()
```

Following is the summary of the discriminator network. It has roughly 1.2M trainable parameters.

Out [5]: Model: "functional_3"

Layer (type) Output Shape Param #

=====

input_2 (InputLayer) [(None, 28, 28, 1)] 0

conv2d (Conv2D) (None, 26, 26, 128) 1280

leaky_re_lu (LeakyReLU) (None, 26, 26, 128) 0

conv2d_1 (Conv2D) (None, 11, 11, 128) 409728

leaky_re_lu_1 (LeakyReLU) (None, 11, 11, 128) 0

batch_normalization_3 (Batch (None, 11, 11, 128) 512

conv2d_2 (Conv2D) (None, 7, 7, 128) 409728

leaky_re_lu_2 (LeakyReLU) (None, 7, 7, 128) 0

batch_normalization_4 (Batch (None, 7, 7, 128) 512

conv2d_3 (Conv2D) (None, 1, 1, 64) 401472

leaky_re_lu_3 (LeakyReLU) (None, 1, 1, 64) 0

batch_normalization_5 (Batch (None, 1, 1, 64) 256

flatten (Flatten) (None, 64) 0

dense_1 (Dense) (None, 1) 65

activation_5 (Activation) (None, 1) 0

=====

Total params: 1,223,553

Trainable params: 1,222,913

Non-trainable params: 640

As the discriminator network works like a binary classifier here, we can compile it with ‘*binary_crossentropy*’ loss function. We will use Adam optimizer with a learning rate of 0.0001 and beta_1 value of 0.5 for updating the weights of the discriminator network.

The following python snippet compiles the discriminator network:

```
adam_optimizer = tensorflow.keras.optimizers.Adam(learning_rate=0.00  
01, beta_1=0.5)
```

```
discriminator_network.compile(loss='binary_crossentropy', optimizer=adam_optimizer, metrics=['accuracy'])
```

Let's define the combined model now.

Step 6: Define Combined model

In this step, we will define the combined DCGAN model. This combined model would be useful for updating the weights of the generator network. Whereas, the weights of the discriminator model can be updated separately. This is because, the gradient for the generator network, first flows through the discriminator network and then updates the weights of the generator network. During this step, the weights of the discriminator are kept frozen, and it only works as a feedback provider for the generator.

Following is a python snippet that defines the combined model:

```
discriminator_network.trainable=False  
g_output = generator_network(random_input)  
d_output = discriminator_network(g_output)  
dcgan_model = tensorflow.keras.models.Model(random_input, d_output)  
  
dcgan_model.summary()
```

Following is the summary of the combined model. It has roughly 2.3M trainable parameters that only belong to the generator network. Remember, we kept the discriminator network frozen for the combined model.

Out [6]: Model: "functional_5"

```
Layer (type) Output Shape Param #
```

```
=====
```

```
input_1 (InputLayer) [(None, 100)] 0
```

```
functional_1 (Functional) (None, 28, 28, 1) 2360833
```

```
functional_3 (Functional) (None, 1) 1223553
```

```
=====
Total params: 3,584,386
```

```
Trainable params: 2,353,921
```

```
Non-trainable params: 1,230,465
```

The combined model can also be compiled with similar parameters that we used for the discriminator network. The following python line compiles the combined DCGAN model:

```
dcgan_model.compile(loss='binary_crossentropy', optimizer=adam_optimizer)
```

DCGAN architecture is ready now. Let's work on the training piece now.

Step 7: Define utility functions

In this step, we will define some utility functions that will help us in creating data batches for our training iterations. We will basically define three utility functions here: one for getting a batch of random noise, second one for getting a batch of real samples from the dataset and third, for getting a batch of fake samples coming from the generator network. In addition to these, we will also define a function that can plot the images generated by the generator network. This can help us in checking model results at regular intervals during training.

Following is a python implementation of the utility functions:

```
indices = [i for i in range(0, len(trainX))]

def get_random_noise(batch_size, noise_size):
    random_values = np.random.randn(batch_size*noise_size)
    random_noise_batch = np.reshape(random_values, (batch_size, noise_size))

    return random_noise_batch

def get_fake_samples(generator_network, batch_size, noise_size):
    random_noise_batch = get_random_noise(batch_size, noise_size)
```

```

fake_samples = generator_network.predict_on_batch(random_noise_batch)

return fake_samples

def get_real_samples(batch_size):
    random_indices = np.random.choice(indices, size=batch_size)
    real_images = trainX[np.array(random_indices),:]
    return real_images

def show_generator_results(generator_network):
    for k in range(9):
        plt.figure(figsize=(7, 7))
        fake_samples = get_fake_samples(generator_network, 9, noise_size)
        for j in range(9):
            plt.subplot(990 + 1 + j)
            plt.imshow(fake_samples[j,:,:,-1], cmap='gray_r')
            plt.axis('off')
            #plt.title(trainY[i])
        plt.show()
    return

```

We are now all set to launch the model training.

Step 8: Training DCGAN on Fashion MNIST

In this step we will write the training iteration loop for our DCGAN model. We will train the model for 200 epochs, with a batch size of 100. Each epoch will have 500 steps for updating model weights. We will also print out some training logs at every 50th training step.

In a single training step, we first create a batch of mixed samples (real and fake). Real samples have a label of 1, and fake samples have label value of 0. This batch will be used for updating the weights of the discriminator network. Once done, we will freeze the weights of the discriminator network and create a batch of only fake samples to update the weights of the generator network. We will pass this batch of fake samples with inverted labels (label 1

for fake samples, to make the discriminator believe that these are real samples and calculate the loss value) into the combined DCGAN model and update the weights of the generator network only. Remember, the discriminator weights are frozen for the combined model. Here, the gradient will flow through the discriminator first, without changing its weights. Then, it will update the weights of the generator network.

Following python code implements the training loop:

```
epochs = 200
batch_size = 100
steps = 500
noise_size = 100

for i in range(0, epochs):
    if (i%10 == 0):
        show_generator_results(generator_network)
    for j in range(steps):
        fake_samples = get_fake_samples(generator_network, batch_size//2, noise_size)
        real_samples = get_real_samples(batch_size=batch_size//2)
        fake_y = np.zeros((batch_size//2, 1))
        real_y = np.ones((batch_size//2, 1))
        input_batch = np.vstack((fake_samples, real_samples))
        output_labels = np.vstack((fake_y, real_y))
        # Updating Discriminator weights
        discriminator_network.trainable=True
        loss_d = discriminator_network.train_on_batch(input_batch, output_labels)
        gan_input = get_random_noise(batch_size, noise_size)
        # Make the Discriminator believe that these are real samples and calculate loss to train the generator
        gan_output = np.ones((batch_size))
```

```

# Updating Generator weights
discriminator_network.trainable=False
loss_g = dcgan_model.train_on_batch(gan_input, gan_output)
if j%50 == 0:
    print ("Epoch:%.0f, Step:%.0f, D-Loss:%.3f, D-Acc:%.3f, G-
Loss:%.3f"%(i,j,loss_d[0],loss_d[1]*100,loss_g))

```

Following are the training logs, displaying the loss values of both models and accuracy of the discriminator network at every 50th training step.

Out [8]:Epoch:0, Step:0, D-Loss:0.553, D-Acc:76.000, G-Loss:0.716

```

Epoch:0, Step:50, D-Loss:0.075, D-Acc:97.000, G-Loss:4.552
Epoch:0, Step:100, D-Loss:0.090, D-Acc:99.000, G-Loss:8.371
Epoch:0, Step:150, D-Loss:0.099, D-Acc:96.000, G-Loss:2.220
Epoch:0, Step:200, D-Loss:0.084, D-Acc:97.000, G-Loss:3.003
Epoch:0, Step:250, D-Loss:0.126, D-Acc:95.000, G-Loss:3.306
Epoch:0, Step:300, D-Loss:0.128, D-Acc:97.000, G-Loss:2.593
Epoch:0, Step:350, D-Loss:0.123, D-Acc:98.000, G-Loss:5.523
Epoch:0, Step:400, D-Loss:0.058, D-Acc:99.000, G-Loss:2.241
Epoch:0, Step:450, D-Loss:0.083, D-Acc:99.000, G-Loss:4.615
Epoch:1, Step:0, D-Loss:0.084, D-Acc:99.000, G-Loss:3.719
...
...
...
Epoch:69, Step:200, D-Loss:0.257, D-Acc:89.000, G-Loss:2.006
Epoch:69, Step:250, D-Loss:0.154, D-Acc:95.000, G-Loss:0.660
Epoch:69, Step:300, D-Loss:0.170, D-Acc:91.000, G-Loss:3.566
Epoch:69, Step:350, D-Loss:0.139, D-Acc:93.000, G-Loss:0.662
Epoch:69, Step:400, D-Loss:0.243, D-Acc:89.000, G-Loss:6.700
Epoch:69, Step:450, D-Loss:0.210, D-Acc:90.000, G-Loss:5.906

```

Our DCGAN model is now trained. Let's check out the results.

Step 9: Results

In this step, we will check the results of our DCGAN model trained in previous step. Figure 4.6 shows some results at different epochs. We can see that model starts giving plausible results very soon. Results at epoch 10 may not look perfect but still they are very good. At epoch 50 and 70, the results look extremely realistic. It is difficult to tell if these results are model generated or the actual images. This proves the capability of DCGANs, for generating high quality images.

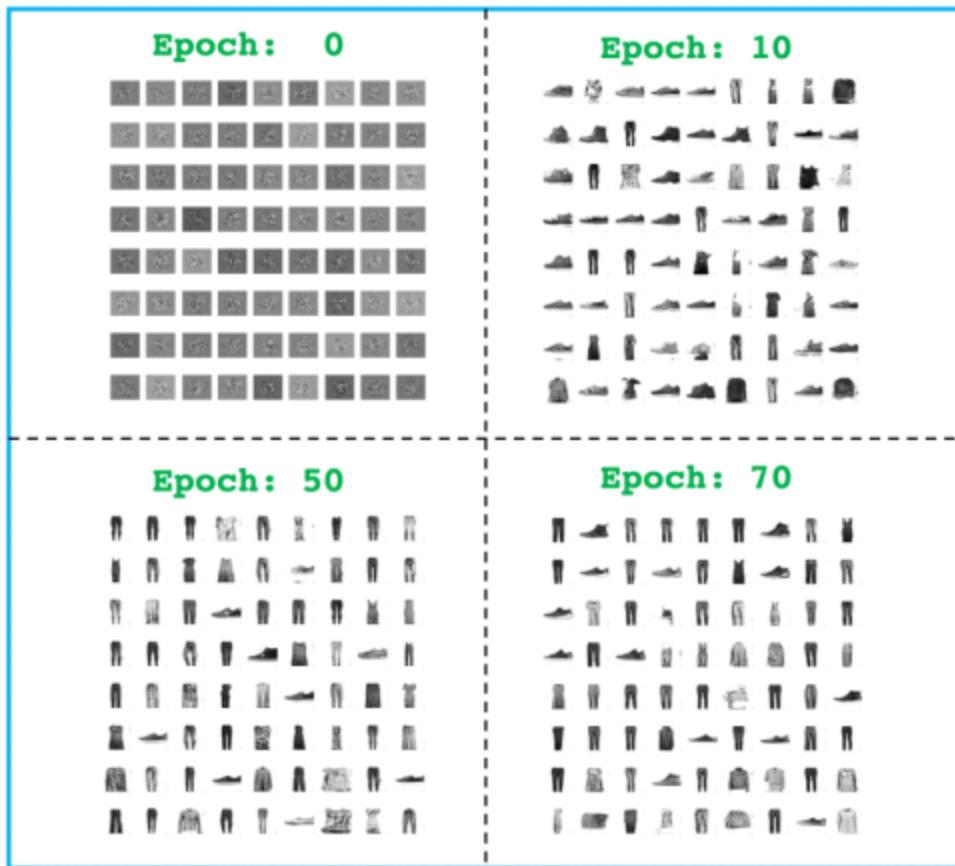


Figure 4.6: DCGAN results on the Fashion MNIST dataset

In the next experiments, we will try to generate little more complicated images using a similar DCGAN architecture. Let's go.

DCGAN FOR ANIME FACE GENERATION

Objective

In this experiment, we will train a DCGAN on Anime Faces dataset and display its generative capabilities on the color images.

This experiment has the following steps:

- Importing Useful Libraries
- Download and Show Anime Faces dataset
- Data Preparation
- Define Generator
- Define Discriminator
- Define Combined model
- Define utility functions
- Training DCGAN on Anime Faces
- Results

Let's get started.

Step 1: Importing Useful Libraries

Open a Jupyter Notebook with python kernel and import useful python packages in a cell. In this experiment, we will ‘*numpy*’ for data manipulation, ‘*matplotlib*’ for plotting images and graphs, and *TensorFlow2* for implementing the model architecture.

Checkout the following snippet for importing libraries:

```
import pandas as pd  
import numpy as np  
import cv2  
import glob  
import matplotlib.pyplot as plt  
from tqdm import tqdm_notebook  
%matplotlib inline  
import tensorflow
```

```
print(tensorflow.__version__)
```

Let's download the dataset now.

Step 2: Download and Show Anime Faces dataset

In this experiment, we will use Anime faces dataset. This dataset is publicly available on Kaggle (<https://www.kaggle.com/datasets/splcher/animefacedataset>). Once we have downloaded the data, we can use the following python snippet for extracting it and displaying few samples:

```
!unzip /content/gdrive/MyDrive/GAN_datasets/anime.zip -d /content/
files = glob.glob('/content/images/*.jpg')
for k in range(9):
    plt.figure(figsize=(11, 11))
    for j in range(9):
        f = np.random.choice(files)
        anime_img = cv2.imread(f)
        anime_img = cv2.cvtColor(anime_img, cv2.COLOR_BGR2RGB)
        anime_img = cv2.resize(anime_img, (56, 56))
        plt.subplot(990 + 1 + j)
        plt.imshow(anime_img)
        plt.axis('off')
    plt.show()
```

Figure 4.7 shows some samples from the Anime Faces dataset. We can see that the dataset has colorful images of Anime faces with varying styles of hairs, eyes and different hair colors. This dataset has roughly 63k high-quality anime face images with varying resolutions.



Figure 4.7: An overview of the Anime Faces dataset

Let's now prepare the data.

Step 3: Data Preparation

In this step, we will prepare the Anime faces dataset for model. It is important to bring all input images into a similar resolution for the modelling purpose. In this experiment, we will resize each image to a resolution of 56 x 56. Finally, we will normalize the pixel values of each image and restrict them into a range of [-1, 1]. We can achieve this by subtracting and dividing each image by 127.5 (as image pixel values range from 0 to 255).

The following python snippet prepares the data:

```
from tqdm import tqdm_notebook  
images = []  
for file in tqdm_notebook(files):  
    anime_img = cv2.imread(file)  
    anime_img = cv2.cvtColor(anime_img, cv2.COLOR_BGR2RGB)  
    anime_img = cv2.resize(anime_img, (56, 56))  
    images.append((anime_img-127.5)/127.5)  
images = np.array(images)  
print (images.shape)
```

Following is the shape of the final prepared dataset:

Out [3]: (63565, 56, 56, 3)

Our dataset is now ready. Let's work on the model architecture.

Step 4: Define Generator

In this step, we will define the generator of our DCGAN. It will accept a noise vector of size 100 as input. We will first convert it into a 3-dimensional vector using a Reshape layer. Then, we will pass this 3D vector through multiple layers of *Conv2DTranspose*, *ReLU* activation and Batch Normalization. The final layer uses '*tanh*' activation function to restrict the pixel values of the output image into range [-1, 1], similar to our training dataset. The final output should have dimensions of 56 x 56 x 3.

The following python code implements the generator architecture:

```
random_input = tensorflow.keras.layers.Input(shape = 100)
x = tensorflow.keras.layers.Dense(128 * 5 * 5)(random_input)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Reshape((5, 5, 128))(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=128, kernel_size=(3,3), strides=(2,2))(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=128, kernel_size=(3,3), strides=(2,2))(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=128, kernel_size=(3,3), strides=(2,2))(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
```

```
x = tensorflow.keras.layers.Conv2DTranspose(filters=128, kernel_size=(4,4))(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=128, kernel_size=(5,5))(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=128, kernel_size=(5,5))(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.Conv2D(filters=3, kernel_size=(3,3))(x)
generated_image = tensorflow.keras.layers.Activation('tanh')(x)
generator_network = tensorflow.keras.models.Model(inputs=random_input, outputs=generated_image)
generator_network.summary()
```

Following is a summary of the generator network. It roughly has 1.8M trainable parameters:

Out [4]: Model: "functional_1"

Layer (type) Output Shape Param #

=====

input_1 (InputLayer) [(None, 100)] 0

dense (Dense) (None, 3200) 323200

activation (Activation) (None, 3200) 0

batch_normalization (BatchNo (None, 3200) 12800

reshape (Reshape) (None, 5, 5, 128) 0

conv2d_transpose (Conv2DTran (None, 11, 11, 128) 147584

activation_1 (Activation) (None, 11, 11, 128) 0

batch_normalization_1 (Batch (None, 11, 11, 128) 512

conv2d_transpose_1 (Conv2DTr (None, 23, 23, 128) 147584

activation_2 (Activation) (None, 23, 23, 128) 0

batch_normalization_2 (Batch (None, 23, 23, 128) 512

conv2d_transpose_2 (Conv2DTr (None, 47, 47, 128) 147584

activation_3 (Activation) (None, 47, 47, 128) 0

batch_normalization_3 (Batch (None, 47, 47, 128) 512

conv2d_transpose_3 (Conv2DTr (None, 50, 50, 128) 262272

activation_4 (Activation) (None, 50, 50, 128) 0

batch_normalization_4 (Batch (None, 50, 50, 128) 512

conv2d_transpose_4 (Conv2DTr (None, 54, 54, 128) 409728

activation_5 (Activation) (None, 54, 54, 128) 0

```
batch_normalization_5 (Batch (None, 54, 54, 128) 512
```

```
conv2d_transpose_5 (Conv2DTr (None, 58, 58, 128) 409728
```

```
activation_6 (Activation) (None, 58, 58, 128) 0
```

```
conv2d (Conv2D) (None, 56, 56, 3) 3459
```

```
activation_7 (Activation) (None, 56, 56, 3) 0
```

```
=====
```

Total params: 1,866,499

Trainable params: 1,858,819

Non-trainable params: 7,680

Let's define the discriminator network now.

Step 5: Define Discriminator

In this step, we will define the discriminator network. It accepts a colorful image of size 56 x 56 x 3, as input and then passes it through multiple layers of *Conv2D*, *LeakyReLU* activation and Batch Normalization. At the end, we flatten the feature vector and pass it through a single neuron, and a '*sigmoid*' activation function to get the probabilistic output. This probability value indicates if a given input sample is real or fake.

The following python snippet implements the discriminator network:

```
image_input = tensorflow.keras.layers.Input(shape=(56, 56, 3))
x = tensorflow.keras.layers.Conv2D(filters=128, kernel_size=(3,3))
(image_input)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
```

```
x = tensorflow.keras.layers.Conv2D(filters=64, kernel_size=(5,5), strides=2)(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2D(filters=64, kernel_size=(5,5), strides=2)(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2D(filters=64, kernel_size=(5,5))(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2D(filters=64, kernel_size=(5,5))(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Flatten()(x)
x = tensorflow.keras.layers.Dense(1)(x)
real_vs_fake_output = tensorflow.keras.layers.Activation('sigmoid')(x)
discriminator_network = tensorflow.keras.models.Model(inputs=image_input, outputs=real_vs_fake_output)
discriminator_network.summary()
```

Following is a summary of the discriminator network. It has roughly 500k trainable parameters.

Out [5]: Model: "functional_5"

Layer (type)	Output Shape	Param #
--------------	--------------	---------

input_3 (InputLayer)	[(None, 56, 56, 3)]	0
----------------------	---------------------	---

conv2d_6 (Conv2D)	(None, 54, 54, 128)	3584
-------------------	---------------------	------

leaky_re_lu_5 (LeakyReLU) (None, 54, 54, 128) 0

conv2d_7 (Conv2D) (None, 25, 25, 64) 204864

leaky_re_lu_6 (LeakyReLU) (None, 25, 25, 64) 0

batch_normalization_10 (BatchNormalization) (None, 25, 25, 64) 256

conv2d_8 (Conv2D) (None, 11, 11, 64) 102464

leaky_re_lu_7 (LeakyReLU) (None, 11, 11, 64) 0

batch_normalization_11 (BatchNormalization) (None, 11, 11, 64) 256

conv2d_9 (Conv2D) (None, 7, 7, 64) 102464

leaky_re_lu_8 (LeakyReLU) (None, 7, 7, 64) 0

batch_normalization_12 (BatchNormalization) (None, 7, 7, 64) 256

conv2d_10 (Conv2D) (None, 3, 3, 64) 102464

leaky_re_lu_9 (LeakyReLU) (None, 3, 3, 64) 0

batch_normalization_13 (BatchNormalization) (None, 3, 3, 64) 256

flatten_1 (Flatten) (None, 576) 0

dense_2 (Dense) (None, 1) 577

```
activation_9 (Activation) (None, 1) 0
```

```
=====
Total params: 517,441
```

```
Trainable params: 516,929
```

```
Non-trainable params: 512
```

We will compile the discriminator model with ‘*binary_crossentropy*’ loss. We will utilize Adam optimizer with a learning rate of 0.0001 and beta_1 value of 0.5 for updating the weights of the discriminator model.

```
adam_optimizer = tensorflow.keras.optimizers.Adam(learning_rate=0.0001, beta_1=0.5)
```

```
discriminator_network.compile(loss='binary_crossentropy', optimizer=adam_optimizer, metrics=['accuracy'])
```

Let’s define the combined model now.

Step 6: Define Combined model

In this step, we will define the combined DCGAN model. This combined model would be useful for updating the weights of the generator network. Whereas, the weights of the discriminator model can be updated separately. This is because, the gradient for the generator network, first flows through the discriminator network and then updates the weights of the generator network. During this step, the weights of the discriminator network are kept frozen, and it only works as a feedback provider for the generator network.

Following is a python snippet that defines the combined model:

```
discriminator_network.trainable=False
```

```
g_output = generator_network(random_input)
```

```
d_output = discriminator_network(g_output)
```

```
dcgan_model = tensorflow.keras.models.Model(random_input, d_output)
```

```
dcgan_model.summary()
```

Following is the summary of the combined DCGAN model. It has roughly 1.8M trainable parameters.

Out [6]: Model: "functional_7"

Layer (type) Output Shape Param #

input_1 (InputLayer) [(None, 100)] 0

functional_1 (Functional) (None, 56, 56, 3) 1866499

functional_5 (Functional) (None, 1) 517441

Total params: 2,383,940

Trainable params: 1,858,819

Non-trainable params: 525,121

We can compile the combined model with same parameters that we used for the discriminator network earlier.

```
dcgan_model.compile(loss='binary_crossentropy', optimizer=adam_optimizer)
```

DCGAN is now ready. Let's work on the training piece now.

Step 7: Define utility functions

In this step, we will define some utility functions that will help us in creating data batches for our training iterations. We will basically define three utility functions here: one for getting a batch of random noise, second one for getting a batch of real samples from the dataset and third, for getting a batch of fake samples coming from the generator network. In addition to these, we will also define a function that can plot the images generated by the generator network. This can help us in checking model results at regular intervals during training.

Following is a python implementation of the utility functions:

```
indices = [i for i in range(0, len(images))]

def get_random_noise(batch_size, noise_size):
    random_values = np.random.randn(batch_size*noise_size)
    random_noise_batch = np.reshape(random_values, (batch_size, noise_size))

return random_noise_batch

def get_fake_samples(generator_network, batch_size, noise_size):
    random_noise_batch = get_random_noise(batch_size, noise_size)
    fake_samples = generator_network.predict_on_batch(random_noise_batch)

return fake_samples

def get_real_samples(batch_size):
    random_indices = np.random.choice(indices, size=batch_size)
    real_images = images[np.array(random_indices),:]
    return real_images

def show_generator_results(generator_network):
    for k in range(9):
        plt.figure(figsize=(11, 11))
        fake_samples = get_fake_samples(generator_network, 9, noise_size)
        fake_samples = (fake_samples+1.0)/2.0
        for j in range(9):
            plt.subplot(990 + 1 + j)
            plt.imshow(fake_samples[j])
            plt.axis('off')
            #plt.title(trainY[i])
        plt.show()
    return
```

Let's train the model now.

Step 8: Training DCGAN on Anime Faces

In this step we will write the training iteration loop for our DCGAN model. We will train the model for 100 epochs, with a batch size of 100. Each epoch will have 500 steps for updating model weights. We will also print out some training logs at every 50th training step.

In a single training step, we first create a batch of mixed samples (real and fake). Real samples have a label of 1, and fake samples have label value of 0. This batch will be used for updating the weights of the discriminator network. Once done, we will freeze the weights of the discriminator network and create a batch of only fake samples to update the weights of the generator network. We will pass this batch of fake samples with inverted labels (label 1 for fake samples, to make the discriminator believe that these are real samples and calculate the loss value) into the combined DCGAN model and update the weights of the generator network only. Remember, the discriminator weights are frozen for the combined model. Here, the gradient will flow through the discriminator first, without changing its weights. Then, it will update the weights of the generator network.

Following python code implements the training loop:

```
epochs = 100
batch_size = 100
steps = 500
noise_size = 100
for i in range(0, epochs):
    if (i%10 == 0):
        show_generator_results(generator_network)
    for j in range(steps):
        fake_samples = get_fake_samples(generator_network, batch_size//2, noise_size)
        real_samples = get_real_samples(batch_size=batch_size//2)
        fake_y = np.zeros((batch_size//2, 1))
        real_y = np.ones((batch_size//2, 1))
```

```

input_batch = np.vstack((fake_samples, real_samples))
output_labels = np.vstack((fake_y, real_y))
# Updating Discriminator weights
discriminator_network.trainable=True
loss_d = discriminator_network.train_on_batch(input_batch, output_label
s)
gan_input = get_random_noise(batch_size, noise_size)
# Make the Discriminator believe that these are real samples and calculate
loss to train the generator
gan_output = np.ones((batch_size))
# Updating Generator weights
discriminator_network.trainable=False
loss_g = dcgan_model.train_on_batch(gan_input, gan_output)
if j%50 == 0:
    print ("Epoch:%.0f, Step:%.0f, D-Loss:%.3f, D-Acc:%.3f, G-
Loss:%.3f"%(i,j,loss_d[0],loss_d[1]*100,loss_g))

```

Following are the training logs. We can monitor the loss values of both models here along with the accuracy of the discriminator network.

Out [8]: Epoch:0, Step:0, D-Loss:0.816, D-Acc:37.000, G-Loss:0.714

```

Epoch:0, Step:50, D-Loss:0.021, D-Acc:100.000, G-Loss:4.888
Epoch:0, Step:100, D-Loss:0.087, D-Acc:96.000, G-Loss:4.839
Epoch:0, Step:150, D-Loss:0.116, D-Acc:97.000, G-Loss:8.926
Epoch:0, Step:200, D-Loss:0.108, D-Acc:96.000, G-Loss:2.893
Epoch:0, Step:250, D-Loss:0.021, D-Acc:100.000, G-Loss:9.395
Epoch:0, Step:300, D-Loss:0.173, D-Acc:93.000, G-Loss:3.882
Epoch:0, Step:350, D-Loss:0.165, D-Acc:94.000, G-Loss:4.459
Epoch:0, Step:400, D-Loss:0.112, D-Acc:95.000, G-Loss:2.763
Epoch:0, Step:450, D-Loss:0.117, D-Acc:94.000, G-Loss:5.952
Epoch:1, Step:0, D-Loss:0.065, D-Acc:98.000, G-Loss:5.853

```

Epoch:1, Step:50, D-Loss:0.011, D-Acc:100.000, G-Loss:7.191

...

...

...

Epoch:99, Step:100, D-Loss:0.064, D-Acc:97.000, G-Loss:1.396

Epoch:99, Step:150, D-Loss:0.065, D-Acc:97.000, G-Loss:0.963

Epoch:99, Step:200, D-Loss:0.020, D-Acc:100.000, G-Loss:2.552

Epoch:99, Step:250, D-Loss:0.087, D-Acc:98.000, G-Loss:0.464

Epoch:99, Step:300, D-Loss:0.322, D-Acc:92.000, G-Loss:0.132

Epoch:99, Step:350, D-Loss:0.042, D-Acc:99.000, G-Loss:1.726

Epoch:99, Step:400, D-Loss:0.051, D-Acc:98.000, G-Loss:0.023

Epoch:99, Step:450, D-Loss:0.000, D-Acc:100.000, G-Loss:5.847

The training is complete now. Let's check out the results.

Step 9: Results

Figure 4.8 shows some results of our DCGAN model at different epochs. We can see that the model starts giving plausible results very soon. Results at epoch 10 may not look perfect but still they are very good. At epoch 50, the results start getting little more realistic. At epoch 100, the results look extremely realistic. It is difficult to tell if these results are model generated or the actual images. This proves the capability of DCGANs, for generating high quality colorful images. Using this trained model, we now have the capability to generate infinite number of anime faces.

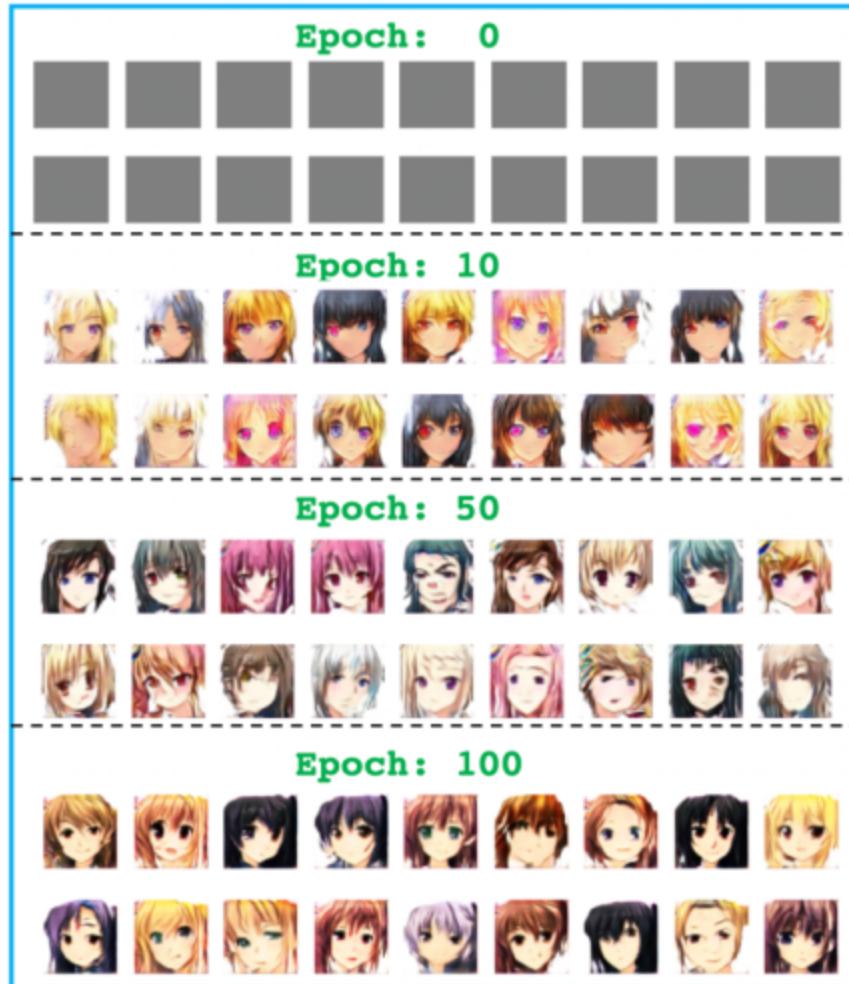


Figure 4.8: DCGAN results for Anime Faces Generation task

Let's jump into a little more complicated experiment now where we will use DCGAN to generate human faces. Let's go.

DCGAN FOR HUMAN FACE GENERATION

Objective

In this experiment, we will train a DCGAN on Human Faces dataset and verify its generative capabilities on colorful and complex images.

This experiment has the following steps:

- Importing Useful Libraries
- Download and Show Human Faces dataset
- Data Preparation
- Define Generator
- Define Discriminator
- Define Combined model
- Define utility functions
- Training DCGAN on Human Faces
- Results

Let's get started.

Step 1: Importing Useful Libraries

Open a Jupyter Notebook with python kernel and import useful python packages in a cell. In this experiment, we will ‘*numpy*’ for data manipulation, ‘*matplotlib*’ for plotting images and graphs, and *TensorFlow2* for implementing the model architecture.

Checkout the following snippet for importing libraries:

```
import pandas as pd  
import numpy as np  
import cv2  
import glob  
import matplotlib.pyplot as plt  
from tqdm import tqdm_notebook  
%matplotlib inline  
import tensorflow
```

```
print(tensorflow.__version__)
```

Let's download the dataset now.

Step 2: Download and Show Human Faces dataset

For this experiment, we need a dataset with few thousand images of human face. We can download any publicly available dataset from Kaggle to do this experiment. The following python snippet displays few samples from the dataset:

```
!unzip /content/gdrive/MyDrive/GAN_datasets/celeb_faces.zip -d /content/
```

```
files = glob.glob('/content/100k/100k/*.jpg')
for k in range(9):
    plt.figure(figsize=(15, 15))
    for j in range(9):
        f = np.random.choice(files)
        celeb_img = cv2.imread(f)
        celeb_img = cv2.cvtColor(celeb_img, cv2.COLOR_BGR2RGB)
        plt.subplot(990 + 1 + j)
        plt.imshow(celeb_img)
        plt.axis('off')
#        plt.title(trainY[i])
    plt.show()
```

Figure 4.9 shows some samples from the dataset.

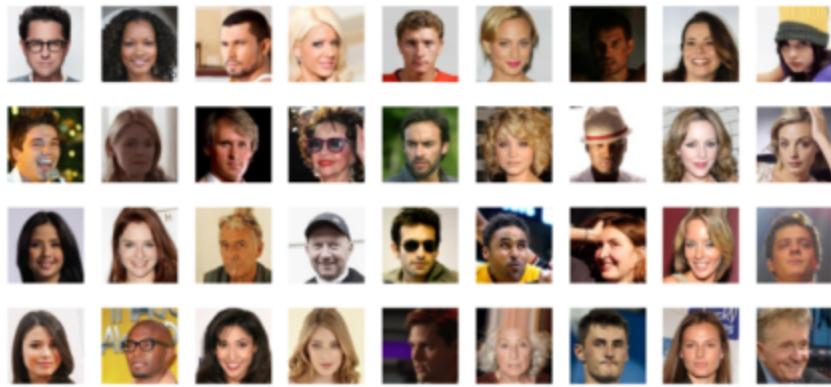


Figure 4.9: Some sample human face images from the dataset

Let's now prepare the data.

Step 3: Data Preparation

In this step, we will prepare the Human faces dataset for the model. It is important to bring all input images into a similar resolution for the modelling purpose. In this experiment, we will resize each image to a resolution of 56 x 56. Finally, we will normalize the pixel values of each image and restrict them into a range of [-1, 1]. We can achieve this by subtracting and dividing each image by 127.5 (as image pixel values range from 0 to 255).

The following python snippet prepares the data:

```
from tqdm import tqdm_notebook  
images = []  
for file in tqdm_notebook(files):  
    celeb_img = cv2.imread(file)  
    celeb_img = cv2.cvtColor(celeb_img, cv2.COLOR_BGR2RGB)  
    celeb_img = cv2.resize(celeb_img, (56, 56))  
    images.append((celeb_img-127.5)/127.5)  
images = np.array(images)  
print (images.shape)
```

Our data is now ready. Let's define the model architecture now.

Step 4: Define Generator

In this step, we will define the generator of our DCGAN. It will accept a noise vector of size 100 as input. We will first convert it into a 3-dimensional vector using a Reshape layer. Then, we will pass this 3D vector through multiple layers of *Conv2DTranspose*, *ReLU* activation and Batch Normalization. The final layer uses ‘*tanh*’ activation function to restrict the pixel values of the output image into range [-1, 1], similar to our training dataset. The final output should have dimensions of 56 x 56 x 3.

The following python code implements the generator architecture:

```
random_input = tensorflow.keras.layers.Input(shape = 100)
x = tensorflow.keras.layers.Dense(64 * 7 * 7)(random_input)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Reshape((7, 7, 64))(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=128, kernel_size=(5,5), strides=(2,2), padding='same')(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=128, kernel_size=(5,5), strides=(2,2), padding='same')(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=128, kernel_size=(5,5), strides=(2,2), padding='same')(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=128, kernel_size=(5,5), padding='same')(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=128, kernel_size=(5,5), padding='same')(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.Conv2D(filters=3, kernel_size=(3,3), padding='same')(x)
generated_image = tensorflow.keras.layers.Activation('tanh')(x)
```

```
generator_network = tensorflow.keras.models.Model(inputs=random_input, outputs=generated_image)
generator_network.summary()
```

Following is the summary of the generator network. It has roughly 1.7M trainable parameters.

Out [4]: Model: "functional_1"

Layer (type)	Output Shape	Param #
--------------	--------------	---------

input_1 (InputLayer)	[(None, 100)]	0
----------------------	---------------	---

dense (Dense)	(None, 3136)	316736
---------------	--------------	--------

activation (Activation)	(None, 3136)	0
-------------------------	--------------	---

batch_normalization (BatchNo	(None, 3136)	12544
------------------------------	--------------	-------

reshape (Reshape)	(None, 7, 7, 64)	0
-------------------	------------------	---

conv2d_transpose (Conv2DTran	(None, 14, 14, 128)	204928
------------------------------	---------------------	--------

activation_1 (Activation)	(None, 14, 14, 128)	0
---------------------------	---------------------	---

batch_normalization_1 (Batch	(None, 14, 14, 128)	512
------------------------------	---------------------	-----

conv2d_transpose_1 (Conv2DTr	(None, 28, 28, 128)	409728
------------------------------	---------------------	--------

activation_2 (Activation)	(None, 28, 28, 128)	0
---------------------------	---------------------	---

batch_normalization_2 (Batch	(None, 28, 28, 128)	512
------------------------------	---------------------	-----

```
conv2d_transpose_2 (Conv2DTr (None, 56, 56, 128) 409728
```

```
activation_3 (Activation) (None, 56, 56, 128) 0
```

```
batch_normalization_3 (Batch (None, 56, 56, 128) 512
```

```
conv2d_transpose_3 (Conv2DTr (None, 56, 56, 128) 409728
```

```
activation_4 (Activation) (None, 56, 56, 128) 0
```

```
conv2d (Conv2D) (None, 56, 56, 3) 3459
```

```
activation_5 (Activation) (None, 56, 56, 3) 0
```

```
=====
```

```
Total params: 1,768,387
```

```
Trainable params: 1,761,347
```

```
Non-trainable params: 7,040
```

Let's define the discriminator network now.

Step 5: Define Discriminator

In this step, we will define the discriminator network. It accepts a colorful image of size 56 x 56 x 3, as input and then passes it through multiple layers of *Conv2D*, *LeakyReLU* activation and Batch Normalization. At the end, we flatten the feature vector and pass it through a single neuron, and a '*sigmoid*' activation function to get the probabilistic output. This probability value indicates if a given input sample is real or fake.

The following python snippet implements the discriminator network:

```

image_input = tensorflow.keras.layers.Input(shape=(56, 56, 3))
x = tensorflow.keras.layers.Conv2D(filters=128, kernel_size=
(5,5), strides=2, padding='same')(image_input)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.Conv2D(filters=64, kernel_size=
(5,5), strides=2, padding='same')(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2D(filters=128, kernel_size=
(5,5), strides=2, padding='same')(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2D(filters=64, kernel_size=
(5,5), padding='same')(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2D(filters=128, kernel_size=(5,5))(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2D(filters=64, kernel_size=(3,3))(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Flatten()(x)
x = tensorflow.keras.layers.Dense(1)(x)
real_vs_fake_output = tensorflow.keras.layers.Activation('sigmoid')(x)
discriminator_network = tensorflow.keras.models.Model(inputs=image_i
nput, outputs=real_vs_fake_output)

discriminator_network.summary()

```

Following is the summary of the discriminator network. It roughly has 900k trainable parameters.

Out [5]: Model: "functional_3"

Layer (type) Output Shape Param #

input_2 (InputLayer) [(None, 56, 56, 3)] 0

conv2d_1 (Conv2D) (None, 28, 28, 128) 9728

leaky_re_lu (LeakyReLU) (None, 28, 28, 128) 0

conv2d_2 (Conv2D) (None, 14, 14, 64) 204864

leaky_re_lu_1 (LeakyReLU) (None, 14, 14, 64) 0

batch_normalization_4 (Batch (None, 14, 14, 64) 256

conv2d_3 (Conv2D) (None, 7, 7, 128) 204928

leaky_re_lu_2 (LeakyReLU) (None, 7, 7, 128) 0

batch_normalization_5 (Batch (None, 7, 7, 128) 512

conv2d_4 (Conv2D) (None, 7, 7, 64) 204864

leaky_re_lu_3 (LeakyReLU) (None, 7, 7, 64) 0

batch_normalization_6 (Batch (None, 7, 7, 64) 256

conv2d_5 (Conv2D) (None, 3, 3, 128) 204928

```
leaky_re_lu_4 (LeakyReLU) (None, 3, 3, 128) 0
```

```
batch_normalization_7 (Batch (None, 3, 3, 128) 512
```

```
conv2d_6 (Conv2D) (None, 1, 1, 64) 73792
```

```
leaky_re_lu_5 (LeakyReLU) (None, 1, 1, 64) 0
```

```
batch_normalization_8 (Batch (None, 1, 1, 64) 256
```

```
flatten (Flatten) (None, 64) 0
```

```
dense_1 (Dense) (None, 1) 65
```

```
activation_6 (Activation) (None, 1) 0
```

```
=====
```

```
Total params: 904,961
```

```
Trainable params: 904,065
```

```
Non-trainable params: 896
```

We will compile the discriminator network with ‘*binary_crossentropy*’ loss.
We will utilize Adam optimizer with a learning rate of 0.0001 and beta_1 value of 0.5 for updating the weights of the discriminator model.

```
adam_optimizer = tensorflow.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5)
```

```
discriminator_network.compile(loss='binary_crossentropy', optimizer=adam_optimizer, metrics=['accuracy'])
```

Let’s define the combined DCGAN model.

Step 6: Define Combined model

In this step, we will define the combined DCGAN model. This combined model would be useful for updating the weights of the generator network. Whereas, the weights of the discriminator model can be updated separately. This is because, the gradient for the generator network, first flows through the discriminator network and then updates the weights of the generator network. During this step, the weights of the discriminator network are kept frozen, and it only works as a feedback provider for the generator network.

Following is a python snippet that defines the combined model:

```
discriminator_network.trainable=False  
g_output = generator_network(random_input)  
d_output = discriminator_network(g_output)  
dcgan_model = tensorflow.keras.models.Model(random_input, d_output)  
  
dcgan_model.summary()
```

Following is the summary of the combined DCGAN model. It has roughly 1.7M trainable parameters.

Out [6]: Model: "functional_5"

```
Layer (type) Output Shape Param #
```

```
=====
```

```
input_1 (InputLayer) [(None, 100)] 0
```

```
functional_1 (Functional) (None, 56, 56, 3) 1768387
```

```
functional_3 (Functional) (None, 1) 904961
```

```
=====
```

```
Total params: 2,673,348
```

```
Trainable params: 1,761,347
```

```
Non-trainable params: 912,001
```

We can compile the combined model similar to the discriminator model.

```
dcgan_model.compile(loss='binary_crossentropy', optimizer=adam_optimizer)
```

Our DCGAN is all set now. Let's work on the training part.

Step 7: Define utility functions

In this step, we will define some utility functions that will help us in creating data batches for our training iterations. We will basically define three utility functions here: one for getting a batch of random noise, second one for getting a batch of real samples from the dataset and third, for getting a batch of fake samples coming from the generator network. In addition to these, we will also define a function that can plot the images generated by the generator network. This can help us in checking model results at regular intervals during training.

Following is a python implementation of the utility functions:

```
def get_random_noise(batch_size, noise_size):
    random_values = np.random.randn(batch_size*noise_size)
    random_noise_batch = np.reshape(random_values, (batch_size, noise_size))
    return random_noise_batch

def get_fake_samples(generator_network, batch_size, noise_size):
    random_noise_batch = get_random_noise(batch_size, noise_size)
    fake_samples = generator_network.predict_on_batch(random_noise_batch)
    return fake_samples

def get_real_samples(batch_size):
    random_files = np.random.choice(files, size=batch_size)
    images = []
    for file in random_files:
        celeb_img = cv2.imread(file)
        celeb_img = cv2.cvtColor(celeb_img, cv2.COLOR_BGR2RGB)
```

```

celeb_img = cv2.resize(celeb_img, (56, 56))
images.append((celeb_img-127.5)/127.5)
real_images = np.array(images)
return real_images

def show_generator_results(generator_network):
    for k in range(9):
        plt.figure(figsize=(11, 11))
        fake_samples = get_fake_samples(generator_network, 9, noise_size)
        fake_samples = (fake_samples+1.0)/2.0
        for j in range(9):
            plt.subplot(990 + 1 + j)
            plt.imshow(fake_samples[j])
            plt.axis('off')
            #plt.title(trainY[i])
        plt.show()
    return

```

We are now all set to launch the training.

Step 8: Training DCGAN on Human Faces

In this step we will write the training iteration loop for our DCGAN model. We will train the model for 200 epochs, with a batch size of 100. Each epoch will have 500 steps for updating model weights. We will also print out some training logs at every 50th training step.

In a single training step, we first create a batch of mixed samples (real and fake). Real samples have a label of 1, and fake samples have label value of 0. This batch will be used for updating the weights of the discriminator network. Once done, we will freeze the weights of the discriminator network and create a batch of only fake samples to update the weights of the generator network. We will pass this batch of fake samples with inverted labels (label 1 for fake samples, to make the discriminator believe that these are real samples and calculate the loss value) into the combined DCGAN model and

update the weights of the generator network only. Remember, the discriminator weights are frozen for the combined model. Here, the gradient will flow through the discriminator first, without changing its weights. Then, it will update the weights of the generator network.

Following python code implements the training loop:

```
epochs = 200
batch_size = 100
steps = 500
noise_size = 100
for i in range(0, epochs):
    if (i%5 == 0):
        show_generator_results(generator_network)
        generator_network.save('/content/gdrive/MyDrive/GAN_datasets/celeb_model_'+ str(i))
    for j in range(steps):
        fake_samples = get_fake_samples(generator_network, batch_size//2, noise_size)
        real_samples = get_real_samples(batch_size=batch_size//2)
        fake_y = np.zeros((batch_size//2, 1))
        real_y = np.ones((batch_size//2, 1))
        input_batch = np.vstack((fake_samples, real_samples))
        output_labels = np.vstack((fake_y, real_y))
        # Updating Discriminator weights
        discriminator_network.trainable=True
        loss_d = discriminator_network.train_on_batch(input_batch, output_labels)
        gan_input = get_random_noise(batch_size, noise_size)
        # Make the Discriminator believe that these are real samples and calculate loss to train the generator
        gan_output = np.ones((batch_size))
```

```

# Updating Generator weights
discriminator_network.trainable=False
loss_g = dcgan_model.train_on_batch(gan_input, gan_output)
if j%50 == 0:
    print ("Epoch:%.0f, Step:%.0f, D-Loss:%.3f, D-Acc:%.3f, G-
Loss:%.3f"%(i,j,loss_d[0],loss_d[1]*100,loss_g))

```

Following are the training logs. We can use them to monitor the loss values of both models during training.

Out [8]: Epoch:0, Step:0, D-Loss:1.143, D-Acc:20.000, G-Loss:0.665

```

Epoch:0, Step:50, D-Loss:0.044, D-Acc:99.000, G-Loss:10.416
Epoch:0, Step:100, D-Loss:0.279, D-Acc:87.000, G-Loss:3.813
Epoch:0, Step:150, D-Loss:0.248, D-Acc:90.000, G-Loss:4.373
Epoch:0, Step:200, D-Loss:0.120, D-Acc:96.000, G-Loss:4.974
Epoch:0, Step:250, D-Loss:0.297, D-Acc:92.000, G-Loss:2.804
Epoch:0, Step:300, D-Loss:0.172, D-Acc:92.000, G-Loss:8.522
Epoch:0, Step:350, D-Loss:0.191, D-Acc:91.000, G-Loss:2.756
Epoch:0, Step:400, D-Loss:0.634, D-Acc:72.000, G-Loss:5.606
Epoch:0, Step:450, D-Loss:0.200, D-Acc:93.000, G-Loss:1.924
...
...
...
...
Epoch:36, Step:250, D-Loss:0.784, D-Acc:51.000, G-Loss:0.624
Epoch:36, Step:300, D-Loss:0.612, D-Acc:68.000, G-Loss:1.069
Epoch:36, Step:350, D-Loss:0.473, D-Acc:84.000, G-Loss:1.211
Epoch:36, Step:400, D-Loss:0.500, D-Acc:79.000, G-Loss:0.331
Epoch:36, Step:450, D-Loss:0.701, D-Acc:56.000, G-Loss:0.625
Epoch:37, Step:0, D-Loss:0.538, D-Acc:69.000, G-Loss:1.308
Epoch:37, Step:50, D-Loss:0.539, D-Acc:71.000, G-Loss:1.728

```

Epoch:37, Step:100, D-Loss:0.590, D-Acc:64.000, G-Loss:1.104

Let's check out the results now.

Step 9: Results

Figure 4.10 shows some results of our DCGAN model at different epochs. We can see that the model starts giving plausible results very soon. Results at epoch 10 may not look perfect but still they are very good. At epoch 25, the results start getting little more realistic. At epoch 37, the results look a little more realistic. This proves the capability of DCGANs, for generating human face images. Using this trained model, we now have the capability to generate infinite number of human faces.

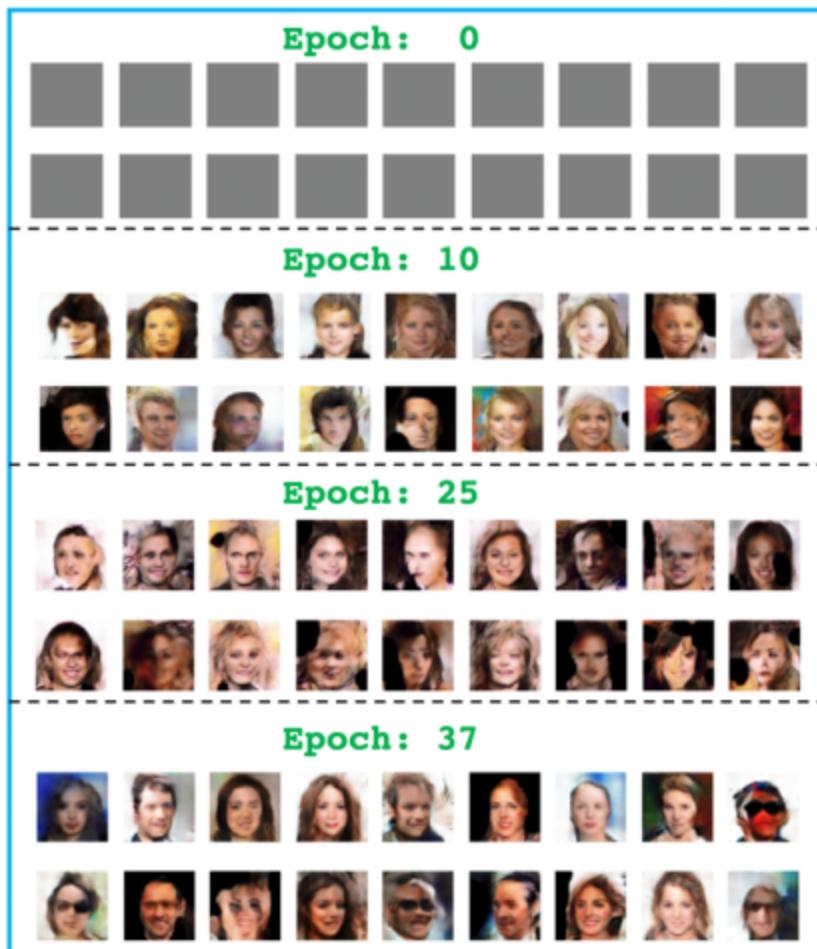


Figure 4.10: DCGAN results for generating human face images.

Our experiments on DCGAN end here. We now have a good background of DCGANs and we should now be able to easily apply them in new application

areas.

1. Summary

In this skill, we have seen how GANs can be scaled using CNNs. Specifically, we have learned about DCGANs. We discussed some of the guidelines for developing and training stable CNN based GANs. We also saw some impressive results of DCGAN based models. After reading this skill, we should be able to answer the following questions about DCGANs:

- How to develop deep Convnets based GANs to generate high quality image data?
- What are the best practices to follow while developing DCGANs so that the training is stable?
- Can unsupervised representations learned through a DCGAN model be used to boost the classification accuracy in supervised tasks?

In the next skill, we will learn about the latent space.

1. you can try next...

- Implement and train a DCGAN based model for a new application area.
- Find out ways to improve DCGAN results even further.
- Learn about different ways of measuring the quality of generated data.

End of Skill-4

SKILL 4 (PART-II)

Into the Latent Space

Generative Adversarial Network, or GAN, is a framework for training generative models that can generate content such as text, images, music and so on. The generator part of the GAN architecture learns to map the latent space (or a noise vector) into realistic content (such as images - in case of image generation task). This latent space is generally noise in case of GANs and does not have any meaning to it.

In the task of image generation, we have seen that a new random latent space always generates a different image. This makes us wonder: Can we actually control what content will be generated, by controlling this latent space somehow? One simple idea to test this, is to interpolate a few points of the latent space and check the generator results. In this skill, we will do some experiments with the latent space of a trained generator model and see if we can find some useful insights from there.

In this skill, we will experiment with already trained GAN generators for image generation task. This skill covers the following main topics:

- Making sense of the Latent Space
- Into the Latent Space: Anime Model
- Into the Latent Space: Human Face Model
- Summary

Let's get started.

1. Making sense of the Latent Space

GAN is a great framework for generating limitless amounts of quality content. The basic idea behind the working of the generator network, is to accept some latent space (could be noise) as input and map it into some meaningful content (such as images) such that the generated content belongs to a distribution that is very similar to a given target distribution (the

distribution of the realistic content used for training). The discriminator network here, helps the generator to bridge the gap between these two distributions (the generated and the realistic one).

Previously, we saw that passing some random noise as input generates an entirely new image every time. This makes us think about the following:

- Is there a way to control the types of generated images?
- What happens to the output, if only a few points in the latent space are changed?
- How changing the latent space slightly, actually affects the generated output?

To answer all such questions, we will pick up a pre-trained generator network and do some experiments with the latent space. We will experiment with two previously trained models here: The Anime Face Generation Model and the Human Face Generation Model. Let's jump into the experiments.

1. Into the Latent Space: Anime Model

In one of the experiments from the previous skill, we trained a DCGAN based generator that was able to generate Anime face images. In this experiment, we will use that pre-trained generator model to understand the latent space. Specifically, we will play around with the latent space inputs, and see how it changes the generated output. We will also check if there is a way to control the generated output, even a little bit, using small changes to the latent space of an already trained generator model. Let's quickly get into the experiment now.

The code samples shown in this skill can be downloaded from the following Github location:

<https://github.com/kartikgill/The-GAN-Book/tree/main/Skill-04>

As a first step, we will launch a new jupyter notebook with python kernel and import some dependencies. (Note that versions of dependencies are similar to those that were used for the training of the network).

```
import pandas as pd  
import numpy as np  
import cv2  
import matplotlib.pyplot as plt  
from tqdm import tqdm_notebook  
%matplotlib inline  
import tensorflow  
print(tensorflow.__version__)
```

Let's now load the pre-trained generator model weights. This was a TensorFlow (TF) based model, so we will utilize the '*load_model*' method here. The following python snippet, loads the pre-trained weights:

```
generator_network = tensorflow.keras.models.load_model('/content/gdrive/MyDrive/GAN_datasets/anime_saved_model')
```

As we already know that our anime generator model accepts the random noise as input, let's create a small function that can generate a batch of inputs with required dimensions filled with random noise.

The following python function creates a batch of noise vectors:

```
def get_random_noise(batch_size, noise_size):  
    random_values = np.random.randn(batch_size*noise_size)  
    random_noise_batch = np.reshape(random_values, (batch_size, noise_size))  
    return random_noise_batch
```

This anime model was trained with input vectors of size 100 x 100 having random noise. Let's first generate a bunch of anime face images using random noise vectors and then, we will study the latent space. The following code snippet first creates 100 latent space inputs of required size and then, generates the generator outputs for them. We also plot first 81 generated images for our experiment purposes (see Figure 4(II).1).

```

noise_size = 100
random_noise_batches = get_random_noise(100, noise_size)
fake_samples = generator_network.predict_on_batch(random_noise_batches)
fake_samples = (fake_samples+1.0)/2.0
counter = 0
for k in range(9):
    plt.figure(figsize=(13, 13))
    for j in range(9):
        plt.subplot(990 + 1 + j)
        plt.imshow(fake_samples[counter])
        counter += 1
    plt.axis('off')
# plt.title(trainY[i])
plt.show()

```

Figure 4(II).1 shows the generated anime faces for our randomly created vectors. We can see that the generated images have different hair colors and other features. We will focus our experiment on the hair colors here as we can visually see good variety for hair colors in the generated images.

Let's pick the indices of the anime faces with similar hair colors and categorize them into the categories: Yellow, Black, Blue and Red respectively. Note, that for simplicity we are considering the purple hairs as Blue, and dark brown (or maroon) hairs into the red category. Figure 4(II).1, has ordered images so it would be easy to pick out their indices.



Figure 4(II).1: Generated Fake Anime face images using GANs

Following are the lists of indices having same color categories as discussed:

yellow_hairs = [3, 9, 16, 17, 19, 22, 29, 33, 34]

black_hairs = [1, 4, 7, 8, 11, 18, 20, 28, 35]

blue_hairs = [0, 12, 23, 25, 38, 57]

red_hairs = [6, 13, 15, 30, 39, 40, 42, 43]

Our first assumption is that the latent space (or the input noise vectors) of anime faces with same hair colors would have some similarities. There should be a part of the latent space that actually controls the color of the hairs in a generated anime face image. To verify our assumption, we will calculate the averages of latent spaces belonging to the outputs of same hair colors and check if they also generate a face with similar hair colors.

Let's first calculate the average latent space for each color category:

```
yellow_avg_latent_code = np.mean(random_noise_batches[np.array(yellow_hairs),:], axis=0)
```

```
black_avg_latent_code = np.mean(random_noise_batches[np.array(black_hairs),:], axis=0)
```

```

blue_avg_latent_code = np.mean(random_noise_batches[np.array(blue_hairs),:], axis=0)
red_avg_latent_code = np.mean(random_noise_batches[np.array(red_hairs),:], axis=0)

```

Now that we have average latent space of all four-color categories. We can quickly generate images with these average vectors and verify if it actually makes any sense.

Let's first generate an anime face image from the average latent space of yellow color category. See the following python snippet:

```

fake_sample = generator_network.predict(np.array([yellow_avg_latent_code]))
fake_sample = (fake_sample[0]+1.0)/2.0
plt.figure(figsize=(3,3))
plt.imshow(fake_sample)
plt.show()

```

Interestingly, the generated output also has yellow color hairs (see Figure 4(II).2). This is aligned to our assumptions, but let's double check for other categories as well to be completely sure about this.

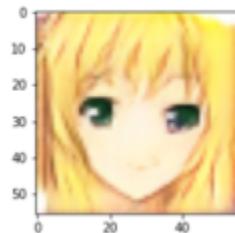


Figure 4(II).2: Output with average latent space of yellow hair category

Let's move to the second color category. Here, we are taking the average latent vector of hair color category 'black', and checking the generated output image.

```

fake_sample = generator_network.predict(np.array([black_avg_latent_code]))
fake_sample = (fake_sample[0]+1.0)/2.0

```

```

plt.figure(figsize=(3,3))
plt.imshow(fake_sample)
plt.show()

```

We can see the output image (Fig. 4(II).3) and it clearly has black hairs. So, this is again aligned with our assumptions.

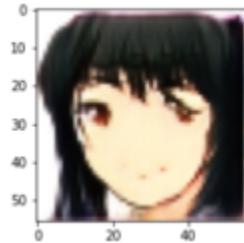


Figure 4(II).3: Output with average latent space of black hair category

We just saw that the average latent vector actually has the properties of similar hair colors. This means that the latent spaces are not entirely random and specific parts of the latent spaces actually represent some important properties related to the generated content. Only issue here is that, there is no way to technically find what part of latent space control which aspects of the generated content.

Figure 4(II).4 shows the outputs for hair color category blue and red as well.

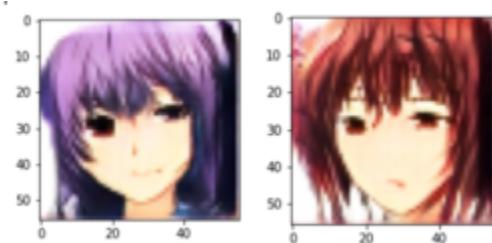


Figure 4(II).4: Output with average latent space of blue and red hair categories

Our first hypothesis was correct. We can now say that some specific parts of latent space control some specific aspects of the generated content. Now, let's

try to make use of this information to do some more fun experiments.

In our second experiment, we will try to check if these latent vectors also follow vector arithmetic. One simple way to verify this is by creating an average latent vector to two color categories. As per our understanding until now, the average latent space of two different color categories should represent a new color differing from both of these color categories.

Let's create an average latent space vector of red and yellow hair category vectors:

```
# RED plus YELLOW  
red_yellow_avg = (red_avg_latent_code + yellow_avg_latent_code)/2.0  
fake_sample = generator_network.predict(np.array([red_yellow_avg]))  
fake_sample = (fake_sample[0]+1.0)/2.0  
plt.figure(figsize=(3,3))  
plt.imshow(fake_sample)  
plt.show()
```

Figure 4(II).5, shows the result of averaging the latent space of two-color categories. Here we averaged the vectors of 'yellow' and 'red' color categories and we can clearly see that the generated image actually has a new hair color that is somewhat in between (slightly orange), and it is different from both the original color categories. So yes, we can say that the latent space supports arithmetic vector operations.

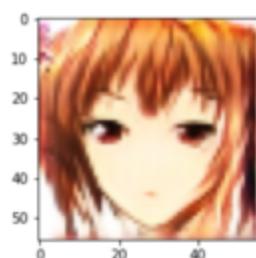


Figure 4(II).5: Output with average latent space of red and yellow hair category

To make sure that we didn't get this result by luck, let's try with the average of blue and yellow color categories as well.

```

# BLUE plus YELLOW
blue_yellow_avg = (blue_avg_latent_code + yellow_avg_latent_code)/2.

0
fake_sample = generator_network.predict(np.array([blue_yellow_avg]))
fake_sample = (fake_sample[0]+1.0)/2.0
plt.figure(figsize=(3,3))
plt.imshow(fake_sample)
plt.show()

```

Check out the Figure 4(II).6, we can clearly see the hair color of the generated image is grayish, which is much different from the original color categories of ‘blue’ and ‘yellow’. This proves our point that vector arithmetic works over latent spaces of GAN based generator models.

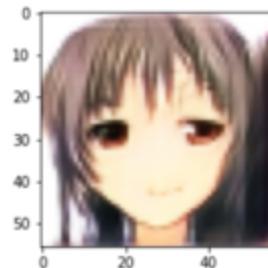


Figure 4(II).6: Output with average latent space of blue and yellow hair category

We have just verified that there is some structure to the latent space and the specific parts of the latent space actually control some key properties of the generated outputs. However, there is no programmatic way to identify that. In our experiments, we found these similarities in the outputs by looking at some of the generated content manually. We have also verified that the latent spaces support vector arithmetic. Now, let’s play some more with the latent space here.

We will do one more interesting experiment now. Here, we will take latent space of ‘yellow’ hair color category and try to update different locations of this space with some random noise. Note that we will only update a few points in the latent space at a time to make sure, it doesn’t

change entirely from the original category (in this example we are updating only 10 locations). Now, let's see what happens if we update 10 points of the latent space with random noise at different locations.

The following python code updates the values at some locations in the latent space:

```
# Lets play with Yellow
for iter in range(2):
    plt.figure(figsize=(15,15))
    for i in range(9):
        code = yellow_avg_latent_code
        mask = np.random.randn(10)
        code[i*10: i*10 + 10] = mask
        fake_sample = generator_network.predict(np.array([code]))
        fake_sample = (fake_sample[0]+1.0)/2.0
        plt.subplot(990 + 1 + i)
        plt.axis('off')
        plt.imshow(fake_sample)
    plt.show()
```

Figure 4(II).7 shows 18 anime face images that were generated through random updates into the latent space of 'yellow' hair color category latent space. We can see that even small updates to the existing latent space, change the generated output greatly. The output image shows that even updating small number of points (such as 10), can change the hair colors and face features drastically.

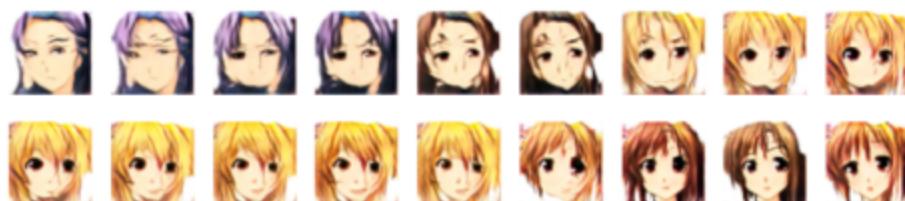


Figure 4(II).7: Output after updating few points in the latent space of yellow hair category

We have just gathered some interesting facts about the latent space. We can now say that specific parts of latent space control some specific features of the generated images. We can get slight modifications into the outputs by incorporating small changes into the latent space but there is no logical way to find out the exact positions to be updated to get the desired outputs.

We have made these remarks on the basis of a single model (anime face model), but does this idea generalize for all GAN based models? To prove that let's play around with the latent space of one more pre-trained model. Let's quickly jump into that experiment.

1. Into the Latent Space: Human Face Model

In this experiment as well, we will try very similar things to the previous experiment. But this time, we will take a pre-trained generator model that was trained to generate human faces. Let's now explore the latent space of this model.

As a first step, create a jupyter notebook and import all the useful dependencies as shown in the following snippet. All the dependency versions are similar to the ones that were used for training this model.

```
import pandas as pd  
import numpy as np  
import cv2  
import matplotlib.pyplot as plt  
from tqdm import tqdm_notebook  
%matplotlib inline  
import tensorflow  
print(tensorflow.__version__)
```

In one of the previous skills, we trained a human face generator model. Here, we will load the pre-trained weights of that generator model. As this generator model was TF based, we can use '*load_model*' method for loading the pre-trained weights.

Check the following snippet.

```
generator_network = tensorflow.keras.models.load_model('/content/gdrive/MyDrive/GAN_datasets/celeb_model_new')
```

Similar to the previous experiment, we are defining a function that can return a batch of noise inputs of desired dimensions.

```
def get_random_noise(batch_size, noise_size):  
    random_values = np.random.randn(batch_size*noise_size)  
    random_noise_batch = np.reshape(random_values, (batch_size, noise_size))  
    return random_noise_batch
```

Our model is now loaded and ready for generating images. Let's create a batch of random noise with 100 x 100 dimensions, as expected by the model's input layer. We will create a big batch of 100 input instances. We will then pass this input batch to the generator model and generate results. We are plotting the first 81 results here for review purpose.

Check the following snippet for generating human faces.

```
noise_size = 100  
random_noise_batches = get_random_noise(100, noise_size)  
fake_samples = generator_network.predict_on_batch(random_noise_batches)  
fake_samples = (fake_samples+1.0)/2.0  
counter = 0  
for k in range(11):  
    plt.figure(figsize=(15, 15))  
for j in range(9):  
    plt.subplot(990 + 1 + j)  
    plt.imshow(fake_samples[counter])
```

```

counter += 1
plt.axis('off')
#plt.title(trainY[i])
plt.show()

```

If everything goes well, we should get the model output very similar to the one in Figure 4(II).8. We can see that our model is able to generate human like faces, the output is not entirely realistic due to the low capacity of the model and small training dataset, but still it looks somewhat like human faces.



Figure 4(II).8: Output of human face generator model trained on celebrity face images

If we look closely at these outputs, we can see that some faces have black hair colors while some of them have golden hair colors irrespective of the gender. Let's choose a few indices where the model has generated women with black hairs and a few indices where the model has generated women images with golden hairs. Using the latent vectors (or noise vectors), let's create average latent vectors for both black and the golden hair color faces.

The following python code calculates the average latent vectors for black and golden hair category:

```
black_hair_women = [0, 1, 3, 27, 28, 31, 48, 67, 76, 81, 88]  
golden_hair_women = [2, 14, 61, 65, 75, 84]  
black_hair_avg_latent_code = np.mean(random_noise_batches[np.array(black_hair_women),:], axis=0)  
golden_hair_avg_latent_code = np.mean(random_noise_batches[np.array(golden_hair_women),:], axis=0)
```

First let's generate a face image using the average latent vector of black hair color women. As per our understanding, the model should also generate the face of a women with black hairs.

See the following code:

```
fake_sample = generator_network.predict(np.array([black_hair_avg_latent_code]))  
fake_sample = (fake_sample[0]+1.0)/2.0  
plt.figure(figsize=(3,3))  
plt.imshow(fake_sample)  
plt.show()
```

If we look at the outputs, we can see that model actually generates the face of a women with black hairs. Similarly, if we try generating the output with the average latent vector of golden hair color, it also produces the face of a women with golden hair color as output. See Figure 4(II).9 for reference. (black hair women face on left, golden hair women face on right side).

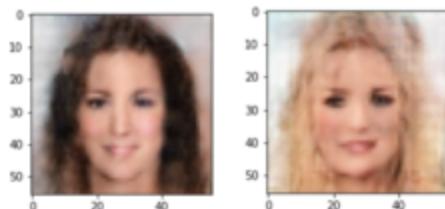


Figure 4(II).9: Output after generating faces with average latent space of black and golden color haired women faces from left to right respectively.

We just saw that the average latent space of black and golden hair color women faces, actually generates the face of a women with same hair color. Let's check out, what happens if we take the average latent space of black hairs and golden hair color faces.

```
# Mixture  
black_golden_avg = (black_hair_avg_latent_code + golden_hair_avg_latent_code)/2.0  
fake_sample = generator_network.predict(np.array([black_golden_avg]))  
  
fake_sample = (fake_sample[0]+1.0)/2.0  
plt.figure(figsize=(3,3))  
plt.imshow(fake_sample)  
plt.show()
```

As per our understanding and findings from the previous experiment, this new average latent space should also generate a face of a women, and the hair color should be somewhat in between the golden and the black color. If we look at the output shown in Figure 4(II).10, our generator model actually generates the face of a women with dark golden hairs. This again proves our point.

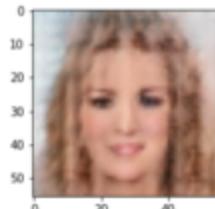


Figure 4(II).10: Generating human face with average latent space of black and golden haired faces

Now, let's move to our final experiment where we take the latent space of a random face from the previously shown outputs and play around with the latent space. In this experiment, we will take one latent space and try to replace 20 points of this latent space with random noise. We will also repeat

this experiment by putting some other values instead of random noise in those 20 locations.

Check out the following snippet that implements the point replacement with random noise:

```
for iter in range(5):
    plt.figure(figsize=(15,15))
    mask = np.random.randn(20)
    for i in range(9):
        code = latent_code.copy()
        code[i*10: i*10 + 20] = mask
        fake_sample = generator_network.predict(np.array([code]))
        fake_sample = (fake_sample[0]+1.0)/2.0
        plt.subplot(990 + 1 + i)
        plt.axis('off')
        plt.imshow(fake_sample)
    plt.show()
```

If we look closely at this output shown in Figure 4(II).11, changing only a small number of points in the latent space has a big impact on the features of the generated face images. Secondly, perturbing the latent space with different techniques (such as zeroes, ones, random noise and so on), also changes the results greatly. That's why the outputs in same columns are not similar, even though the location of the changes applied is same.

Some of these small perturbations also change the orientation of the faces slightly (such as left look vs right look). Another interesting observation is that: at some of the places (in 7th column: 2nd and 4th outputs), the face actually has changed the gender and looks like a face of a man.

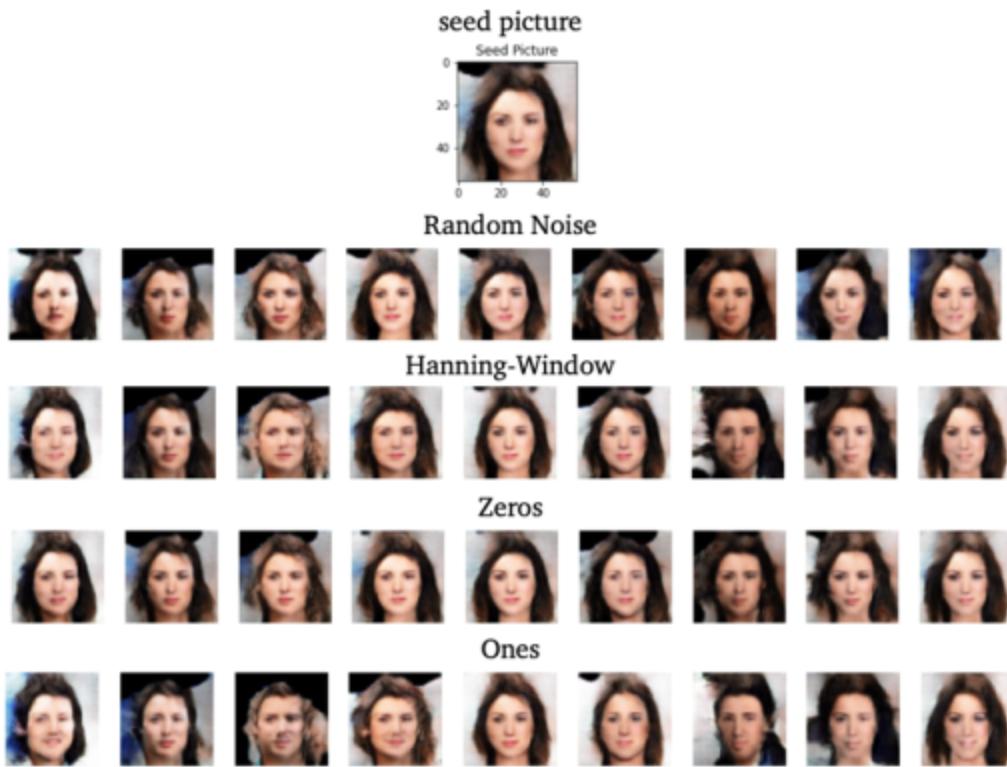


Figure 4(II).11: Generating human faces with random perturbations to the latent space of seed input image. Outputs shown in this image represent the following from top- 1. See image used for latent space, 2 – latent space perturbation with random noise, 3 – latent space perturbation with hanning window, 4 – latent space perturbation with zeros, 5 – latent space perturbation with ones.

This experiment shows how complicated the latent space of a GAN based model is. Very small changes to the latent space can result in significant changes to the output. With this our little experimental study of exploring the latent space ends here.

1. Summary

In this skill, we performed some fun experiments with two pre-trained generator models: Anime face model and human face model. We explored the latent space and found some interesting facts about it. Although, the latent space is a mystery but it definitely has some structure to it, there definitely are specific parts of the latent space that control some specific aspects of the type of output generated. I hope this was a useful study for the readers.

After completing this skill, we should be able to answer the following questions about the latent space:

- Is latent space completely random or does it make sense with respect to the output of generator model?
- How can we control the generated content of a generator with the help of controlled latent inputs?
- Is regular arithmetic on the latent space meaningful in order to control certain aspects of the generator results?
- How much control does knowing the latent space of a certain generated output can provide?

In this next skill, we will learn about some ways to make GANs more stable for training.

End of Skill-4(II)

SKILL 5

Towards Stable GANs

Generative Adversarial Networks, or GANs for short, are quite difficult to train in practice (We have seen that in skill 3 as well). This is due to the nature of GAN training where two networks compete with each other in a zero-sum game. This means that one model improves at the cost of degradation in the performance of the other model. This contest makes the training process very unstable.

Although there is no hard and fast rule for creating a stable GAN architecture, many researchers have studied the behavior of GAN trainings under different conditions over the decade. These studies have led to a set of guidelines that improve the overall stability of the training process. It has taken a lot of research and experimentation to find out these settings that stabilize the training process. In this skill, we will be going through these best-practices and guidelines.

After reading this skill, we should feel confident about implementing and training stable GANs. Let's understand some of the best-practices of training stable GANs.

1. Best Practices for Training Stable GANs

Most of the best-practices highlighted in this skill are based on a 2016 paper by Salimans, Tim, et al. titled "*Improved techniques for training GANs*." Some techniques are taken from other studies/research-papers and very few of these are based on my own experience from training multiple GAN architectures while writing this book.

Following is a consolidated list of some of the best-practices that are important to understand and remember:

- Convolutional Layers

- Regularization
- Activations
- Optimizers
- Feature Matching
- Image Scaling and Mini Batch
- Label Smoothing
- Weight Initialization

Now, let's go through each of these best-practices one-by-one.

1.1 Convolutional Layers

Convolutional Neural Network, or CNN, is an obvious choice when it comes to image related use cases. CNNs are specifically designed for localized feature learning from Images. So, if we want to generate images from a GAN based framework, we should develop the GAN with CNN layers. As discussed earlier as well, GAN training is highly unstable but fortunately we can make it work, if we choose the network architectures and hyperparameters carefully.

Radford et. al. (2015) presented a CNN based GAN architecture, known as DCGAN, that implements some of the best practices. DCGAN resulted in quite stable training and performed really well, as seen in Skill 4 as well.

Following is a list of some common guidelines for designing CNN based stable GAN architectures:

- For down-sampling and up-sampling purposes, utilize the strided convolutions instead of pooling layers.
- Use normalization techniques such as batch-normalization in both discriminator and generator networks.
- Don't keep too many fully-connected hidden layers in any of the networks.
- Use *LeakyReLU* activation in discriminator network and *ReLU* in the generator network.

These guidelines can help us in designing more stable CNN based GAN architectures. Next, let's learn about regularization.

1.2 Regularization

In machine learning, regularization techniques are used to prevent model from overfitting and underfitting on training data and as a result it performs well on test data. Batch normalization is a commonly used regularization technique while training neural networks. It makes the training of networks faster and stable by adding few extra layers (batch normalization layers) within the network.

Batch Normalization has significant impact in stabilizing the training of GANs as well. The batch normalization layers are usually added after the convolutional layers and before the activation layers (*ReLU* or *LeakyReLU*) in both generator and the discriminator networks.

1.3 Activations

Most of the times, when we design neural networks, we don't pay much attention to the activation functions and generally go with a simple activation function such as Rectified Linear Unit, or *ReLU*, within the intermediate layers. While the final layer activation depends upon the type of problem we are solving, for example – '*sigmoid*' is used for binary classification tasks.

Interestingly, in case of GANs the activation function also plays a critical role when it comes to the stability of training. Experiments in the DCGAN paper have shown that using a *LeakyReLU* activation function within all the layers of the discriminator network and *ReLU* activation function in the layers (except for the final layer) of the generator network, results in a stable DCGAN architecture. However, some other studies claim that using *LeakyReLU* in both networks, performs even better.

LeakyReLU is a slight variant of the *ReLU* activation and it allows some negative values to flow as well (whereas *ReLU* converts every negative input to zero). *LeakyReLU* also provides a parameter for negative slope and keeping it at the default value of 0.2 generally gives good results in case of GANs.

1.4 Optimizers

The basic optimization algorithm used for the training of neural networks is Stochastic Gradient Descent (or SGD). In practice, researchers have found few hacks to make the optimization faster and smooth. These new derivative optimization techniques are knowns as optimizers. Optimization algorithms also plays a key role in the stability of training of convolutional GANs.

Adaptive Moment Estimation optimizer, or *Adam* for short, is a well-known and frequently used variant of SGD. *Adam* optimizer is also recommended for training convolutional GANs with a learning rate of 0.0002 and the *beta_1* momentum value of 0.5 (whereas the default value of *beta_1* momentum is 0.9).

1.5 Feature Matching

In this approach, we modify the objective of GAN training a little bit. Instead of training the generator for maximizing output of discriminator, we train the generator to match the statistics of generated data with real data. In simpler words, we train the generator to match the features of real and generated images within an intermediate layer of the discriminator network. Experiments have shown that using feature matching technique, we can make the training stable when it is unstable.

1.6 Image Scaling and Mini Batches

Images are stored as arrays of pixel values where each pixel can have an integer value from 0 to 255, depicting the intensity or brightness of a pixel. For stable training of GANs for image synthesis, it is recommended that the pixel values in real images are scaled to a range of [-1, 1], and the generator network uses the hyperbolic tangent (*tanh*) as the activation function in the output layer. This scaling is generally performed using min-max scaling technique.

Secondly, it is recommended to train GANs with mini-batches. This helps the model in learning faster. In some GAN variants, researchers have recommended the batch sizes as small as 1 or 2. It is also recommended that we update the weights of the discriminator network with separate mini batches of real and fake (or generated) images.

1.7 Label Smoothing

Label smoothing is a technique in which we replace original classification labels of training data with smoothed values. For example: In a binary classification problem the original labels of 0 and 1 are replaced with smoothed values such as 0.1 and 0.9 before training. Doing this small trick improves the stability of training and also makes the model more robust. Label smoothing has also been shown to reduce vulnerability of the neural networks to adversarial examples (we will learn about the adversarial examples later in this book).

As per the findings of the researchers, label smoothing also helps in stabilizing the training of GANs.

1.8 Weight Initialization

Generally, neural network weights are initialized with random values before starting the training. Weight initialization can also affect the results and training behavior of GANs. It is recommended to initialize all network weights using zero-centered gaussian distribution (or normal bell-shaped distribution) with a standard deviation of 0.02. We can do this by passing the gaussian distribution into '*kernel_initializer*' parameter of convolutional layers using keras library.

1. Conclusion

As we already know that GANs are very difficult to train due to their nature of training. Thus, it becomes really important to understand the best-practices and recommended hacks that result in stable training of GANs.

In this skill, we learned about few ways of making GAN training stable. These practices are quite important to remember, as these have been taken from research papers where researchers have done tons of experiments to come up with these hacks.

After reading these best practices, hopefully the readers will be able to define a stable GAN architecture for their experimentations very easily. In the next skill, we will learn about conditional GANs.

End of Skill-5

SKILL 6

Conditional GANs

So far, we have seen some extraordinary generative capabilities of Generative Adversarial Network framework. It beats all previous works on generative models in terms of the quality of generated content. Though the generated images are very much realistic, there is not much control on what one wants to generate. A control that gives the flexibility to provide certain attributes about the samples one is interested in generating. Quite some research has been done on this, since the advent of GANs in 2014.

In this skill, we will learn about some the techniques to get conditional outputs from GANs. The resulted GAN architectures are broadly known as conditional GANs.

This skill covers the following topics about conditional GANs:

- What are conditional GANs?
- Introduction to CGAN
- Introduction to SGAN
- Introduction to Info-GAN
- Introduction to ACGAN
- Experiments
- Conclusion

Let's get started.

1. What are conditional GANs?

Conditional Generative Adversarial Networks or Conditional GANs, are modifications to the original GAN architecture aimed at providing some control over the generated content. In other words, these methods provide the flexibility of training GANs in such a way that there is some control over the data generation process. This control is very useful in practical applications, where we don't just want to generate realistic content, we want

it to be of some desired type (e.g., black cat with long furs). In a traditional GAN, this kind of control is not feasible.

In a traditional GAN, as we have seen that changing the latent vector slightly, changes the model output in a deterministic way, but this is not the kind of control we want, as it doesn't let us choose the modifications that we actually want in the output.

To solve this issue and gain more control over the generated content, the conditional GANs were introduced. Conditional GANs are slight modifications to the architectures and training paradigms of the traditional GANs.

In this skill, we will learn and implement four different ways of training conditional GANs:

- CGAN
- Semi-Supervised GAN (SGAN)
- Info-GAN
- AC-GAN

Let's get started with CGAN.

1. Introduction to CGAN

Conditional GAN, or CGAN for short, was introduced by Mehdi Mirza and Simon Osindero in their research paper titled '*Conditional Generative Adversarial Nets*' in 2014, only a few months after the introduction of original GAN paper. In their paper, they achieve a conditional GAN by passing control information to both generator and the discriminator networks and slightly changing the GAN training objective.

The 'control information' could be in the form of the class labels or any other known attributes related to the samples in the training dataset. This control information forces the generator to generate a specific type of image, and the discriminator is responsible for checking its validity (real or fake) as

well as the type (whether it belongs to the desired type or not as per the control information provided along with the sample).

The original GAN objective is defined as:

$$\min_{\theta} \max_{\phi} V(G_{\theta}, D_{\phi}) = E_{x \sim p_{\text{data}}} [\log D_{\phi}(x)] + E_{z \sim p_z} [\log(1 - D_{\phi}(G_{\theta}(z)))]$$

In case of CGAN, the new objective for minimax game can be written as:

$$\min_{\theta} \max_{\phi} V(G_{\theta}, D_{\phi}) = E_{x \sim p_{\text{data}}} [\log D_{\phi}(x|y)] + E_{z \sim p_z} [\log(1 - D_{\phi}(G_{\theta}(z|y)))]$$

Here, y is the control information (such as class labels).



Figure 6.1: CGAN results on MNIST data from the original paper

Though, CGAN was able to provide control over the outputs as shown in Figure 6.1, the quality of generated outputs degraded a little bit from the

ordinary method. But this was a good start of getting control over the results. Let's learn about SGAN next.

1. Introduction to SS-GAN

Semi-supervised GAN, or SGAN for short, is an extension to the traditional GAN framework that allows us to learn a generative model along with a classifier simultaneously.

SGAN was introduced by Augustus Odena in their research paper *titled ‘Semi Supervised Learning with Generative Adversarial Networks’* in 2016. The application of GANs for semi-supervised learning tasks was first highlighted by Tim Salimans in their paper *titled ‘Improved techniques for training GANs’* in 2016.

The idea is simple, instead of discriminator predicting just real vs fake, make it predict the class labels for real samples with one extra class corresponding to the fake samples. Hence, if there are k possible classes of real data, we will make the discriminator to predict $k+1$ classes as outputs and here one extra class corresponds to the fake samples coming from the generator. Once the training is complete, the discriminator model can be applied as a classifier ignoring the generator model.

The discriminator model as a classifier in this case defeats the supervised classifier by a big margin. To make this comparison fair, the authors train the discriminator without a generator (as a vanilla classifier) for supervised model and then, train the same discriminator in SGAN setting for semi-supervised version, with same number of training examples. While training the discriminator as a plain classifier on test dataset, the output of class related to the fake samples is ignored and most probable class is taken as the final output. The authors saw that SGAN based classifier performs better when training samples are limited as shown in the table from Figure 6.2 as well.

Table 1. Classifier Accuracy

EXAMPLES	CNN	SGAN
1000	0.965	0.964
100	0.895	0.928
50	0.859	0.883
25	0.750	0.802

Figure 6.2: From the original SGAN paper by Augustus Odena

As an added advantage, SGANs generate better quality of samples in very small amount of training time. Results shown in Figure 6.3, confirm that SGAN learns very fast to generate good quality images.



Figure 1. Output samples from SGAN and GAN after 2 MNIST epochs. SGAN is on the left and GAN is on the right.

Figure 6.3: Results taken from the SGAN paper

We just saw that SGAN can generate better quality results and also trains a high performing classifier. Next, let's learn about info GANs.

1. Introduction to Info-GAN

Info-GAN is an information theory related extension to the traditional GAN framework that was introduced by Xi Chen *et al.*, in their paper titled ‘*Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets*’ in 2016.

Interesting fact about Info-GANs is that they are capable of learning disentangled representations in a completely unsupervised manner. Disentangled representations are very useful and learning them in an unsupervised manner is like icing on the cake. A representation is said to be disentangled when different dimensions of the representation, represent some specific attributes related to the dataset.

For example: In a disentangled representation of human faces, one dimension could correspond to hairstyles, second might be related to skin color, another dimension might be related to emotion and so on. Disentangled representations are useful to learn as they can be directly used in so many downstream tasks.

Info-GAN has the capability to learn meaningful and interpretable representations using a fairly simple trick applied over the traditional GANs. It keeps a small subset of the latent variables (apart from the random noise or unit gaussian) fixed and maximizes the mutual information between them and the observations. This simple trick makes this subset of latent variables (also called control codes) learn important things about the generated content in completely unsupervised manner. The resulting disentangled representations are comparable to the representations learned by supervised techniques. Let's get into more details about the new objective function of Info GANs.

Suppose, c denotes the subset of latent variables (control codes) as discussed above and z represents the remaining latent variables (from noise vector). In this case our generator can be written as $G(z, c)$, which is similar to $G(z)$, unless we change the GAN objective function. As per the trick described above, we will try to maximize the mutual information between control codes c , and the generated distribution $G(z, c)$. It means that $I(c; G(z, c))$ should be higher.

We know that the original mini-max objective of GAN can be written as:

$$\min_{\theta} \max_{\phi} V(G_{\theta}, D_{\phi}) = E_{x \sim p_{\text{data}}} [\log D_{\phi}(x)] + E_{z \sim p_z} [\log(1 - D_{\phi}(G_{\theta}(z)))]$$

Minimax objective for Info-Gan (or information regularized mini-max game objective) can be defined as:

$$\min_{\theta} \max_{\emptyset} V_{\text{info-gan}}(G_{\theta}, D_{\emptyset}) = V(G_{\theta}, D_{\emptyset}) - \lambda I(c; G_{\theta}(z, c))$$

Here, λ is a hyperparameter.

In practice, Maximizing the mutual information $I(c; G(z, c))$ is not straight forward as it requires a reverse mapping function $P(c|x)$. Info-GAN defines an auxiliary distribution $Q(c|x)$, in order to approximate the lower bound of posterior $P(c|x)$. This approach of estimating lower bound of mutual information is called variational information maximization.

Auxiliary distribution Q is defined as a neural network and most of the times it shares initial few convolutional layers with the discriminator network D , and there is one final FC layer to predict the conditional $Q(c|x)$. Computationally, Info-GAN does not add lot of complexity to the existing GAN training, and often converges faster than the regular GAN as an added advantage.

Control codes c , can be discrete, categorical, continuous and also the combination of discrete and continuous variables. Discrete variables usually represent the drastic changes in output (e.g., output class), while the continuous codes can be used to capture continuous variations (e.g., image rotations). Let's look at the results now.

4.1 Info GAN Results

Info-GAN paper presents results on MNIST handwritten digits dataset. Info-GAN was trained on MNIST with three control codes where c_1 is discrete, while c_2 and c_3 are continuous latent codes. The results shown in Figure 6.4 depict that the control digit c_1 automatically learns to represent drastic shape changes (digit classes), and c_2 represents continuous rotations and c_3 captures digit width in a continuous manner.

Part(b) of the same Figure shows that the regular GAN (without variational information maximization) does not have very meaningful (or deterministic) relationships between the generated outputs and latent codes.

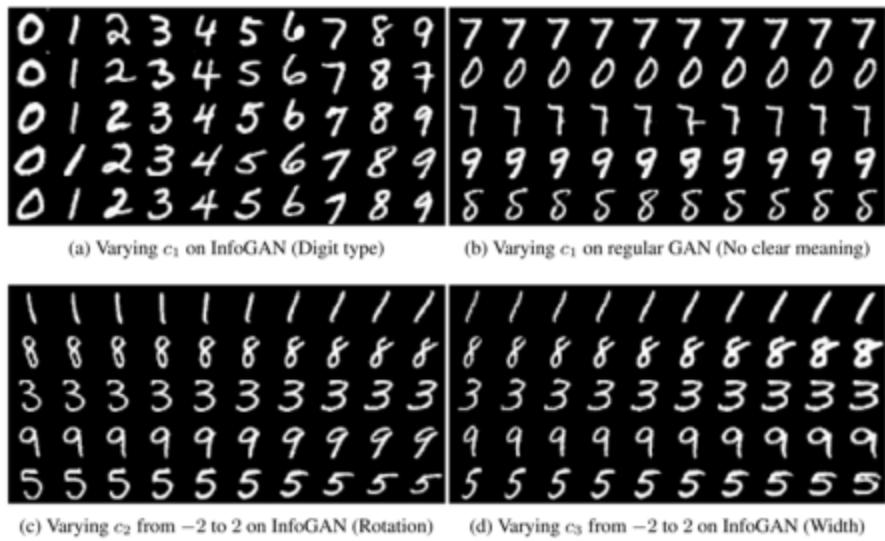


Figure 6.4: Results taken from original Info-GAN paper

In one of the experiments related to 3D face generation tasks, Info-GAN learns continuous control codes capable of capturing attributes like poses and elevations in a completely unsupervised manner. See Figure 6.5.

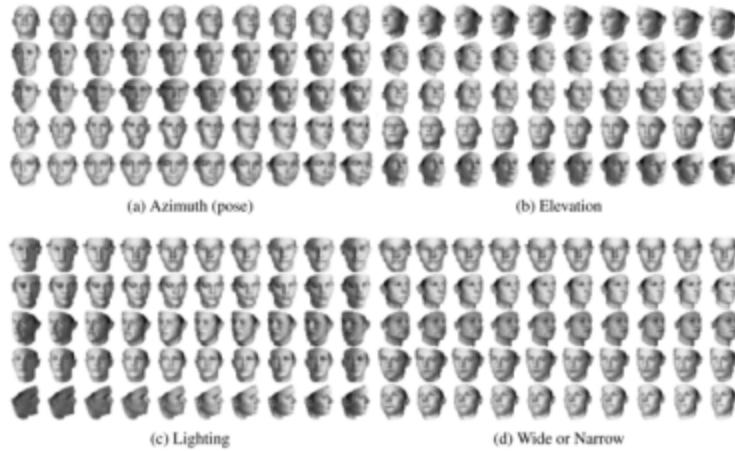


Figure 6.5: Info-GAN results from the original paper

Another experiment on 3D chairs learns to capture unsupervised attributes that can control the rotation and the width of chairs as shown in the Figure 6.6.



Figure 6.6: Info-GAN results taken from the original paper

We just looked at some of the impressive results of Info GAN. With just a small training trick, it can provide us lot of control over the outputs generated by GANs.

Next, let's learn about ACGAN.

1. Introduction to ACGAN

Auxiliary Classifier GAN, or ACGAN for short, is another variant of Generative Adversarial Networks that improves GAN training for image synthesis, when image labels are present.

ACGANs were introduced by Augustus Odena *et. el.*, in their research paper titled '*Conditional Image Synthesis with Auxiliary Classifier GANs*' (2017). This paper shows that ACGANs are capable of generating high-quality well-diversified images as compared to other methods known by that time.

The experiments from conditional GAN (CGAN), semi-supervised GAN (SGAN) and info-GAN show that the additional information passed (in terms of class labels, control codes and so on) to the ordinary GAN model can improve the quality of generated samples. Keeping this in mind, ACGANs combine the qualities of above three models and feed the class label information into both generator as well as the discriminator networks, resulting into a class conditioned GAN model.

In case of ACGAN, the generator network uses class labels c (in addition to noise z) to generate images $X_{fake} = G(z, c)$. The discriminator generates two probability distributions, one corresponding to the data source (dataset or generator) $P(S|X)$ and other one over the class labels $P(C|X)$. ACGAN objective function has two parts: one for data source (L_s) and the other one for the class labels (L_c).

Log-likelihood related to source of data:

$$L_s = E[\log P(S = \text{real} \mid X_{\text{real}})] + E[\log P(S = \text{fake} \mid X_{\text{fake}})]$$

Log-likelihood related to the class labels:

$$L_c = E[\log P(C = c \mid X_{\text{real}})] + E[\log P(C = c \mid X_{\text{fake}})]$$

In the ACGAN model, the discriminator network is trained to maximize the log-likelihood $L_s + L_c$, which renders a strong discriminator as well as a strong classifier.

The generator model is trained to maximize $L_c - L_s$, which makes sure that the generated samples belong to the correct class and they are realistic enough to fool the discriminator. The representations learned by AC-GAN for z are independent of the class labels.

The main advantages of using ACGANs, are high quality results and stable training. However, the output quality decreases with increase in number of classes. The original paper suggests breaking the high-class datasets into multiple small datasets each with few classes and train separate pair of generators and discriminators for each partition. Let's look at the results.

5.1 ACGAN Results

Figure 6.7 shows some ACGAN results from the original paper. Where, (Top) Latent space interpolations for selected ImageNet classes. Left-most and right-columns show three pairs of image samples: each pair from a distinct class. Intermediate columns highlight linear interpolations in the latent space between these three pairs of images. (Bottom) Class-independent information contains global structure about the synthesized image. Each

column is a distinct bird class while each row corresponds to a fixed latent code z .

As we can see that the results from ACGAN model look very impressive and good thing is that we have some control over the results.



Figure 6.7: ACGAN results taken from the original paper

Now that we know about different variants of conditional GANs. Let's implement them ourselves and verify results.

1. Experiments

In this skill, we will implement and train four variants of conditional GANs as described in previous sections. Specifically, we will perform the following four experiments:

- Conditional GAN or CGAN for MNIST handwritten digits
- Semi-Supervised GAN or SGAN for Fashion MNIST
- Information GAN or Info-GAN for MNIST handwritten digits
- Auxiliary Classifier GAN or ACGAN for Fashion MNIST

The code samples shown in this skill can be downloaded from the following Github location:

<https://github.com/kartikgill/The-GAN-Book/tree/main/Skill-06>

Let's get started.

CONDITIONAL GAN OR CGAN FOR MNIST

Objective

In this experiment, we will implement and train a CGAN model on MNIST Handwritten Digits dataset.

This experiment has the following steps:

- Import Useful Libraries
- Download and Show Data
- Data Normalization
- Define Generator Model
- Define Discriminator Model
- Define Combined Model: CGAN
- Utility Functions
- Training CGAN
- Results

Let's get started.

Step 1: Import Useful Libraries

Open a Jupyter Notebook with python kernel and import useful python packages in a cell. In this experiment, we will use ‘*numpy*’ for data manipulation, ‘*matplotlib*’ for plotting images and graphs, and *TensorFlow2* for implementing the model architecture.

Checkout the following snippet for importing libraries:

```
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
from tqdm import tqdm_notebook  
%matplotlib inline  
import tensorflow  
print(tensorflow.__version__)
```

Let's download the dataset now.

Step 2: Download and Show Data

In this step, we will download the MNIST Handwritten digits dataset and show some samples. The following python code loads the data and plots a few samples:

```
from tensorflow.keras.datasets import fashion_mnist, mnist
(trainX, trainY), (testX, testY) = mnist.load_data()
print('Training data shapes: X=%s, y=%s' % (trainX.shape, trainY.shape))

print('Testing data shapes: X=%s, y=%s' % (testX.shape, testY.shape))
for k in range(7):
    plt.figure(figsize=(7, 7))
    for j in range(7):
        i = np.random.randint(0, 10000)
        plt.subplot(770 + 1 + j)
        plt.imshow(trainX[i], cmap='gray_r')
        plt.axis('off')
    plt.show()
```

Figure 6.8 shows some samples from the dataset. MNIST Handwritten Digits dataset has 60k training images and 10k test images of size 28 x 28, and each image is gray scale and comes with a label value indicating the digit class.



Figure 6.8: Some samples from the MNIST Handwritten Digits dataset

Let's prepare the data for model.

Step 3: Data Normalization

In this step, we will prepare the data for model. We will first normalize pixel values of each image and restrict them to range [-1, 1] by subtracting and dividing with 127.5. We will also add a channel dimension to each image, as expected by the convolutional neural network layers.

The following python snippet prepares the dataset for model:

```
trainX = [(image-127.5)/127.5 for image in trainX]
testX = [(image-127.5)/127.5 for image in testX]
trainX = np.reshape(trainX, (60000, 28, 28, 1))
testX = np.reshape(testX, (10000, 28, 28, 1))
print (trainX.shape, testX.shape, trainY.shape, testY.shape)
```

Following output shows the final shapes of prepared data:

```
(60000, 28, 28, 1) (10000, 28, 28, 1) (60000,) (10000,)
```

Our data is now ready, let's define the model architecture now.

Step 4: Define Generator Model

In this step, we will define the generator network. In our setup, the generator network accepts two inputs: a class input of size 10, and a random

noise vector of size 100. These two inputs are concatenated and passed through multiple layers of *Conv2DTranspose*, *ReLU* activation, and Batch Normalization with a momentum value of 0.8. The final layer has a ‘*tanh*’ activation function to restrict the pixel values of generated image within range [-1, 1], similar to our training dataset.

The following python code implements the generator network:

```
class_input = tensorflow.keras.layers.Input(shape = 10)
x1 = tensorflow.keras.layers.Dense(5*5*4)(random_input)
x1 = tensorflow.keras.layers.Activation('relu')(x1)
x1 = tensorflow.keras.layers.Reshape((5, 5, 4))(x1)
#Class Input
x2 = tensorflow.keras.layers.Dense(25)(class_input)
x2 = tensorflow.keras.layers.Activation('relu')(x2)
x2 = tensorflow.keras.layers.Reshape((5, 5, 1))(x2)
x = tensorflow.keras.layers.concatenate([x1, x2])
x = tensorflow.keras.layers.Conv2DTranspose(filters=64, kernel_size=(5,5))(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=64, kernel_size=(7,7))(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=128, kernel_size=(7,7))(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=1, kernel_size=(8,8))(x)
generated_image = tensorflow.keras.layers.Activation('tanh')(x)
```

```
generator_network = tensorflow.keras.models.Model(inputs=[random_input, class_input], outputs=generated_image)
generator_network.summary()
```

Following is the summary of the generator network. It roughly has 630k trainable parameters.

Out [4]: Model: "model"

Layer (type) Output Shape Param # Connected to

input_1 (InputLayer) [(None, 100)] 0

input_2 (InputLayer) [(None, 10)] 0

dense (Dense) (None, 100) 10100 input_1[0][0]

dense_1 (Dense) (None, 25) 275 input_2[0][0]

activation (Activation) (None, 100) 0 dense[0][0]

activation_1 (Activation) (None, 25) 0 dense_1[0][0]

reshape (Reshape) (None, 5, 5, 4) 0 activation[0][0]

reshape_1 (Reshape) (None, 5, 5, 1) 0 activation_1[0][0]

concatenate (Concatenate) (None, 5, 5, 5) 0 reshape[0][0]

reshape_1[0][0]

conv2d_transpose (Conv2DTranspo (None, 9, 9, 64) 8064 concatenate[0][0]

activation_2 (Activation) (None, 9, 9, 64) 0 conv2d_transpose[0][0]

batch_normalization (BatchNorma (None, 9, 9, 64) 256 activation_2[0][0]

conv2d_transpose_1 (Conv2DTrans (None, 15, 15, 64) 200768 batch_normalization[0][0]

activation_3 (Activation) (None, 15, 15, 64) 0 conv2d_transpose_1[0][0]

batch_normalization_1 (BatchNor (None, 15, 15, 64) 256 activation_3[0][0]

conv2d_transpose_2 (Conv2DTrans (None, 21, 21, 128) 401536 batch_normalization_1[0][0]

activation_4 (Activation) (None, 21, 21, 128) 0 conv2d_transpose_2[0][0]

conv2d_transpose_3 (Conv2DTrans (None, 28, 28, 1) 8193 activation_4[0][0]

activation_5 (Activation) (None, 28, 28, 1) 0 conv2d_transpose_3[0][0]

Total params: 629,448

Trainable params: 629,192

Non-trainable params: 256

Let's work on the discriminator network now.

Step 5: Define Discriminator Model

In this step, we will define the discriminator network for our experiment. In our setup, the discriminator network accepts two inputs: an image of size 28 x 28 x 1 and a class input of size 10. These inputs are then concatenated and passed through multiple layers of *Conv2D*, *LeakyReLU* activation and Batch normalization with a momentum value of 0.8. Final layer has a single neuron and ‘sigmoid’ activation that will generate a probability value indicating if the input sample is real or fake.

The following python code implements the discriminator architecture:

```
image_input = tensorflow.keras.layers.Input(shape=(28, 28, 1))
class_input = tensorflow.keras.layers.Input(shape = 10)
#Class Input
x2 = tensorflow.keras.layers.Dense(28*28)(class_input)
x2 = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x2)
x2 = tensorflow.keras.layers.Reshape((28, 28, 1))(x2)
x = tensorflow.keras.layers.concatenate([image_input, x2])
x = tensorflow.keras.layers.Conv2D(filters=128, kernel_size=
(3,3), strides=2)(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.Conv2D(filters=128, kernel_size=
(3,3), strides=2)(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2D(filters=64, kernel_size=(3,3))(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2D(filters=64, kernel_size=(3,3))(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Flatten()(x)
x = tensorflow.keras.layers.Dense(1)(x)
real_vs_fake_output = tensorflow.keras.layers.Activation('sigmoid')(x)
discriminator_network = tensorflow.keras.models.Model(inputs=
[image_input, class_input], outputs=real_vs_fake_output)
discriminator_network.summary()
```

Following is the summary of the discriminator network. It has roughly 270k trainable parameters:

Out [5]: Model: "model_1"

Layer (type) Output Shape Param # Connected to

input_4 (InputLayer) [(None, 10)] 0

dense_2 (Dense) (None, 784) 8624 input_4[0][0]

leaky_re_lu (LeakyReLU) (None, 784) 0 dense_2[0][0]

input_3 (InputLayer) [(None, 28, 28, 1)] 0

reshape_2 (Reshape) (None, 28, 28, 1) 0 leaky_re_lu[0][0]

concatenate_1 (Concatenate) (None, 28, 28, 2) 0 input_3[0][0]

reshape_2[0][0]

conv2d (Conv2D) (None, 13, 13, 128) 2432 concatenate_1[0][0]

leaky_re_lu_1 (LeakyReLU) (None, 13, 13, 128) 0 conv2d[0][0]

conv2d_1 (Conv2D) (None, 6, 6, 128) 147584 leaky_re_lu_1[0][0]

leaky_re_lu_2 (LeakyReLU) (None, 6, 6, 128) 0 conv2d_1[0][0]

batch_normalization_2 (BatchNor (None, 6, 6, 128) 512 leaky_re_lu_2[0][0]

conv2d_2 (Conv2D) (None, 4, 4, 64) 73792 batch_normalization_2[0][0]

leaky_re_lu_3 (LeakyReLU) (None, 4, 4, 64) 0 conv2d_2[0][0]

batch_normalization_3 (BatchNor (None, 4, 4, 64) 256 leaky_re_lu_3[0][0]

conv2d_3 (Conv2D) (None, 2, 2, 64) 36928 batch_normalization_3[0][0]

leaky_re_lu_4 (LeakyReLU) (None, 2, 2, 64) 0 conv2d_3[0][0]

```
batch_normalization_4 (BatchNor (None, 2, 2, 64) 256 leaky_re_lu_4[0][0]
```

```
flatten (Flatten) (None, 256) 0 batch_normalization_4[0][0]
```

```
dense_3 (Dense) (None, 1) 257 flatten[0][0]
```

```
activation_6 (Activation) (None, 1) 0 dense_3[0][0]
```

```
=====
```

Total params: 270,641

Trainable params: 270,129

Non-trainable params: 512

We can now compile the discriminator network with ‘*binary_crossentropy*’ loss function. We will use Adam optimizer with a learning rate of 0.0001 and beta_1 value of 0.5, for updating the weights of the discriminator model.

```
adam_optimizer = tensorflow.keras.optimizers.Adam(learning_rate=0.0001  
, beta_1=0.5)
```

```
discriminator_network.compile(loss='binary_crossentropy', optimizer=adam_o  
ptimizer, metrics=['accuracy'])
```

Let’s define the combined model now.

Step 6: Define Combined Model: CGAN

In this step, we will define the combined CGAN model. Similar to the traditional GANs, we will freeze the parameters of the discriminator network and combine both networks by passing the output of the generator network into the discriminator network as input. Additionally, we will pass the class input vectors into both generator and the discriminator networks as a second input.

The following python code defines the combined CGAN model:

```
discriminator_network.trainable=False  
gan_input = generator_network([random_input, class_input])  
gan_output = discriminator_network([gan_input, class_input])  
gan_model = tensorflow.keras.models.Model([random_input, class_input]  
, gan_output)  
gan_model.summary()
```

Following is the summary of the combined model:

Out[6]: Model: "model_2"

Layer (type) Output Shape Param # Connected to

```
=====
```

input_1 (InputLayer) [(None, 100)] 0

input_4 (InputLayer) [(None, 10)] 0

model (Functional) (None, 28, 28, 1) 629448 input_1[0][0]

input_4[0][0]

```
model_1 (Functional) (None, 1) 270641 model[0][0]
```

```
input_4[0][0]
```

```
Total params: 900,089
```

```
Trainable params: 629,192
```

```
Non-trainable params: 270,897
```

We can compile the combined model in the similar way to the discriminator model. See the following python code:

```
adam_optimizer = tensorflow.keras.optimizers.Adam(learning_rate=0.0001  
, beta_1=0.5)
```

```
gan_model.compile(loss='binary_crossentropy', optimizer=adam_optimizer)
```

Our dataset and model are ready now. Let's work on the training piece.

Step 7: Utility Functions

In this step, we will define some utility functions that will help us in creating data batches for the model training. Specifically, we will define the utility functions for: generating random noise input batches with class inputs, generating batches of fake samples along with class inputs, generating batches of real samples along with class inputs. In addition to these, we will also define a utility function for plotting the generator model results.

The following python code implements the utility functions for training:

```
indices = [i for i in range(0, len(trainX))]
```

```
def encode_class(value):
```

```
    x = np.zeros((10))
```

```
    x[value]=1
```

```
    return x
```

```
def decode_class(value):
```

```

return np.where(value==1)[0][0]

def get_random_noise(batch_size, noise_size):
    random_values = np.random.randn(batch_size*noise_size)
    random_noise_batches = np.reshape(random_values, (batch_size, noise_size))
    class_names = []
    for i in range(batch_size):
        class_name = np.random.choice([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], size=1)
        class_names.append(encode_class(class_name))
    class_names = np.array(class_names)
    return random_noise_batches, class_names

def get_fake_samples(generator_network, batch_size, noise_size, class_name=-1):
    random_noise_batches, _ = get_random_noise(batch_size, noise_size)
    if class_name == -1:
        class_names = []
        for i in range(batch_size):
            class_name = np.random.choice([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], size=1)
            class_names.append(encode_class(class_name))
        class_names = np.array(class_names)
    else:
        class_names = []
        for i in range(batch_size):
            class_names.append(encode_class(class_name))
        class_names = np.array(class_names)
    fake_samples = generator_network.predict_on_batch([random_noise_batches, class_names])
    return [fake_samples, class_names]

def get_real_samples(batch_size, class_names=-1):

```

```

if class_names == -1:
    random_indices = np.random.choice(indices, size=batch_size)
    class_names = []
for ind in random_indices:
    class_names.append(encode_class(trainY[ind]))
else:
    random_indices = []
for cls in class_names:
    ind = np.random.choice(np.where(trainY==cls), size=1)
    random_indices.append(ind)
real_images = trainX[np.array(random_indices),:]
return [real_images, class_names]

def show_generator_results(generator_network):
    for k in range(10):
        fake_samples = get_fake_samples(generator_network, 10, noise_size, k)
        plt.figure(figsize=(10, 6))
        for j in range(9):
            i = j
            plt.subplot(990 + 1 + j)
            plt.imshow(fake_samples[0][i,:,:,-1], cmap='gray_r')
            #plt.title(decode_class(fake_samples[1][i]))
            plt.axis('off')
        plt.show()
    return

```

We can now go ahead and start training the model.

Step 8: Training CGAN

In this step, we will write the training iteration loop for our CGAN model. We plan to train our model for 500 epochs, with a batch size of 100, class input size of 10 and noise vector size of 100. In each iteration, we will first

update the weights of the discriminator network with mix batch of real and fake samples, along with the class inputs. Then, we will freeze the discriminator weights and update the generator weights using only fake samples along with the class inputs. Just like traditional GANs, we will pass the inverted labels (1 instead of 0) of fake samples while updating the generator weights, to make the discriminator believe that these are real samples and calculate the loss value.

The following python code implements the training loop for our CGAN model:

```
epochs = 500
batch_size = 100
steps = 500
noise_size = 100
tensorflow.config.run_functions_eagerly(True)
for i in range(0, epochs):
    if (i%5 == 0):
        op = show_generator_results(generator_network)
        #print (op)
    for j in range(steps):
        fake_samples = get_fake_samples(generator_network, batch_size//2, noise_size)
        real_samples = get_real_samples(batch_size=batch_size//2)
        fake_y = np.zeros((batch_size//2, 1))
        real_y = np.ones((batch_size//2, 1))
        input_batch_part1 = np.vstack((fake_samples[0], real_samples[0]))
        input_batch_part2 = np.vstack((fake_samples[1], real_samples[1]))
        input_batch_final = [input_batch_part1, input_batch_part2]
        output_labels = np.vstack((fake_y, real_y))
        # Updating Discriminator weights
        discriminator_network.trainable=True
```

```

loss_d = discriminator_network.train_on_batch(input_batch_final, output_labels)

noise_batches, class_values = get_random_noise(batch_size, noise_size)
gan_input = [noise_batches, class_values]

# Make the Discriminator believe that these are real samples and calculate loss to train the generator

gan_output = np.ones((batch_size))

# Updating Generator weights

discriminator_network.trainable=False

loss_g = gan_model.train_on_batch(gan_input, gan_output)

if j%50 == 0:

    print ("Epoch:%.0f, Step:%.0f, D-Loss:%.3f, D-Acc:%.3f, G-Loss:%.3f"%
(i,j,loss_d[0],loss_d[1]*100,loss_g))

```

Following are the training logs of our model at every 50th step:

Out [8]: Epoch:0, Step:0, D-Loss:0.907, D-Acc:24.000, G-Loss:0.707

Epoch:0, Step:50, D-Loss:0.149, D-Acc:96.000, G-Loss:4.448

Epoch:0, Step:100, D-Loss:0.050, D-Acc:99.000, G-Loss:4.502

Epoch:0, Step:150, D-Loss:0.167, D-Acc:94.000, G-Loss:3.893

Epoch:0, Step:200, D-Loss:0.047, D-Acc:99.000, G-Loss:4.180

Epoch:0, Step:250, D-Loss:0.126, D-Acc:95.000, G-Loss:6.380

Epoch:0, Step:300, D-Loss:0.095, D-Acc:97.000, G-Loss:6.760

Epoch:0, Step:350, D-Loss:0.151, D-Acc:94.000, G-Loss:5.525

Epoch:0, Step:400, D-Loss:0.131, D-Acc:97.000, G-Loss:4.646

Epoch:0, Step:450, D-Loss:0.231, D-Acc:93.000, G-Loss:4.047

Epoch:1, Step:0, D-Loss:0.230, D-Acc:91.000, G-Loss:3.654

...

...

...

...

```
Epoch:29, Step:150, D-Loss:0.301, D-Acc:85.000, G-Loss:3.745
Epoch:29, Step:200, D-Loss:0.220, D-Acc:94.000, G-Loss:0.921
Epoch:29, Step:250, D-Loss:0.217, D-Acc:93.000, G-Loss:5.268
Epoch:29, Step:300, D-Loss:0.275, D-Acc:89.000, G-Loss:5.769
Epoch:29, Step:350, D-Loss:0.230, D-Acc:84.000, G-Loss:4.726
Epoch:29, Step:400, D-Loss:0.193, D-Acc:91.000, G-Loss:0.288
Epoch:29, Step:450, D-Loss:0.261, D-Acc:88.000, G-Loss:2.102
```

As our model training is complete now. Let's check out the results.

Step 9: Results

Figure 6.9 shows the generator results at different number of epochs. We can see that even after just 5 epochs, the model starts learning to generate the digits. At epoch 15, the results start looking realistic. At epoch 30, the results look very neat and realistic. Important thing to notice here is that each row is generating a particular class of digit images. This was only possible using the class input variables. For more information check out the Jupyter Notebook present in the Github repository of this book.

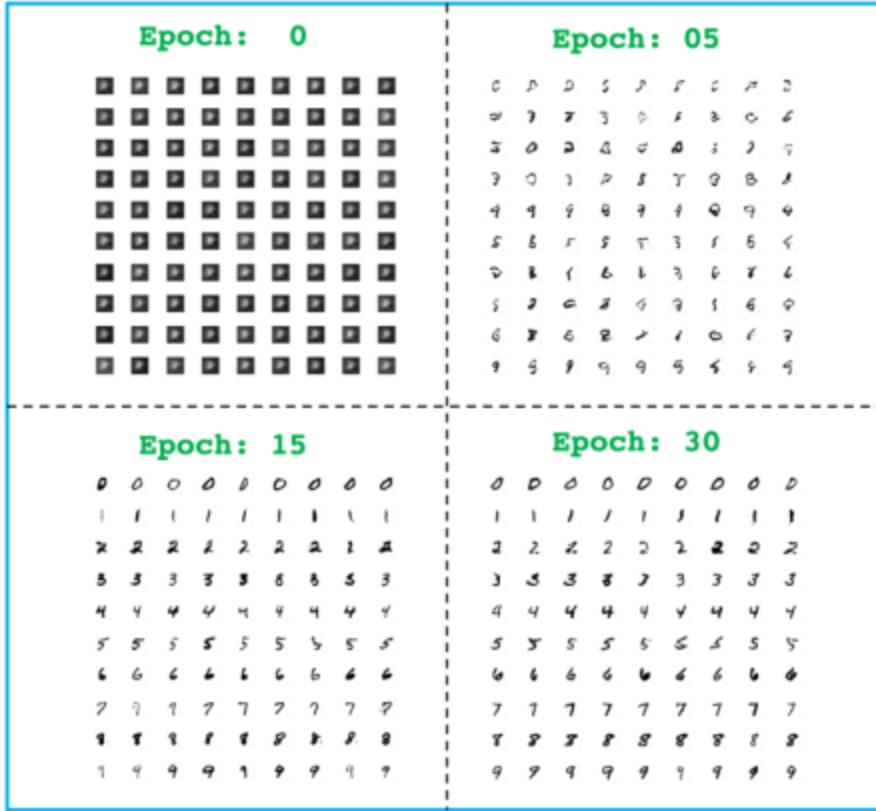


Figure 6.9: Handwritten Digit images generated through a CGAN model

We just saw that CGANs are capable of generating class-controlled results in high quality. The training was also very fast and we started seeing good results even after 5 epochs of training.

Let's now implement the SGAN model.

SEMI-SUPERVISED OR SGAN FOR FASHION MNIST

Objective

In this experiment, we will train a SGAN model on Fashion MNIST dataset and verify its results.

This experiment has the following steps:

- Import Useful Libraries
- Download and Show Data
- Data Normalization
- Define Generator Model
- Define Discriminator Model
- Define Combined Model: SGAN
- Utility Functions
- Training SGAN
- Results

Let's get started.

Step 1: Import Useful Libraries

Open a Jupyter Notebook with python kernel and import useful python packages in a cell. In this experiment, we will use ‘*numpy*’ for data manipulation, ‘*matplotlib*’ for plotting images and graphs, and *TensorFlow2* for implementing the model architecture.

Checkout the following snippet for importing libraries:

```
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
from tqdm import tqdm_notebook  
%matplotlib inline  
import tensorflow  
print(tensorflow.__version__)
```

Let's download the dataset now.

Step 2: Download and Show Data

In this step, we will download the MNIST Handwritten digits dataset and show some samples. The following python code loads the data and plots a few samples:

```
from tensorflow.keras.datasets import fashion_mnist, mnist  
(trainX, trainY), (testX, testY) = fashion_mnist.load_data()  
print('Training data shapes: X=%s, y=%s' % (trainX.shape, trainY.shape))  
  
print('Testing data shapes: X=%s, y=%s' % (testX.shape, testY.shape))
```

```
for k in range(9):
```

```
    plt.figure(figsize=(10, 7))  
    for j in range(9):  
        i = np.random.randint(0, 10000)  
        plt.subplot(990 + 1 + j)  
        plt.imshow(trainX[i], cmap='gray_r')  
        #plt.title(trainY[i])  
        plt.axis('off')  
    plt.show()
```

Figure 6.10 shows some samples from the dataset. Fashion MNIST dataset has 60k training images and 10k test images of size 28 x 28, and each sample is gray scale image and comes with a label value indicating the class value.



Figure 6.10: Few samples for the Fashion MNIST dataset

Let's now prepare the dataset for model.

Step 3: Data Normalization

In this step, we will prepare the dataset for the model. We will first normalize pixel values of each image and restrict them to range [-1, 1] by subtracting and dividing with 127.5. We will also add a channel dimension to each image, as expected by the convolutional neural network layers.

The following python snippet prepares the dataset for model:

```
trainX = [(image-127.5)/127.5 for image in trainX]
testX = [(image-127.5)/127.5 for image in testX]
trainX = np.reshape(trainX, (60000, 28, 28, 1))
testX = np.reshape(testX, (10000, 28, 28, 1))
print (trainX.shape, testX.shape, trainY.shape, testY.shape)
```

The final dataset has the following shape:

(60000, 28, 28, 1) (10000, 28, 28, 1) (60000,) (10000,)

Our dataset is now ready. Let's define the model now.

Step 4: Define Generator Model

In this step, we will define the generator network. In our setup, the generator accepts a random noise vector of size 100 as input. This input vector is first reshaped into a three-dimensional vector and then passed through the multiple layers of *Conv2DTranspose*, *ReLU* activation, and Batch Normalization with a momentum value of 0.8. The final layer has a ‘*tanh*’ activation function to restrict the pixel values of the generated image within range [-1, 1], similar to our training dataset.

The following python code implements the generator network:

```
random_input = tensorflow.keras.layers.Input(shape = 100)
x = tensorflow.keras.layers.Dense(5*5*64)(random_input)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Reshape((5, 5, 64))(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=128, kernel_size=(3,3), strides=2)(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=128, kernel_size=(3,3), strides=2)(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=128, kernel_size=(3,3))(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=1, kernel_size=(4,4))(x)
generated_image = tensorflow.keras.layers.Activation('tanh')(x)
generator_network = tensorflow.keras.models.Model(inputs=random_input, outputs=generated_image)
generator_network.summary()
```

Following is the summary of the generator network. It has about 530k trainable parameters:

Out [4]: Model: "model"

Layer (type)	Output Shape	Param #
--------------	--------------	---------

input_1 (InputLayer)	[(None, 100)]	0
----------------------	---------------	---

dense (Dense)	(None, 1600)	161600
---------------	--------------	--------

activation (Activation)	(None, 1600)	0
-------------------------	--------------	---

batch_normalization (BatchNo	(None, 1600)	6400
------------------------------	--------------	------

reshape (Reshape)	(None, 5, 5, 64)	0
-------------------	------------------	---

conv2d_transpose (Conv2DTran	(None, 11, 11, 128)	73856
------------------------------	---------------------	-------

activation_1 (Activation)	(None, 11, 11, 128)	0
---------------------------	---------------------	---

batch_normalization_1 (Batch	(None, 11, 11, 128)	512
------------------------------	---------------------	-----

conv2d_transpose_1 (Conv2DTr	(None, 23, 23, 128)	147584
------------------------------	---------------------	--------

activation_2 (Activation)	(None, 23, 23, 128)	0
---------------------------	---------------------	---

batch_normalization_2 (Batch	(None, 23, 23, 128)	512
------------------------------	---------------------	-----

conv2d_transpose_2 (Conv2DTr	(None, 25, 25, 128)	147584
------------------------------	---------------------	--------

```
activation_3 (Activation) (None, 25, 25, 128) 0
```

```
conv2d_transpose_3 (Conv2DTr (None, 28, 28, 1) 2049
```

```
activation_4 (Activation) (None, 28, 28, 1) 0
```

```
=====
```

Total params: 540,097

Trainable params: 536,385

Non-trainable params: 3,712

Let's define the discriminator network now.

Step 5: Define Discriminator Model

In this step, we will define the discriminator network for our experiment. In our setup, the discriminator network accepts an image of size 28 x 28 x 1 as input. The input image is then passed through multiple layers of *Conv2D*, *LeakyReLU* activation and Batch normalization with a momentum value of 0.8. The discriminator network has two output layers: one for classifying the image into real or fake, another for predicting the class of the output. The class output layer has 11 neurons (10 for 10 classes, and one extra for the generated or fake image class), and a *softmax* activation function to generate multiclass probabilities.

The following python code implements the discriminator architecture:

```
image_input = tensorflow.keras.layers.Input(shape=(28, 28, 1))
x = tensorflow.keras.layers.Conv2D(filters=128, kernel_size=
(3,3), strides=2)(image_input)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2D(filters=128, kernel_size=
(3,3), strides=2)(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
```

```
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2D(filters=128, kernel_size=(3,3))(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2D(filters=128, kernel_size=(4,4))(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
features = tensorflow.keras.layers.Flatten()(x)
d_out1 = tensorflow.keras.layers.Dense(1)(features)
real_vs_fake_output = tensorflow.keras.layers.Activation('sigmoid')(d_out1)
d_out2 = tensorflow.keras.layers.Dense(11)(features)
class_output = tensorflow.keras.layers.Activation('softmax')(d_out2)
discriminator_network = tensorflow.keras.models.Model(inputs=image_input, outputs=[real_vs_fake_output, class_output])
discriminator_network.summary()
```

Following is the summary of the discriminator network. It has roughly 560k trainable parameters.

Out [5]: Model: "model_1"

Layer (type) Output Shape Param # Connected to

input_2 (InputLayer) [(None, 28, 28, 1)] 0

conv2d (Conv2D) (None, 13, 13, 128) 1280 input_2[0][0]

leaky_re_lu (LeakyReLU) (None, 13, 13, 128) 0 conv2d[0][0]

batch_normalization_3 (BatchNor (None, 13, 13, 128) 512 leaky_re_lu[0][0]

conv2d_1 (Conv2D) (None, 6, 6, 128) 147584 batch_normalization_3[0][0]

leaky_re_lu_1 (LeakyReLU) (None, 6, 6, 128) 0 conv2d_1[0][0]

batch_normalization_4 (BatchNor (None, 6, 6, 128) 512 leaky_re_lu_1[0][0]

conv2d_2 (Conv2D) (None, 4, 4, 128) 147584 batch_normalization_4[0][0]

leaky_re_lu_2 (LeakyReLU) (None, 4, 4, 128) 0 conv2d_2[0][0]

batch_normalization_5 (BatchNor (None, 4, 4, 128) 512 leaky_re_lu_2[0][0]

conv2d_3 (Conv2D) (None, 1, 1, 128) 262272 batch_normalization_5[0][0]

leaky_re_lu_3 (LeakyReLU) (None, 1, 1, 128) 0 conv2d_3[0][0]

batch_normalization_6 (BatchNor (None, 1, 1, 128) 512 leaky_re_lu_3[0][0]

flatten (Flatten) (None, 128) 0 batch_normalization_6[0][0]

dense_1 (Dense) (None, 1) 129 flatten[0][0]

dense_2 (Dense) (None, 11) 1419 flatten[0][0]

activation_5 (Activation) (None, 1) 0 dense_1[0][0]

activation_6 (Activation) (None, 11) 0 dense_2[0][0]

=====

Total params: 562,316

Trainable params: 561,292

Non-trainable params: 1,024

None

As our discriminator network has two outputs, we will need two loss functions, one for each output layer. As the class output layer performs a multi-class classification, we will utilize the '*categorical_crossentropy*' as loss function. For the image validity output, we will utilize '*binary_crossentropy*' loss. We will Adam optimizer with a learning rate of 0.0001 and beta_1 value of 0.5 for updating the weights of the discriminator network.

The following python snippet compiles the discriminator network:

```
adam_optimizer = tensorflow.keras.optimizers.Adam(learning_rate=0.0001, beta_1=0.5)
```

```
discriminator_network.compile(loss=['binary_crossentropy', 'categorical_crossentropy'], optimizer=adam_optimizer,\nmetrics=['accuracy'], loss_weights=[0.5, 0.5])
```

Let's now define the combined GAN setup.

Step 6: Define Combined Model: SGAN

In this step, we will define the combined SGAN model, by passing the output of the generator network through a frozen discriminator network. Note that in the combined model, we will only use one output of the discriminator network that is related to the image validity: real vs. fake. We will ignore the class outputs as they do not make sense in case of the generator network.

The following python code defines the combined SGAN model:

```
discriminator_network.trainable=False\n\ng_output = generator_network(random_input)\nd_output = discriminator_network(g_output)\nreal_vs_fake = d_output[0]\nclass_output = d_output[1]\n\nss_gan_model = tensorflow.keras.models.Model(inputs = random_input,\noutputs=real_vs_fake)\n\nss_gan_model.summary()
```

Following is the summary of the combined SGAN model:

Out [6]: Model: "model_2"

Layer (type) Output Shape Param #

input_1 (InputLayer) [(None, 100)] 0

model (Functional) (None, 28, 28, 1) 540097

model_1 (Functional) [(None, 1), (None, 11)] 562316

Total params: 1,102,413

Trainable params: 536,385

Non-trainable params: 566,028

As our combined SGAN model only has a single binary classification output, we can compile it with '*binary_crossentropy*' loss. See the following code:

```
ss_gan_model.compile(loss=[  
    'binary_crossentropy'], optimizer=adam_optimizer)
```

Our model architecture is now ready. Let's work on the training part now.

Step 7: Utility Functions

In this step, we will define some utility functions that will help us in creating data batches for the model training. Specifically, we will define the utility functions for: generating random noise input batches, generating batches of fake samples, generating batches of real samples along with class labels. In addition to these, we will also define a utility function for plotting the generator model results.

The following python code implements the utility functions for training:

```

indices = [i for i in range(0, len(trainX))]

def encode_class_input(value):
    x = np.zeros((11))
    x[value] = 1
    return x

def decode_class_input(value):
    return np.where(value==1)[0][0]

def get_random_noise(batch_size, noise_size):
    random_values = np.random.randn(batch_size*noise_size)
    random_noise_batches = np.reshape(random_values, (batch_size, noise_size))
    return random_noise_batches

def get_fake_samples(generator_network, batch_size, noise_size):
    random_noise_batches = get_random_noise(batch_size, noise_size)
    fake_samples = generator_network.predict_on_batch(random_noise_batches)
    return fake_samples

def get_real_samples(batch_size):
    random_indices = np.random.choice(indices, size=batch_size)
    real_images = trainX[np.array(random_indices),:]
    real_classes = np.array([encode_class_input(x) for x in trainY[np.array(random_indices),]])
    return real_images, real_classes

def get_fake_classes(batch_size):
    fake_classes = []
    for i in range(batch_size):
        fake_class = encode_class_input(10)
        fake_classes.append(fake_class)
    return np.array(fake_classes)

```

```

def show_generator_results(generator_network):
    for k in range(9):
        plt.figure(figsize=(9, 6))
        random_noise_batches = get_random_noise(10, noise_size)
        fake_samples = generator_network.predict_on_batch(random_noise_batc
hes)
        for j in range(9):
            i = j
            plt.subplot(990 + 1 + j)
            plt.imshow(fake_samples[i,:,:,-1], cmap='gray_r')
            plt.axis('off')
        plt.show()
    return

```

We are now all set. Let's train the SGAN model now.

Step 8: Training SGAN

In this step, we will write the training iteration loop for our SGAN model. We plan to train our model for 500 epochs, with a batch size of 100, and noise vector size of 100. In each iteration, we will first update the weights of the discriminator network with mix batch of real and fake samples, along with the class labels. Then, we will freeze the discriminator weights and update the generator weights using only the noise inputs. Just like traditional GANs, we will pass the inverted labels (1 instead of 0) of fake samples while updating the generator weights, to make the discriminator believe that these are real samples and calculate the loss value.

The following python code implements the training loop for our SGAN model:

```

epochs = 500
batch_size = 100
steps = 500
noise_size = 100

```

```

for i in range(0, epochs):
    if (i%5 == 0):
        op = show_generator_results(generator_network)
        #print (op)
    for j in range(steps):
        fake_samples = get_fake_samples(generator_network, batch_size//2, noise_size)
        fake_classes = get_fake_classes(batch_size//2)
        real_samples, real_classes = get_real_samples(batch_size=batch_size//2)

        fake_y = np.zeros((batch_size//2, 1))
        real_y = np.ones((batch_size//2, 1))
        input_samples = np.vstack((fake_samples, real_samples))
        class_labels = np.vstack((fake_classes, real_classes))
        output_labels = np.vstack((fake_y, real_y))

        # Updating Discriminator weights
        discriminator_network.trainable=True
        loss_d = discriminator_network.train_on_batch(input_samples, [output_labels, class_labels])

        noise_batches = get_random_noise(batch_size, noise_size)
        ss_gan_input = noise_batches

        # Make the Discriminator believe that these are real samples and calculate
        # loss to train the generator
        ss_gan_output = np.ones((batch_size))

        # Updating Generator weights
        discriminator_network.trainable=False
        loss_g = ss_gan_model.train_on_batch(ss_gan_input, ss_gan_output)
        if j%50 == 0:

```

```
print ("Epoch:%.0f, Step:%.0f, D-Loss:%.3f, D-Acc:%.3f, D-Acc-  
Classification:%.3f, G-Loss:%.3f"\%\\
```

```
(i,j,loss_d[0],loss_d[3]*100,loss_d[4]*100,loss_g))
```

Following are the training logs of our setup:

Out [8]: Epoch:0, Step:0, D-Loss:1.805, D-Acc:80.000, D-Acc-Classification:4.000, G-Loss:0.784

Epoch:0, Step:50, D-Loss:0.388, D-Acc:96.000, D-Acc-Classification:84.000, G-Loss:3.382

Epoch:0, Step:100, D-Loss:0.287, D-Acc:90.000, D-Acc-Classification:87.000, G-Loss:2.939

Epoch:0, Step:150, D-Loss:0.283, D-Acc:92.000, D-Acc-Classification:89.000, G-Loss:3.567

Epoch:0, Step:200, D-Loss:0.232, D-Acc:94.000, D-Acc-Classification:89.000, G-Loss:3.301

Epoch:0, Step:250, D-Loss:0.370, D-Acc:95.000, D-Acc-Classification:85.000, G-Loss:3.579

Epoch:0, Step:300, D-Loss:0.216, D-Acc:97.000, D-Acc-Classification:88.000, G-Loss:3.899

Epoch:0, Step:350, D-Loss:0.250, D-Acc:95.000, D-Acc-Classification:86.000, G-Loss:3.823

Epoch:0, Step:400, D-Loss:0.238, D-Acc:99.000, D-Acc-Classification:88.000, G-Loss:4.513

Epoch:0, Step:450, D-Loss:0.192, D-Acc:98.000, D-Acc-Classification:92.000, G-Loss:3.608

Epoch:1, Step:0, D-Loss:0.194, D-Acc:97.000, D-Acc-Classification:94.000, G-Loss:4.898

Epoch:1, Step:50, D-Loss:0.248, D-Acc:98.000, D-Acc-Classification:86.000, G-Loss:3.116

....

....

....

Epoch:60, Step:0, D-Loss:0.348, D-Acc:87.000, D-Acc-Classification:85.000, G-Loss:2.961

Epoch:60, Step:50, D-Loss:0.340, D-Acc:81.000, D-Acc-Classification:86.000, G-Loss:2.093

Epoch:60, Step:100, D-Loss:0.336, D-Acc:84.000, D-Acc-Classification:81.000, G-Loss:2.042

Epoch:60, Step:150, D-Loss:0.390, D-Acc:79.000, D-Acc-Classification:82.000, G-Loss:1.578

Epoch:60, Step:200, D-Loss:0.401, D-Acc:82.000, D-Acc-Classification:82.000, G-Loss:2.509

Epoch:60, Step:250, D-Loss:0.284, D-Acc:90.000, D-Acc-Classification:88.000, G-Loss:2.651

Epoch:60, Step:300, D-Loss:0.321, D-Acc:85.000, D-Acc-Classification:86.000, G-Loss:2.866

Epoch:60, Step:350, D-Loss:0.399, D-Acc:83.000, D-Acc-Classification:85.000, G-Loss:1.956

Epoch:60, Step:400, D-Loss:0.370, D-Acc:85.000, D-Acc-Classification:85.000, G-Loss:2.083

Now that we have successfully trained our SGAN model, let's go ahead and verify the results on the test dataset.

Step 9: Results

Figure 6.11 shows the results of our SGAN model on Fashion MNIST dataset. We can see that the model is able to train really fast and starts showing good results even after just 10 or 15 epochs of training. At epoch 60, we were able to get very realistic results from the generator network.

Now that we have successfully implemented and verified the results of SGAN model. Let's move to the next experiment, where we will experiment with the Info GAN model.

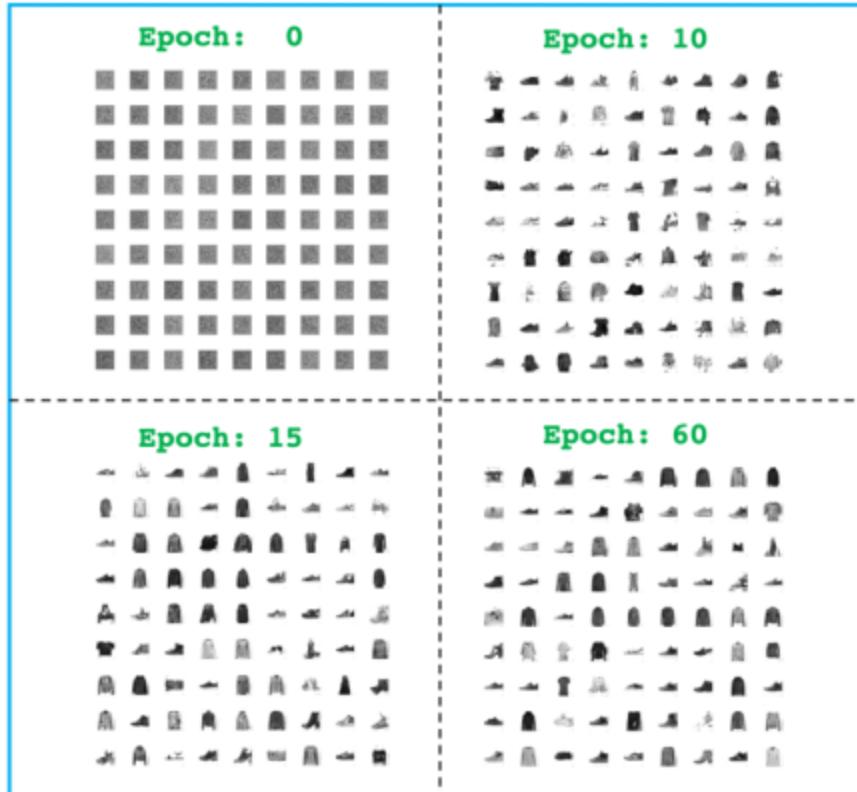


Figure 6.11: SGAN results on Fashion MNIST dataset

Let's implement the Info GAN model now.

INFO-GAN FOR MNIST HANDWRITTEN DIGITS

Objective

In this experiment, we will implement and train the Info GAN variant of conditional GAN. We will train it on the MNIST Handwritten Digits dataset.

This experiment has following steps:

- Import Useful Libraries
- Download and Show Data
- Data Normalization
- Define Generator Model
- Define Discriminator Model
- Define Combined Model: Info-GAN
- Utility Functions
- Training Info-GAN
- Results

Let's get started.

Step 1: Import Useful Libraries

Open a Jupyter Notebook with python kernel and import useful python packages in a cell. In this experiment, we will use ‘*numpy*’ for data manipulation, ‘*matplotlib*’ for plotting images and graphs, and *TensorFlow2* for implementing the model architecture.

Checkout the following snippet for importing libraries:

```
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
from tqdm import tqdm_notebook  
%matplotlib inline  
import tensorflow  
print(tensorflow.__version__)
```

Let's get the data now.

Step 2: Download and Show Data

In this step, we will download the MNIST Handwritten digits dataset and show few samples. Following python code loads the dataset and plots a few samples:

```
from tensorflow.keras.datasets import fashion_mnist, mnist
(trainX, trainY), (testX, testY) = mnist.load_data()
print('Training data shapes: X=%s, y=%s' % (trainX.shape, trainY.shape))

print('Testing data shapes: X=%s, y=%s' % (testX.shape, testY.shape))
for k in range(9):
    plt.figure(figsize=(9, 6))
    for j in range(9):
        i = np.random.randint(0, 10000)
        plt.subplot(990 + 1 + j)
        plt.imshow(trainX[i], cmap='gray_r')
        #plt.title(trainY[i])
        plt.axis('off')
    plt.show()
```

Figure 6.12 shows some samples from the dataset. MNIST Handwritten Digits dataset has 60k training images and 10k test images of size 28 x 28, and each image is gray scale and comes with a label value indicating the digit class.

```

Training data shapes: X=(60000, 28, 28), y=(60000,)
Testing data shapes: X=(10000, 28, 28), y=(10000,)

9 4 8 2 2 3 5 9 9
6 1 9 4 1 9 2 9 3
0 1 7 6 1 2 0 6 8
5 6 3 8 2 6 8 0 1
2 0 9 3 9 6 0 1 3
4 7 1 8 5 9 2 5 4
6 2 1 2 3 4 3 0 9
4 1 8 7 0 3 8 3 0
3 3 4 3 6 4 9 0 3

```

Figure 6.12: A few samples from the MNIST Handwritten Digits dataset

Let's prepare the data for model.

Step 3: Data Normalization

In this step, we will prepare the dataset for the model. We will first normalize pixel values of each image and restrict them to range [-1, 1] by subtracting and dividing with 127.5. We will also add a channel dimension to each image, as expected by the convolutional neural network layers.

The following python snippet prepares the dataset for model:

```

trainX = [(image-127.5)/127.5 for image in trainX]
testX = [(image-127.5)/127.5 for image in testX]
trainX = np.reshape(trainX, (60000, 28, 28, 1))
testX = np.reshape(testX, (10000, 28, 28, 1))
print (trainX.shape, testX.shape, trainY.shape, testY.shape)

```

Following is the shape of the final dataset:

```
(60000, 28, 28, 1) (10000, 28, 28, 1) (60000,) (10000,)
```

Our dataset is now ready. Let's define the model architecture now.

Step 4: Define Generator Model

In this step, we will define the generator network. In our setup, the generator accepts two inputs: a random noise vector of size 50 and a control

vector of size 10. These inputs are then concatenated and reshaped into a three-dimensional vector and then passed through the multiple layers of *Conv2DTranspose*, *ReLU* activation, and Batch Normalization with a momentum value of 0.8. The final layer has a ‘*tanh*’ activation function to restrict the pixel values of the generated image within range [-1, 1], similar to our training dataset.

The following python code implements the generator network:

```
random_input = tensorflow.keras.layers.Input(shape = 50)
control_input = tensorflow.keras.layers.Input(shape = 10)
x1 = tensorflow.keras.layers.Dense(64)(random_input)
#Class Input
x2 = tensorflow.keras.layers.Dense(32)(control_input)
x = tensorflow.keras.layers.concatenate([x1, x2])
x = tensorflow.keras.layers.Dense(5*5*64)(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.Reshape((5, 5, 64))(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=128, kernel_size=(3,3), strides=2)(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=128, kernel_size=(3,3), strides=2)(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=128, kernel_size=(3,3))(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=1, kernel_size=(4,4))(x)
```

```
generated_image = tensorflow.keras.layers.Activation('tanh')(x)
generator_network = tensorflow.keras.models.Model(inputs=
[random_input, control_input], outputs=generated_image)
generator_network.summary()
```

Following is the summary of the generator network. It roughly has 530k trainable parameters:

Out [4]: Model: "model"

Layer (type) Output Shape Param # Connected to

input_1 (InputLayer) [(None, 50)] 0

input_2 (InputLayer) [(None, 10)] 0

dense (Dense) (None, 64) 3264 input_1[0][0]

dense_1 (Dense) (None, 32) 352 input_2[0][0]

concatenate (Concatenate) (None, 96) 0 dense[0][0]

dense_1[0][0]

dense_2 (Dense) (None, 1600) 155200 concatenate[0][0]

activation (Activation) (None, 1600) 0 dense_2[0][0]

reshape (Reshape) (None, 5, 5, 64) 0 activation[0][0]

conv2d_transpose (Conv2DTranspo (None, 11, 11, 128) 73856 reshape[0][0]

activation_1 (Activation) (None, 11, 11, 128) 0 conv2d_transpose[0][0]

batch_normalization (BatchNorma (None, 11, 11, 128) 512 activation_1[0][0]

conv2d_transpose_1 (Conv2DTrans (None, 23, 23, 128) 147584 batch_normalization[0][0]

activation_2 (Activation) (None, 23, 23, 128) 0 conv2d_transpose_1[0][0]

batch_normalization_1 (BatchNor (None, 23, 23, 128) 512 activation_2[0][0]

conv2d_transpose_2 (Conv2DTrans (None, 25, 25, 128) 147584 batch_normalization_1[0][0]

activation_3 (Activation) (None, 25, 25, 128) 0 conv2d_transpose_2[0][0]

batch_normalization_2 (BatchNor (None, 25, 25, 128) 512 activation_3[0][0]

conv2d_transpose_3 (Conv2DTrans (None, 28, 28, 1) 2049 batch_normalization_2[0][0]

activation_4 (Activation) (None, 28, 28, 1) 0 conv2d_transpose_3[0][0]

Total params: 531,425

Trainable params: 530,657

Non-trainable params: 768

Let's work on the discriminator network now.

Step 5: Define Discriminator Model

In this step, we will define the discriminator network for our experiment. In our setup, the discriminator network accepts an image of size 28 x 28 x 1 as input. The input image is then passed through multiple layers of *Conv2D*, *LeakyReLU* activation and Batch normalization with a momentum value of 0.8. The discriminator network has two output heads: one validity output for classifying the image into real or fake, another for predicting the control values. The control output network, also termed as info network, has 10 neurons in its final layer (similar to the 10 control inputs of the generator

network), and a *softmax* activation function to generate multi-class probabilities for the control vector output.

The following python code implements the discriminator network:

```
image_input = tensorflow.keras.layers.Input(shape=(28, 28, 1))
x = tensorflow.keras.layers.Conv2D(filters=128, kernel_size=
(3,3), strides=2)(image_input)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.Conv2D(filters=128, kernel_size=
(3,3), strides=2)(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2D(filters=128, kernel_size=(3,3))(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2D(filters=64, kernel_size=(4,4))(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
features = tensorflow.keras.layers.Flatten()(x)
d_out = tensorflow.keras.layers.Dense(1)(features)
real_vs_fake_output = tensorflow.keras.layers.Activation('sigmoid')(d_out)

discriminator_network = tensorflow.keras.models.Model(inputs=image_i
nput, outputs=real_vs_fake_output)
info = tensorflow.keras.layers.Dense(10)(features)
info_output = tensorflow.keras.layers.Activation('softmax')(info)
info_network = tensorflow.keras.models.Model(inputs=image_input, outp
uts=info_output)
print (discriminator_network.summary())
print (info_network.summary())
```

Following are the summaries of the discriminator and the info network respectively. Both the networks have roughly 430k trainable parameters and most of the weights are shared:

Out [5]: Model: "model_7"

Layer (type)	Output Shape	Param #
=====		
input_6 (InputLayer)	[(None, 28, 28, 1)]	0
=====		
conv2d_12 (Conv2D)	(None, 13, 13, 128)	1280
=====		
leaky_re_lu_12 (LeakyReLU)	(None, 13, 13, 128)	0
=====		
conv2d_13 (Conv2D)	(None, 6, 6, 128)	147584
=====		
leaky_re_lu_13 (LeakyReLU)	(None, 6, 6, 128)	0
=====		
batch_normalization_15 (BatchNormalization)	(None, 6, 6, 128)	512
=====		
conv2d_14 (Conv2D)	(None, 4, 4, 128)	147584
=====		
leaky_re_lu_14 (LeakyReLU)	(None, 4, 4, 128)	0
=====		
batch_normalization_16 (BatchNormalization)	(None, 4, 4, 128)	512
=====		
conv2d_15 (Conv2D)	(None, 1, 1, 64)	131136
=====		
leaky_re_lu_15 (LeakyReLU)	(None, 1, 1, 64)	0
=====		
batch_normalization_17 (BatchNormalization)	(None, 1, 1, 64)	256
=====		
flatten_3 (Flatten)	(None, 64)	0
=====		
dense_12 (Dense)	(None, 1)	65
=====		
activation_14 (Activation)	(None, 1)	0

```
=====
Total params: 428,929
Trainable params: 428,289
Non-trainable params: 640
```

None

Model: "model_8"

Layer (type)	Output Shape	Param #
--------------	--------------	---------

input_6 (InputLayer)	[(None, 28, 28, 1)]	0
----------------------	---------------------	---

conv2d_12 (Conv2D)	(None, 13, 13, 128)	1280
--------------------	---------------------	------

leaky_re_lu_12 (LeakyReLU)	(None, 13, 13, 128)	0
----------------------------	---------------------	---

conv2d_13 (Conv2D)	(None, 6, 6, 128)	147584
--------------------	-------------------	--------

leaky_re_lu_13 (LeakyReLU)	(None, 6, 6, 128)	0
----------------------------	-------------------	---

batch_normalization_15 (BatchNormalization)	(None, 6, 6, 128)	512
---	-------------------	-----

conv2d_14 (Conv2D)	(None, 4, 4, 128)	147584
--------------------	-------------------	--------

leaky_re_lu_14 (LeakyReLU)	(None, 4, 4, 128)	0
----------------------------	-------------------	---

batch_normalization_16 (BatchNormalization)	(None, 4, 4, 128)	512
---	-------------------	-----

conv2d_15 (Conv2D)	(None, 1, 1, 64)	131136
--------------------	------------------	--------

leaky_re_lu_15 (LeakyReLU)	(None, 1, 1, 64)	0
----------------------------	------------------	---

batch_normalization_17 (BatchNormalization)	(None, 1, 1, 64)	256
---	------------------	-----

flatten_3 (Flatten)	(None, 64)	0
---------------------	------------	---

```
dense_13 (Dense) (None, 10) 650
```

```
activation_15 (Activation) (None, 10) 0
```

```
=====
```

```
Total params: 429,514
```

```
Trainable params: 428,874
```

```
Non-trainable params: 640
```

```
None
```

Let's now compile the networks. We will use '*binary_crossentropy*' loss function for the discriminator network and '*categorical_crossentropy*' loss function for the info network as it performs multi-class classification.

The following python snippet compiles both networks:

```
adam_optimizer = tensorflow.keras.optimizers.Adam(learning_rate=0.0001, beta_1=0.5)

discriminator_network.compile(loss='binary_crossentropy', optimizer=adam_optimizer, metrics=['accuracy'])

info_network.compile(loss='categorical_crossentropy', optimizer=adam_optimizer)
```

Let's now define the combined Info GAN model.

Step 6: Define Combined Model: Info-GAN

The combined Info GAN network has two input heads and two output heads. It accepts random noise and control vector as input, and passes them into the generator network. The output of the generator network is then passed as input to the discriminator network and the info network separately. Finally, outputs from both of these networks are also the outputs of the combined Info GAN model. Note that the discriminator network weights are kept frozen in this setup.

The following python code implements the combined Info GAN network:

```
discriminator_network.trainable=False

g_output = generator_network([random_input, control_input])
```

```
d_output = discriminator_network(g_output)
info_output = info_network(g_output)
info_gan_model = tensorflow.keras.models.Model(inputs = [random_input, control_input], outputs=[d_output, info_output])
info_gan_model.summary()
```

Following is the summary of the combined model:

Out [6]: Model: "model_9"

Layer (type) Output Shape Param # Connected to

input_1 (InputLayer) [(None, 50)] 0

input_2 (InputLayer) [(None, 10)] 0

model (Functional) (None, 28, 28, 1) 531425 input_1[0][0]

input_2[0][0]

model_7 (Functional) (None, 1) 428929 model[0][0]

model_8 (Functional) (None, 10) 429514 model[0][0]

Total params: 961,004

Trainable params: 531,307

Non-trainable params: 429,697

As the combined model has two outputs; we need two loss functions to compile it. We will use the same two loss functions here that we used for compiling the discriminator and the info network.

The following python snippet compiles the combined Info GAN model:

```
adam_optimizer = tensorflow.keras.optimizers.Adam(learning_rate=0.0001, beta_1=0.5)

info_gan_model.compile(loss=['binary_crossentropy', 'categorical_crossentropy'], optimizer=adam_optimizer)
```

Our model architecture is ready now. Let's work on the training piece.

Step 7: Utility Functions

In this step, we will define some utility functions that will help us in creating data batches for the model training. Specifically, we will define the utility functions for: generating random noise input batches along with random control inputs, generating batches of fake samples, and generating batches of real samples. In addition to these, we will also define a utility function for plotting the generator model results.

The following python code implements the utility functions for training:

```
indices = [i for i in range(0, len(trainX))]

def encode_control_input(value):
    x = np.zeros((10))
    x[value]=1
    return x

def decode_control_input(value):
    return np.where(value==1)[0][0]
```

```

def get_random_noise(batch_size, noise_size):
    random_values = np.random.randn(batch_size*noise_size)
    random_noise_batches = np.reshape(random_values, (batch_size, noise_size))
control_inputs = []
for i in range(batch_size):
    control_input = np.random.choice([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], size=1)
    control_inputs.append(encode_control_input(control_input))
control_inputs = np.array(control_inputs)
return random_noise_batches, control_inputs

def get_fake_samples(generator_network, batch_size, noise_size):
    random_noise_batches, control_inputs= get_random_noise(batch_size, noise_size)
    fake_samples = generator_network.predict_on_batch([random_noise_batches, control_inputs])
return fake_samples

def get_real_samples(batch_size):
    random_indices = np.random.choice(indices, size=batch_size)
    real_images = trainX[np.array(random_indices),:]
return real_images

def show_generator_results(generator_network):
for k in range(10):
    random_noise_batches, _ = get_random_noise(10, noise_size)
    control_inputs = []
for bs in range(10):
    control_inputs.append(encode_control_input(k))
    control_inputs = np.array(control_inputs)
    fake_samples = generator_network.predict_on_batch([random_noise_batches, control_inputs])

```

```

plt.figure(figsize=(9, 6))
for j in range(9):
    i = j
    plt.subplot(990 + 1 + j)
    plt.imshow(fake_samples[i,:,:,-1], cmap='gray_r')
    #plt.title(decode_control_input(control_inputs[i]))
    plt.axis('off')
    plt.show()
return

```

We are now all set to start training our Info GAN model.

Step 8: Training Info-GAN

In this step, we will write the training iteration loop for our Info GAN model. We plan to train our model for 500 epochs, with a batch size of 100, noise vector size of 50, and control input size of 10. In each iteration, we will first update the weights of the discriminator network with a mix batch of real and fake samples. Then, we will freeze the discriminator weights and update the info GAN weights using the noise inputs and control inputs. Just like traditional GANs, we will pass the inverted labels (1 instead of 0) of fake samples while updating the generator weights, to make the discriminator believe that these are real samples and calculate the loss value.

The following python code implements the training loop for our Info GAN model:

```

epochs = 500
batch_size = 100
steps = 500
noise_size = 50
for i in range(0, epochs):
    if (i%5 == 0):
        op = show_generator_results(generator_network)
        #print (op)

```

```

for j in range(steps):
    fake_samples = get_fake_samples(generator_network, batch_size//2, noise_size)
    real_samples = get_real_samples(batch_size=batch_size//2)
    fake_y = np.zeros((batch_size//2, 1))
    real_y = np.ones((batch_size//2, 1))
    input_samples = np.vstack((fake_samples, real_samples))
    output_labels = np.vstack((fake_y, real_y))
    # Updating Discriminator weights
    discriminator_network.trainable=True
    loss_d = discriminator_network.train_on_batch(input_samples, output_labels)
    noise_batches, control_inputs = get_random_noise(batch_size, noise_size)
    info_gan_input = [noise_batches, control_inputs]
    # Make the Discriminator believe that these are real samples and calculate
    # loss to train the generator
    d_output = np.ones((batch_size))
    info_output = control_inputs
    info_gan_output = [d_output, info_output]
    # Updating Generator weights
    discriminator_network.trainable=False
    loss_g = info_gan_model.train_on_batch(info_gan_input, info_gan_output)
    if j%50 == 0:
        print ("Epoch:%.0f, Step:%.0f, D-Loss:%.3f, D-Acc:%.3f, G-
Loss:%.3f, Info-Loss:%.3f"%  

(i,j,loss_d[0],loss_d[1]*100,loss_g[0], loss_g[1]))

```

Following are the training logs:

Out [8]: Epoch:0, Step:0, D-Loss:1.389, D-Acc:10.000, G-Loss:2.999, Info-Loss:0.684

Epoch:0, Step:50, D-Loss:0.079, D-Acc:97.000, G-Loss:6.304, Info-Loss:3.760

Epoch:0, Step:100, D-Loss:0.147, D-Acc:93.000, G-Loss:7.117, Info-Loss:4.507

Epoch:0, Step:150, D-Loss:0.052, D-Acc:99.000, G-Loss:7.862, Info-Loss:5.318

Epoch:0, Step:200, D-Loss:0.088, D-Acc:96.000, G-Loss:8.136, Info-Loss:5.443

Epoch:0, Step:250, D-Loss:0.130, D-Acc:94.000, G-Loss:7.268, Info-Loss:4.967

Epoch:0, Step:300, D-Loss:0.095, D-Acc:96.000, G-Loss:8.229, Info-Loss:5.858

Epoch:0, Step:350, D-Loss:0.159, D-Acc:94.000, G-Loss:6.439, Info-Loss:4.223

Epoch:0, Step:400, D-Loss:0.088, D-Acc:97.000, G-Loss:6.769, Info-Loss:4.659

Epoch:0, Step:450, D-Loss:0.117, D-Acc:95.000, G-Loss:6.056, Info-Loss:3.898

Epoch:1, Step:0, D-Loss:0.083, D-Acc:98.000, G-Loss:6.766, Info-Loss:4.861

Epoch:1, Step:50, D-Loss:0.201, D-Acc:89.000, G-Loss:5.915, Info-Loss:3.985

Epoch:1, Step:100, D-Loss:0.184, D-Acc:93.000, G-Loss:5.747, Info-Loss:4.000

Epoch:1, Step:150, D-Loss:0.140, D-Acc:95.000, G-Loss:6.596, Info-Loss:5.074

Epoch:1, Step:200, D-Loss:0.192, D-Acc:93.000, G-Loss:7.923, Info-Loss:6.357

...

...

...

Epoch:57, Step:300, D-Loss:0.572, D-Acc:69.000, G-Loss:0.380, Info-Loss:0.369

Epoch:57, Step:350, D-Loss:0.622, D-Acc:70.000, G-Loss:1.138, Info-Loss:1.109

Epoch:57, Step:400, D-Loss:0.510, D-Acc:80.000, G-Loss:3.072, Info-Loss:3.060

Epoch:57, Step:450, D-Loss:0.442, D-Acc:77.000, G-Loss:4.840, Info-Loss:4.837

Epoch:58, Step:0, D-Loss:0.423, D-Acc:82.000, G-Loss:2.579, Info-Loss:2.556

Epoch:58, Step:50, D-Loss:0.475, D-Acc:82.000, G-Loss:0.132, Info-Loss:0.108

Epoch:58, Step:100, D-Loss:0.383, D-Acc:83.000, G-Loss:0.035, Info-Loss:0.019

Epoch:58, Step:150, D-Loss:0.486, D-Acc:76.000, G-Loss:0.326, Info-Loss:0.307

Now that we have successfully trained the Info GAN model, let's check out the results.

Step 9: Results

Figure 6.13 shows the results of our Info GAN generator network. We can see that the network learns very fast and starts generating results even after 10 epochs. At epoch 60, the results start looking very realistic. Interesting thing to notice here is that the class of the generated digits in a single row is same. This is possible due to the control input variables without even needing the actual labels. However, it remains a manual task to identify which control input corresponds to which class of the output. Anyway, this is a great way to control the outputs without even needing the labelled data.

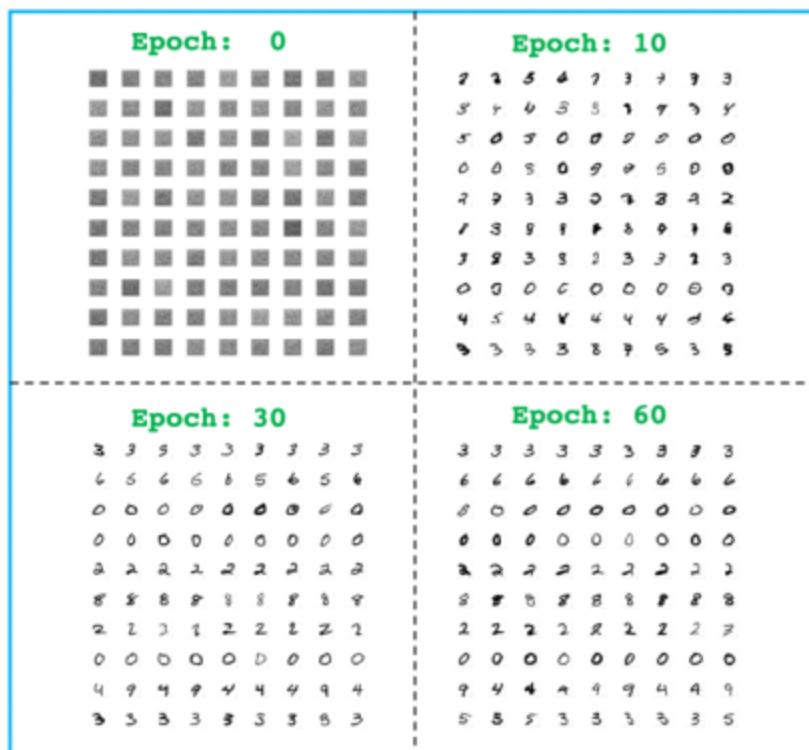


Figure 6.13: Handwritten Digit images generated using the Info GAN network

We have just seen the capability of Info GAN network of learning some output control without even needing the labelled data. Next, let's implement the ACGAN variant of conditional GANs.

AUXILIARY CLASSIFIER OR ACGAN ON FASHION MNIST

Objective

In this experiment, we will implement and train ACGAN model on Fashion MNIST dataset and verify its capabilities.

This experiment has the following steps:

- Import Useful Libraries
- Download and Show Data
- Data Normalization
- Define Generator Model
- Define Discriminator Model
- Define Combined Model: ACGAN
- Utility Functions
- Training ACGAN
- Results

Let's get started.

Step 1: Import Useful Libraries

Open a Jupyter Notebook with python kernel and import useful python packages in a cell. In this experiment, we will use '*numpy*' for data manipulation, '*matplotlib*' for plotting images and graphs, and *TensorFlow2* for implementing the model architecture.

Checkout the following snippet for importing libraries:

```
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
from tqdm import tqdm_notebook  
%matplotlib inline  
import tensorflow  
print(tensorflow.__version__)
```

Let's get the data now.

Step 2: Download and Show Data

In this step, we will download and display the Fashion MNIST dataset. The following python code loads the dataset and plots few samples:

```
from tensorflow.keras.datasets import fashion_mnist, mnist  
(trainX, trainY), (testX, testY) = fashion_mnist.load_data()  
print('Training data shapes: X=%os, y=%os' % (trainX.shape, trainY.shape))  
  
print('Testing data shapes: X=%os, y=%os' % (testX.shape, testY.shape))
```

for k in range(9):

```
    plt.figure(figsize=(9, 6))  
    for j in range(9):  
        i = np.random.randint(0, 10000)  
        plt.subplot(990 + 1 + j)  
        plt.imshow(trainX[i], cmap='gray_r')  
        #plt.title(trainY[i])  
        plt.axis('off')  
    plt.show()
```

Figure 6.14 shows some samples from the dataset. Fashion MNIST dataset has 60k training images and 10k test images of size 28 x 28, and each sample is gray scale image and comes with a label value indicating the class value.



Figure 6.14: Few samples from the Fashion MNIST dataset

Let's prepare the data.

Step 3: Data Normalization

In this step, we will prepare the dataset for the model. We will first normalize pixel values of each image and restrict them to range [-1, 1] by subtracting and dividing with 127.5. We will also add a channel dimension to each image, as expected by the convolutional neural network layers.

The following python snippet prepares the dataset for model:

```
trainX = [(image-127.5)/127.5 for image in trainX]
testX = [(image-127.5)/127.5 for image in testX]
trainX = np.reshape(trainX, (60000, 28, 28, 1))
testX = np.reshape(testX, (10000, 28, 28, 1))
print (trainX.shape, testX.shape, trainY.shape, testY.shape)
```

Following is the shape of the final dataset:

```
(60000, 28, 28, 1) (10000, 28, 28, 1) (60000,) (10000,)
```

Let's define the model architecture now.

Step 4: Define Generator Model

In this step, we will define the generator network. In our setup, the generator accepts two inputs: a random noise vector of size 100 and a class input of size 10. These inputs are then concatenated and reshaped into a three-dimensional vector and then passed through the multiple layers of *Conv2DTranspose*, *ReLU* activation, and Batch Normalization with a momentum value of 0.8. The final layer has a ‘*tanh*’ activation function to restrict the pixel values of the generated image within range [-1, 1], similar to our training dataset.

The following python code implements the generator network:

```
random_input = tensorflow.keras.layers.Input(shape = 100)
class_input = tensorflow.keras.layers.Input(shape = 10)
x1 = tensorflow.keras.layers.Dense(128)(random_input)
#Class Input
x2 = tensorflow.keras.layers.Dense(32)(class_input)
x = tensorflow.keras.layers.concatenate([x1, x2])
x = tensorflow.keras.layers.Dense(5*5*64)(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Reshape((5, 5, 64))(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=128, kernel_size=(3,3), strides=2)(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=128, kernel_size=(3,3), strides=2)(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=128, kernel_size=(3,3))(x)
x = tensorflow.keras.layers.Activation('relu')(x)
```

```
x = tensorflow.keras.layers.Conv2DTranspose(filters=1, kernel_size=(4,4))  
(x)
```

```
generated_image = tensorflow.keras.layers.Activation('tanh')(x)  
generator_network = tensorflow.keras.models.Model(inputs=[random_input, class_input], outputs=generated_image)
```

```
generator_network.summary()
```

Following is the summary of the generator network. It roughly has 650k trainable parameters:

Out [4]: Model: "model"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 100)]	0	
input_2 (InputLayer)	[(None, 10)]	0	
dense (Dense)	(None, 128)	12928	input_1[0][0]
dense_1 (Dense)	(None, 32)	352	input_2[0][0]

input_1 (InputLayer)	[(None, 100)]	0	
----------------------	---------------	---	--

input_2 (InputLayer)	[(None, 10)]	0	
----------------------	--------------	---	--

dense (Dense)	(None, 128)	12928	input_1[0][0]
---------------	-------------	-------	---------------

dense_1 (Dense)	(None, 32)	352	input_2[0][0]
-----------------	------------	-----	---------------

concatenate (Concatenate) (None, 160) 0 dense[0][0]

dense_1[0][0]

dense_2 (Dense) (None, 1600) 257600 concatenate[0][0]

activation (Activation) (None, 1600) 0 dense_2[0][0]

batch_normalization (BatchNorma (None, 1600) 6400 activation[0][0]

reshape (Reshape) (None, 5, 5, 64) 0 batch_normalization[0][0]

conv2d_transpose (Conv2DTranspo (None, 11, 11, 128) 73856 reshape[0][0]

activation_1 (Activation) (None, 11, 11, 128) 0 conv2d_transpose[0][0]

batch_normalization_1 (BatchNor (None, 11, 11, 128) 512 activation_1[0][0]

conv2d_transpose_1 (Conv2DTrans (None, 23, 23, 128) 147584 batch_normalization_1[0][0]

activation_2 (Activation) (None, 23, 23, 128) 0 conv2d_transpose_1[0][0]

batch_normalization_2 (BatchNor (None, 23, 23, 128) 512 activation_2[0][0]

conv2d_transpose_2 (Conv2DTrans (None, 25, 25, 128) 147584 batch_normalization_2[0][0]

activation_3 (Activation) (None, 25, 25, 128) 0 conv2d_transpose_2[0][0]

conv2d_transpose_3 (Conv2DTrans (None, 28, 28, 1) 2049 activation_3[0][0]

activation_4 (Activation) (None, 28, 28, 1) 0 conv2d_transpose_3[0][0]

Total params: 649,377

Trainable params: 645,665

Non-trainable params:

3,712

Let's define the discriminator network now.

Step 5: Define Discriminator Model

In this step, we will define the discriminator network for our experiment. In our setup, the discriminator network accepts an image of size 28 x 28 x 1 as input. The input image is then passed through multiple layers of *Conv2D*, *LeakyReLU* activation and Batch normalization with a momentum value of

0.8. The discriminator network has two output heads: one validity output for classifying the image into real or fake, another for predicting the class. The class output has 10 neurons in its final layer (similar to the 10 class inputs of the generator network), and a *softmax* activation function to generate multi-class probabilities for the control vector output.

The following python code implements the discriminator network:

```
image_input = tensorflow.keras.layers.Input(shape=(28, 28, 1))
x = tensorflow.keras.layers.Conv2D(filters=128, kernel_size=
(3,3), strides=2)(image_input)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.Conv2D(filters=128, kernel_size=
(3,3), strides=2)(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2D(filters=128, kernel_size=(3,3))(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Conv2D(filters=128, kernel_size=(4,4))(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
features = tensorflow.keras.layers.Flatten()(x)
d_out1 = tensorflow.keras.layers.Dense(1)(features)
real_vs_fake_output = tensorflow.keras.layers.Activation('sigmoid')
(d_out1)
d_out2 = tensorflow.keras.layers.Dense(10)(features)
class_output = tensorflow.keras.layers.Activation('softmax')(d_out2)
discriminator_network = tensorflow.keras.models.Model(inputs=image_i
nput, outputs=[real_vs_fake_output, class_output])
discriminator_network.summary()
```

Following is the summary of the discriminator network. It roughly has 560k trainable parameters.

Out [5]: Model: "model_1"

Layer (type) Output Shape Param # Connected to

input_3 (InputLayer) [(None, 28, 28, 1)] 0

conv2d (Conv2D) (None, 13, 13, 128) 1280 input_3[0][0]

leaky_re_lu (LeakyReLU) (None, 13, 13, 128) 0 conv2d[0][0]

conv2d_1 (Conv2D) (None, 6, 6, 128) 147584 leaky_re_lu[0][0]

leaky_re_lu_1 (LeakyReLU) (None, 6, 6, 128) 0 conv2d_1[0][0]

batch_normalization_3 (BatchNor (None, 6, 6, 128) 512 leaky_re_lu_1[0][0]

conv2d_2 (Conv2D) (None, 4, 4, 128) 147584 batch_normalization_3[0][0]

leaky_re_lu_2 (LeakyReLU) (None, 4, 4, 128) 0 conv2d_2[0][0]

batch_normalization_4 (BatchNor (None, 4, 4, 128) 512 leaky_re_lu_2[0][0]

conv2d_3 (Conv2D) (None, 1, 1, 128) 262272 batch_normalization_4[0][0]

leaky_re_lu_3 (LeakyReLU) (None, 1, 1, 128) 0 conv2d_3[0][0]

batch_normalization_5 (BatchNor (None, 1, 1, 128) 512 leaky_re_lu_3[0][0]

flatten (Flatten) (None, 128) 0 batch_normalization_5[0][0]

dense_3 (Dense) (None, 1) 129 flatten[0][0]

dense_4 (Dense) (None, 10) 1290 flatten[0][0]

activation_5 (Activation) (None, 1) 0 dense_3[0][0]

```
activation_6 (Activation) (None, 10) 0 dense_4[0][0]
```

```
=====
=====
```

Total params: 561,675

Trainable params: 560,907

Non-trainable params: 768

None

As the discriminator network has two output heads; we need two loss functions to compile this model. We will use '*binary_crossentropy*' loss for the image validity output (real vs. fake) as it is a binary classification problem. And, we will use '*categorical_crossentropy*' loss for the class output as it is a multi-class classification problem. Each loss function has an equal weight of 0.5.

The following python code compiles the discriminator network:

```
adam_optimizer = tensorflow.keras.optimizers.Adam(learning_rate=0.00  
01, beta_1=0.5)
```

```
discriminator_network.compile(loss=  
['binary_crossentropy', 'categorical_crossentropy'], optimizer=adam_optimize  
r, metrics=['accuracy'], loss_weights=[0.5, 0.5])
```

Let's now define the combined model.

Step 6: Define Combined Model: ACGAN

The combined ACGAN model accepts two inputs: a random noise vector and a class input vector. These inputs are first passed into the generator network. The output of the generator network is further passed into the discriminator network. The output of the discriminator network is also the

output of the combined ACGAN model. In this combined model setup, the weights of the discriminator network are kept frozen.

The following python code defines the combined ACGAN model:

```
discriminator_network.trainable=False  
g_output = generator_network([random_input, class_input])  
d_output = discriminator_network(g_output)  
ac_gan_model = tensorflow.keras.models.Model(inputs = [random_input,  
class_input], outputs=d_output)  
ac_gan_model.summary()
```

Following is the summary of the combined ACGAN model:

Out [6]: Model: "model_2"

Layer (type) Output Shape Param # Connected to

input_1 (InputLayer) [(None, 100)] 0

input_2 (InputLayer) [(None, 10)] 0

model (Functional) (None, 28, 28, 1) 649377 input_1[0][0]

input_2[0][0]

model_1 (Functional) [(None, 1), (None, 1) 561675 model[0][0]

```
=====
=====
Total params: 1,211,052
```

```
Trainable params: 645,665
```

```
Non-trainable params: 565,387
```

We can now compile the combined ACGAN model with similar parameters that we used for the discriminator network.

The following python code compiles the ACGAN:

```
ac_gan_model.compile(loss=
['binary_crossentropy', 'categorical_crossentropy'], optimizer=adam_optimizer)
```

Our dataset and model are set now. Let's work on the training piece.

Step 7: Utility Functions

In this step, we will define some utility functions that will help us in creating data batches for the model training. Specifically, we will define the utility functions for: generating random noise input batches along with random class inputs, generating batches of fake samples along with random class inputs, and generating batches of real samples along with real class labels. In addition to these, we will also define a utility function for plotting the generator model results.

The following python code implements the utility functions for training:

```
indices = [i for i in range(0, len(trainX))]

def encode_class_input(value):
    x = np.zeros((10))
    x[value]=1
    return x

def decode_class_input(value):
```

```

return np.where(value==1)[0][0]

def get_random_noise(batch_size, noise_size):
    random_values = np.random.randn(batch_size*noise_size)
    random_noise_batches = np.reshape(random_values, (batch_size, noise_size))

    class_inputs = []
    for i in range(batch_size):
        class_input = np.random.choice([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], size=1)
        class_inputs.append(encode_class_input(class_input))
    class_inputs = np.array(class_inputs)

    return random_noise_batches, class_inputs

def get_fake_samples(generator_network, batch_size, noise_size):
    random_noise_batches, class_inputs= get_random_noise(batch_size, noise_size)
    fake_samples = generator_network.predict_on_batch([random_noise_batches, class_inputs])

    return fake_samples, class_inputs

def get_real_samples(batch_size):
    random_indices = np.random.choice(indices, size=batch_size)
    real_images = trainX[np.array(random_indices),:]
    real_classes = np.array([encode_class_input(x) for x in trainY[np.array(random_indices),]])

    return real_images, real_classes

def show_generator_results(generator_network):
    for k in range(10):
        random_noise_batches, _ = get_random_noise(10, noise_size)
        class_inputs = []
        for bs in range(10):
            class_inputs.append(encode_class_input(k))

```

```

class_inputs = np.array(class_inputs)
fake_samples = generator_network.predict_on_batch([random_noise_batches, class_inputs])
plt.figure(figsize=(9, 6))
for j in range(9):
    i = j
    plt.subplot(990 + 1 + j)
    plt.imshow(fake_samples[i,:,:,-1], cmap='gray_r')
    plt.axis('off')
# plt.title(decode_class_input(class_inputs[i]))
plt.show()
return

```

We are now all set to start the training of our ACGAN model.

Step 8: Training ACGAN

In this step, we will write the training iteration loop for our ACGAN model. We plan to train our model for 500 epochs, with a batch size of 100, noise vector size of 100, and class input size of 10. In each iteration, we will first update the weights of the discriminator network with a mix batch of real and fake samples along with class labels. Then, we will freeze the discriminator weights and update the info GAN weights using the noise inputs and random class inputs. Just like traditional GANs, we will pass the inverted labels (1 instead of 0) of fake samples while updating the generator weights, to make the discriminator believe that these are real samples and calculate the loss value.

The following python code implements the training loop for our ACGAN model:

```

epochs = 500
batch_size = 100
steps = 500
noise_size = 100

```

```

for i in range(0, epochs):
    if (i%5 == 0):
        op = show_generator_results(generator_network)
        #print (op)
    for j in range(steps):
        fake_samples, fake_classes = get_fake_samples(generator_network, batch_size//2, noise_size)
        real_samples, real_classes = get_real_samples(batch_size=batch_size//2)

        fake_y = np.zeros((batch_size//2, 1))
        real_y = np.ones((batch_size//2, 1))
        input_samples = np.vstack((fake_samples, real_samples))
        class_labels = np.vstack((fake_classes, real_classes))
        output_labels = np.vstack((fake_y, real_y))

        # Updating Discriminator weights
        discriminator_network.trainable=True
        loss_d = discriminator_network.train_on_batch(input_samples, [output_labels, class_labels])

        noise_batches, class_inputs = get_random_noise(batch_size, noise_size)
        ac_gan_input = [noise_batches, class_inputs]
        # Make the Discriminator believe that these are real samples and calculate
        # loss to train the generator
        d_output = np.ones((batch_size))
        ac_gan_output = [d_output, class_inputs]

        # Updating Generator weights
        discriminator_network.trainable=False
        loss_g = ac_gan_model.train_on_batch(ac_gan_input, ac_gan_output)
        if j%50 == 0:

```

```
print ("Epoch:%.0f, Step:%.0f, D-Loss:%.3f, D-Acc:%.3f, D-Acc-  
Classification:%.3f, G-Loss:%.3f"\%\\  
(i,j,loss_d[0],loss_d[3]*100,loss_d[4]*100,loss_g[0]))
```

Following are the training logs:

Out [8]: Epoch:0, Step:0, D-Loss:1.833, D-Acc:25.000, D-Acc-Classification:14.000, G-Loss:2.978

Epoch:0, Step:50, D-Loss:0.861, D-Acc:96.000, D-Acc-Classification:43.000, G-Loss:5.147

Epoch:0, Step:100, D-Loss:0.744, D-Acc:92.000, D-Acc-Classification:56.000, G-Loss:4.786

Epoch:0, Step:150, D-Loss:0.601, D-Acc:93.000, D-Acc-Classification:64.000, G-Loss:4.144

Epoch:0, Step:200, D-Loss:0.516, D-Acc:87.000, D-Acc-Classification:76.000, G-Loss:3.450

Epoch:0, Step:250, D-Loss:0.361, D-Acc:93.000, D-Acc-Classification:84.000, G-Loss:4.271

Epoch:0, Step:300, D-Loss:0.379, D-Acc:96.000, D-Acc-Classification:77.000, G-Loss:4.378

Epoch:0, Step:350, D-Loss:0.300, D-Acc:93.000, D-Acc-Classification:83.000, G-Loss:2.850

Epoch:0, Step:400, D-Loss:0.303, D-Acc:95.000, D-Acc-Classification:87.000, G-Loss:3.212

Epoch:0, Step:450, D-Loss:0.269, D-Acc:98.000, D-Acc-Classification:88.000, G-Loss:5.276

Epoch:1, Step:0, D-Loss:0.318, D-Acc:90.000, D-Acc-Classification:87.000, G-Loss:2.484

Epoch:1, Step:50, D-Loss:0.356, D-Acc:94.000, D-Acc-Classification:86.000, G-Loss:3.667

Epoch:1, Step:100, D-Loss:0.260, D-Acc:100.000, D-Acc-Classification:88.000, G-Loss:3.246

Epoch:1, Step:150, D-Loss:0.238, D-Acc:98.000, D-Acc-Classification:88.000, G-Loss:3.184

.....

.....

.....

.....

Epoch:50, Step:0, D-Loss:0.383, D-Acc:69.000, D-Acc-Classification:95.000, G-Loss:1.465

Epoch:50, Step:50, D-Loss:0.236, D-Acc:80.000, D-Acc-Classification:99.000, G-Loss:1.754

Epoch:50, Step:100, D-Loss:0.279, D-Acc:75.000, D-Acc-Classification:97.000, G-Loss:1.256

Epoch:50, Step:150, D-Loss:0.296, D-Acc:73.000, D-Acc-Classification:94.000, G-Loss:1.410

Epoch:50, Step:200, D-Loss:0.290, D-Acc:72.000, D-Acc-Classification:99.000, G-Loss:1.674

Epoch:50, Step:250, D-Loss:0.272, D-Acc:76.000, D-Acc-Classification:98.000, G-Loss:1.229

We have successfully trained the ACGAN network now. Let's check out some results.

Step 9: Results

Figure 6.15 shows the results of our ACGAN generator network. We can see that the network learns very fast and starts generating good results just after 5 epochs of training. At epoch 20, results start looking very realistic. At epoch 50, we stop the training as the results look entirely realistic. Interesting thing to notice here is that the class of the generated fashion images in a single row is same. This is possible due to the class input variables that our ACGAN accepts as input. The quality and variety of the results generated though ACGAN is amazing.

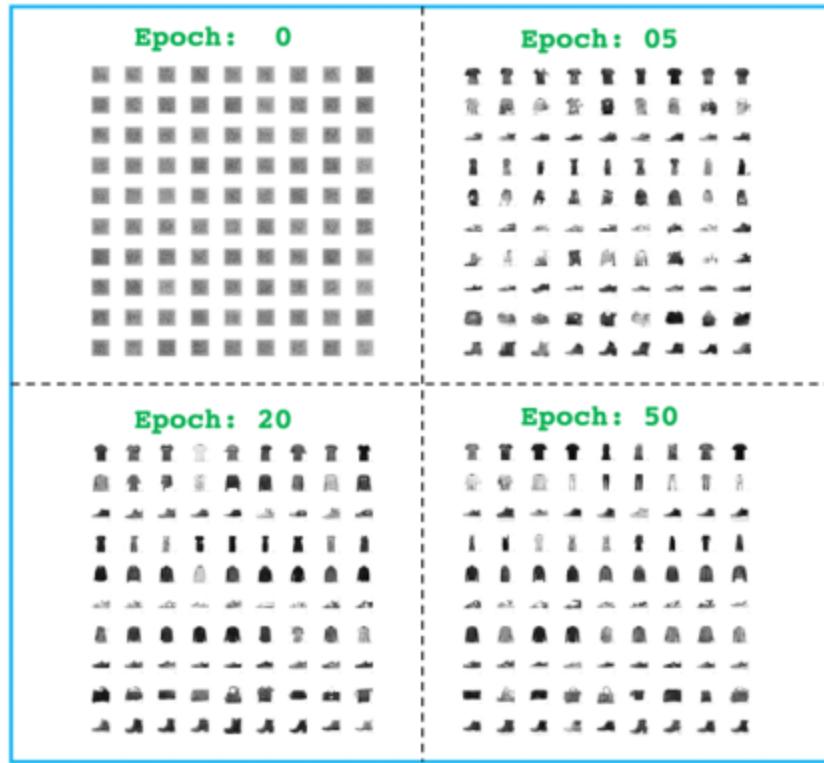


Figure 6.15: Fashion MNIST samples generated through ACGAN model

We have just completed four implementation experiments for CGAN, SGAN, Info GAN and ACGAN variants of the conditional GANs and verified the great results.

1. Conclusion

This skill has covered the conditional GANs in details, especially focusing on four popular variants: CGAN, SGAN, Info GAN and ACGAN. After reading this skill, we should be able to answer the following questions about conditional GANs:

- What are conditional GANs? How are they different from the ordinary GANs?

- How to use a GAN based model for semi-supervised learning and improve accuracy on the classification tasks.
- What are the information maximizing generative adversarial networks? How do they learn disentangled representations in an unsupervised manner?
- How ACGANs are different from other conditional GANs, what are the advantages of using them over other conditional GANs?

In the next skill, we will learn about using different loss functions for training GANs to achieve even better performance.

End of Skill-6

SKILL 7

Better Loss Functions

Generative Adversarial Network, or GAN for short, has a simple architecture where two networks (the generator and the discriminator) are trained simultaneously. The discriminator network is trained as a traditional classifier as its main purpose is to distinguish fake images from the real ones. Whereas, the generator network uses discriminator network as its loss function. So, the loss function for the generator network is implicit.

In this skill, we will talk briefly about the challenges of the traditional GAN's loss function and also introduce some new loss functions that can be utilized for training GANs. After reading this skill, the readers should feel confident about introducing new loss functions within their GAN experiments.

This skill covers the following main topics:

- Why other loss functions?
- What is a Wasserstein GAN (WGAN)?
- What is least squares GAN (LSGAN)?
- Improving WGAN using gradient penalty (WGAN-GP)
- Conclusion

Let's get started.

1. Why other loss functions?

Finding better loss functions for training GANs is an active area of research. Over the years many different loss functions have been introduced and evaluated. In this skill, we will talk about two such popular loss functions: the Wasserstein loss and the least squares loss.

In this skill, we will train the following three popular GAN variants using the aforementioned new loss functions.

- Wasserstein GAN or WGAN
- Least Square GAN or LSGAN
- Wasserstein GAN with gradient penalty or WGAN-GP

Let's learn about these three popular variants.

1. What is a Wasserstein GAN (WGAN)?

Wasserstein GAN, or WGAN for short, is a variant of GAN architecture that improves the stability of GAN training by introducing a new loss function called the Wasserstein loss.

WGAN was introduced by *Martin Arjovsky et. al.* in their 2017 paper titled '*Wasserstein GAN*'. The Wasserstein loss has a deep mathematical significance but implementing and using it in a network is quite straightforward.

The discriminator network in a traditional GAN is binary classifier that distinguishes the fake images (generated images) from the real ones and the generator network uses this feedback to improve itself. In case of WGAN, the discriminator network is replaced by a critic network. Instead of classifying images into real or fake, the critic network calculates the realness or fakeness of an image by giving them a score.

The main focus of the generator network in a WGAN is to reduce the distance between the observed distribution of real images and the distribution of generated images (or fake images). This distance can be measured using different ways such as Kullback-Leibler (KL) divergence, Jensen-Shannon (JS) divergence and so on. Another way of measuring distance between two distributions is the Earth-Mover (EM) distance. EM distance is also termed as Wasserstein distance.

In the WGAN paper, authors claim that the critic network can be used to approximate the Wasserstein distance between two distributions. As Wasserstein distance is continuous and differentiable it keeps providing the gradient even when critic is well trained, thus always improving the generator network. The traditional GAN loss however, may fail to provide useful gradient when the discriminator is well-trained. Another important point

about WGAN is that the Wasserstein loss value is correlated with the quality of generated outputs.

The fact that Wasserstein loss function is continuous and differentiable almost everywhere, it gives us the flexibility to train the critic till optimality. So, the more we train the critic, better gradient we get to improve the generator. In case of traditional loss function, the discriminator learns very quickly and thus fails to provide reliable gradient after some time. However, the critic converges to a linear function and thus provides a clean gradient everywhere. Training critic till optimality also doesn't let the generator to fall into collapse modes.

Now that, we have a basic understanding of WGAN, let's look into the implementation details.

2.1 Implementation Details of WGAN

Wasserstein loss function has the objective of increasing the gap between the scores of the real and the fake (generated) images. A simple way to look at the loss function is as follows:

- *Critic Loss = [average score on real images] – [average score on fake images]*
- *Generator Loss = – [average critic score on fake images]*

Here, the average critic scores are calculated on the mini batches of real and fake images. As the loss of generator has a negative sign, it will encourage the critic to output bigger scores for fake images so that the loss of generator becomes smaller (due to negative sign, as bigger negative numbers are actually smaller or, -10 is bigger than -20).

WGAN paper also provides the information about writing the training loop. This can help us in defining our training procedure with recommended hyperparameters. See Figure 7.1 for the training loop details from the original paper.

Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

Require: : α , the learning rate. c , the clipping parameter. m , the batch size. n_{critic} , the number of iterations of the critic per generator iteration.

Require: : w_0 , initial critic parameters. θ_0 , initial generator's parameters.

```
1: while  $\theta$  has not converged do
2:   for  $t = 0, \dots, n_{\text{critic}}$  do
3:     Sample  $\{x^{(i)}\}_{i=1}^m \sim \mathbb{P}_r$  a batch from the real data.
4:     Sample  $\{z^{(i)}\}_{i=1}^m \sim p(z)$  a batch of prior samples.
5:      $g_w \leftarrow \nabla_w [\frac{1}{m} \sum_{i=1}^m f_w(x^{(i)}) - \frac{1}{m} \sum_{i=1}^m f_w(g_\theta(z^{(i)}))]$ 
6:      $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$ 
7:      $w \leftarrow \text{clip}(w, -c, c)$ 
8:   end for
9:   Sample  $\{z^{(i)}\}_{i=1}^m \sim p(z)$  a batch of prior samples.
10:   $g_\theta \leftarrow -\nabla_\theta \frac{1}{m} \sum_{i=1}^m f_w(g_\theta(z^{(i)}))$ 
11:   $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$ 
12: end while
```

Figure 7.1: Training loop information of Wasserstein GAN from the original paper

The key highlights about implementing WGAN are as follows:

- Instead of *sigmoid*, we will be using a linear activation function in the output layer of critic model, to generate the critic score.
- Labels for the real images will be -1, and +1 for the fake or generated images.
- After each minibatch update, we need to clip all the critic weights to the following range [-0.01, 0.01].
- We will use *RMSProp* optimizer with a very small learning rate (0.00005) and without momentum for training WGAN.
- At each training iteration, we will update the critic network more than the generator network.

Note that, the clipping of all critic weights to a small range of [-0.01, 0.01] after every gradient update is very important here. If we don't apply any restrictions on the range of critic weights, it may take a very long time for these weights to saturate and thus, training the critic till optimality will become harder. So, we clip all the critic weights to a compact space, so that they can reach their limit faster and the critic model learns faster. We can't restrict the weights to an extremely small range as well, because otherwise it can easily lead to vanishing gradient problems in larger networks.

Now that we have a good understanding of the basic concept and key implementation details of Wasserstein GAN. Let's jump into the coding part and implement a WGAN on MNIST Handwritten Digits dataset.

The code samples shown in this skill can be downloaded from the following Github location:

<https://github.com/kartikgill/The-GAN-Book/tree/main/Skill-07>

Let's get started.

W-GAN ON MNIST DIGITS DATASET

Objective

In this experiment, we will implement and train a Wasserstein GAN on MNIST Handwritten Digits dataset.

This experiment has the following steps:

- Importing Libraries
- Download and Show Data
- Data Normalization
- Define Generator Network
- Define Critic Network
- Define Wasserstein Loss
- Define combined model: W-GAN
- Define Utility Functions
- Training WGAN
- Results

Let's get started.

Step 1: Importing Libraries

Open a Jupyter Notebook with python kernel and import useful python packages in a cell. In this experiment, we will use ‘*numpy*’ for data manipulation, ‘*matplotlib*’ for plotting images and graphs, and *TensorFlow2* for implementing the model architecture.

Checkout the following snippet for importing libraries:

```
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
from tqdm import tqdm_notebook  
%matplotlib inline  
import tensorflow  
print(tensorflow.__version__)
```

Out [1]: 2.4.1

Let's now download the dataset.

Step 2: Download and Show Data

In this step, we will download the MNIST handwritten digits dataset and display a few samples. The following python code loads the MNIST data and plots a few samples:

```
from tensorflow.keras.datasets import fashion_mnist, mnist  
(trainX, trainY), (testX, testY) = mnist.load_data()  
print('Training data shapes: X=%s, y=%s' % (trainX.shape, trainY.shape))  
  
print('Testing data shapes: X=%s, y=%s' % (testX.shape, testY.shape))
```

```
for k in range(9):
```

```
    plt.figure(figsize=(9, 6))
```

```
    for j in range(9):
```

```
        i = np.random.randint(0, 10000)
```

```
        plt.subplot(990 + 1 + j)
```

```
        plt.imshow(trainX[i], cmap='gray_r')
```

```
        #plt.title(trainY[i])
```

```
        plt.axis('off')
```

```
    plt.show()
```

Figure 7.2 shows some samples from the dataset. MNIST Handwritten Digits dataset has 60k training images and 10k test images of size 28 x 28, and each image is gray scale and comes with a label value indicating the digit class.

Out [2]:



Figure 7.2: Few samples from MNIST Handwritten Digits dataset

Let's prepare the dataset for model.

Step 3: Data Normalization

In this step, we will prepare the dataset for the model. We will first normalize pixel values of each image and restrict them to range [-1, 1] by subtracting and dividing with 127.5. We will also add a channel dimension to each image, as expected by the convolutional neural network layers.

The following python snippet prepares the dataset for model:

```
trainX = [(image-127.5)/127.5 for image in trainX]
testX = [(image-127.5)/127.5 for image in testX]
trainX = np.reshape(trainX, (60000, 28, 28, 1))
testX = np.reshape(testX, (10000, 28, 28, 1))
print (trainX.shape, testX.shape, trainY.shape, testY.shape)
```

Following is the shape of our final dataset:

Out [3]: (60000, 28, 28, 1) (10000, 28, 28, 1) (60000,) (10000,)

Our dataset is now ready. Let's define the model architecture now.

Step 4: Define Generator Network

In this step, we will define the generator network. In our setup, the generator accepts a random noise vector of size 100 as input. This input is reshaped into a three-dimensional vector and then passed through the multiple layers of *Conv2DTranspose*, *ReLU* activation, and Batch Normalization with a momentum value of 0.8. The final layer has a ‘*tanh*’ activation function to restrict the pixel values of the generated image within range [-1, 1], similar to our training dataset.

The following python code implements the generator network:

```
random_input = tensorflow.keras.layers.Input(shape = 100)
x = tensorflow.keras.layers.Dense(7*7*128)(random_input)
x = tensorflow.keras.layers.Reshape((7, 7, 128))(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=128, kernel_size=(3,3), strides=2, padding='same')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=128, kernel_size=(3,3), strides=2, padding='same')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=128, kernel_size=(3,3), padding='same')(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.Conv2DTranspose(filters=1, kernel_size=(4,4), padding='same')(x)
generated_image = tensorflow.keras.layers.Activation('tanh')(x)
generator_network = tensorflow.keras.models.Model(inputs=random_input, outputs=generated_image)
generator_network.summary()
```

Following is the summary of the generator network. It roughly has 1M trainable parameters:

Out [4]: Model: "model"

Layer (type)	Output Shape	Param #
=====	=====	=====
input_1 (InputLayer)	[(None, 100)]	0
=====	=====	=====
dense (Dense)	(None, 6272)	633472
=====	=====	=====
reshape (Reshape)	(None, 7, 7, 128)	0
=====	=====	=====
conv2d_transpose (Conv2DTran)	(None, 14, 14, 128)	147584
=====	=====	=====
batch_normalization (BatchNo)	(None, 14, 14, 128)	512
=====	=====	=====
activation (Activation)	(None, 14, 14, 128)	0
=====	=====	=====
conv2d_transpose_1 (Conv2DTr)	(None, 28, 28, 128)	147584
=====	=====	=====
batch_normalization_1 (Batch)	(None, 28, 28, 128)	512
=====	=====	=====
activation_1 (Activation)	(None, 28, 28, 128)	0
=====	=====	=====
conv2d_transpose_2 (Conv2DTr)	(None, 28, 28, 128)	147584
=====	=====	=====
activation_2 (Activation)	(None, 28, 28, 128)	0
=====	=====	=====
conv2d_transpose_3 (Conv2DTr)	(None, 28, 28, 1)	2049
=====	=====	=====
activation_3 (Activation)	(None, 28, 28, 1)	0
=====	=====	=====

Total params: 1,079,297

Trainable params: 1,078,785

Non-trainable params: 512

Let's now define the critic network.

Step 5: Define Critic Network

In this step, we will define the critic network for our experiment. In our setup, the critic network accepts an image of size 28 x 28 x 1 as input. The input image is then passed through multiple layers of *Conv2D*, *LeakyReLU* activation, Batch normalization with a momentum value of 0.8, and dropout layers. The critic network has a linear output layer with just a single neuron without any activations. This final linear layer generates the critic score value for given input samples.

The following python code implements the critic network:

```
image_input = tensorflow.keras.layers.Input(shape=(28, 28, 1))
x = tensorflow.keras.layers.Conv2D(filters=64, kernel_size=
(3,3), strides=2, padding='same')(image_input)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.Dropout(0.25)(x)
x = tensorflow.keras.layers.Conv2D(filters=128, kernel_size=
(3,3), strides=2, padding='same')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.Dropout(0.25)(x)
x = tensorflow.keras.layers.Conv2D(filters=128, kernel_size=
(3,3), strides=2, padding='same')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.Dropout(0.25)(x)
```

```

x = tensorflow.keras.layers.Conv2D(filters=256, kernel_size=
(3,3), padding='same')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.Dropout(0.25)(x)
x = tensorflow.keras.layers.Flatten()(x)
# No activation in final layer
c_out = tensorflow.keras.layers.Dense(1)(x)
critic_network = tensorflow.keras.models.Model(inputs=image_input, out
puts=c_out)
critic_network.summary()

```

Following is the summary of our critic network. It has roughly 520k trainable parameters:

Out [5]: Model: "model_1"

Layer (type)	Output Shape	Param #
--------------	--------------	---------

input_2 (InputLayer)	[(None, 28, 28, 1)]	0
----------------------	---------------------	---

conv2d (Conv2D)	(None, 14, 14, 64)	640
-----------------	--------------------	-----

leaky_re_lu (LeakyReLU)	(None, 14, 14, 64)	0
-------------------------	--------------------	---

dropout (Dropout)	(None, 14, 14, 64)	0
-------------------	--------------------	---

conv2d_1 (Conv2D)	(None, 7, 7, 128)	73856
-------------------	-------------------	-------

batch_normalization_2 (Batch	(None, 7, 7, 128)	512
------------------------------	-------------------	-----

leaky_re_lu_1 (LeakyReLU) (None, 7, 7, 128) 0

dropout_1 (Dropout) (None, 7, 7, 128) 0

conv2d_2 (Conv2D) (None, 4, 4, 128) 147584

batch_normalization_3 (Batch (None, 4, 4, 128) 512

leaky_re_lu_2 (LeakyReLU) (None, 4, 4, 128) 0

dropout_2 (Dropout) (None, 4, 4, 128) 0

conv2d_3 (Conv2D) (None, 4, 4, 256) 295168

batch_normalization_4 (Batch (None, 4, 4, 256) 1024

leaky_re_lu_3 (LeakyReLU) (None, 4, 4, 256) 0

dropout_3 (Dropout) (None, 4, 4, 256) 0

flatten (Flatten) (None, 4096) 0

dense_1 (Dense) (None, 1) 4097

=====

Total params: 523,393

Trainable params: 522,369

Non-trainable params: 1,024

None

Now that our critic network is ready. Let's define the Wasserstein Loss function.

Step 6: Define Wasserstein Loss

In this step, we will define the Wasserstein loss function as a custom loss function and then compile the critic network.

Following python code defines the custom Wasserstein loss function:

```
# custom loss function  
def wasserstein_loss(y_true, y_pred):  
    return tensorflow.keras.backend.mean(y_true * y_pred)
```

we can now compile the critic network with Wasserstein loss. We will use *RMSprop* optimizer with a learning rate of 0.00005 (as suggested in the paper) for updating the weights of our critic network.

The following python code compiles the critic network:

```
RMSprop_optimizer = tensorflow.keras.optimizers.RMSprop(lr=0.00005)
```

```
critic_network.compile(loss=wasserstein_loss, optimizer=RMSprop_optimizer, metrics=['accuracy'])
```

Let's define the combined WGAN model now.

Step 7: Define combined model: WGAN

We can define the combined WGAN model by simply passing the output of the generator network as input to the critic network. Note that the critic weights are kept frozen for this combined WGAN setup.

The following python snippet defines the combined WGAN model:

```
critic_network.trainable=False  
g_output = generator_network(random_input)  
c_output = critic_network(g_output)  
wgan_model = tensorflow.keras.models.Model(inputs = random_input, outputs = c_output)  
wgan_model.summary()
```

Following is the summary of the combined WGAN model:

Out [7]: Model: "model_2"

Layer (type) Output Shape Param #

=====

input_1 (InputLayer) [(None, 100)] 0

model (Functional) (None, 28, 28, 1) 1079297

model_1 (Functional) (None, 1) 523393

=====

Total params: 1,602,690

Trainable params: 1,078,785

Non-trainable params: 523,905

We can now compile the combined WGAN model with similar parameters that we used for the critic network. The following python code compiles the WGAN:

```
wgan_model.compile(loss=wasserstein_loss, optimizer=RMSprop_optimizer)
```

Our model architecture is now ready. Let's work on the training part now.

Step 8: Define Utility Functions

In this step, we will define some utility functions that will help us in creating data batches for the model training. Specifically, we will define the utility functions for: generating random noise input batches, generating batches of fake samples, and generating batches of real samples. In addition to these, we will also define a utility function for plotting the generator model results.

The following python code implements the utility functions for training:
indices = [i for i in range(0, len(trainX))]

```

def get_random_noise(batch_size, noise_size):
    random_values = np.random.randn(batch_size*noise_size)
    random_noise_batches = np.reshape(random_values, (batch_size, noise_size))
return random_noise_batches

def get_fake_samples(generator_network, batch_size, noise_size):
    random_noise_batches = get_random_noise(batch_size, noise_size)
    fake_samples = generator_network.predict_on_batch(random_noise_batches)
return fake_samples

def get_real_samples(batch_size):
    random_indices = np.random.choice(indices, size=batch_size)
    real_images = trainX[np.array(random_indices),:]
return real_images

def show_generator_results(generator_network):
    for k in range(7):
        plt.figure(figsize=(9, 6))
        random_noise_batches = get_random_noise(7, noise_size)
        fake_samples = generator_network.predict_on_batch(random_noise_batches)
        for j in range(7):
            i = j
            plt.subplot(770 + 1 + j)
            plt.imshow(((fake_samples[i,:,:,-1])/2.0)+0.5, cmap='gray_r')
            plt.axis('off')
        plt.show()
return

```

We are all set to start training the WGAN model now.

Step 9: Training WGAN

In this step, we will write the training iteration loop for our WGAN model. We plan to train our model for 500 epochs, with a batch size of 64, and noise vector size of 100. In each iteration, we will first update the weights of the critic network with a mix batch of real and fake samples. Note that, we will be training critic more often and thus, we will update the critic weights 5 times for each update in the weights of the generator network. After every gradient update, we will clip the critic weights into range [-1, 1] as discussed earlier in this skill. Then, we will freeze the critic weights and update the weights of the generator network using the noise inputs. Just like traditional GANs, we will pass the inverted labels (-1 instead of +1) of the fake samples while updating the generator weights, to make the critic believe that these are real samples and calculate the loss value.

The following python code implements the training loop for our WGAN model:

```
epochs = 500
batch_size = 64
steps = 500
noise_size = 100
for i in range(0, epochs):
    if (i%1 == 0):
        op = show_generator_results(generator_network)
        #print (op)
    for j in range(steps):
        # With Number of Critics=5
        for _ in range(5):
            fake_samples = get_fake_samples(generator_network, batch_size//2, noise_size)
            real_samples = get_real_samples(batch_size=batch_size//2)
            fake_y = np.ones((batch_size//2, 1))
            real_y = -1 * np.ones((batch_size//2, 1))
            # Updating Critic weights
```

```

critic_network.trainable=True
loss_c_real = critic_network.train_on_batch(real_samples, real_y)
loss_c_fake = critic_network.train_on_batch(fake_samples, fake_y)
loss_c = np.add(loss_c_real, loss_c_fake)/2.0
# Clip critic weights
for l in critic_network.layers:
    weights = l.get_weights()
    weights = [np.clip(w, -0.01, 0.01) for w in weights]
    l.set_weights(weights)
if False:
    print ("C_real_loss: %.3f, C_fake_loss: %.3f, C_loss: %.3f"% (loss_c_real[0], loss_c_fake[0], loss_c[0]))
noise_batches = get_random_noise(batch_size, noise_size)
wgan_input = noise_batches
# Make the Discriminator believe that these are real samples and calculate
loss to train the generator
wgan_output = -1 * np.ones((batch_size, 1))
# Updating Generator weights
critic_network.trainable=False
loss_g = wgan_model.train_on_batch(wgan_input, wgan_output)
if j%50 == 0:
    print ("Epoch:%.0f, Step:%.0f, C-Loss:%.6f, G-Loss:%.6f"% (i,j,loss_c[0] ,loss_g))

```

Following are the training logs:

Out [9]: Epoch:0, Step:0, C-Loss:0.000156, G-Loss:-0.000021

Epoch:0, Step:50, C-Loss:-0.005784, G-Loss:-0.002175

Epoch:0, Step:100, C-Loss:0.000427, G-Loss:-0.001326

Epoch:0, Step:150, C-Loss:0.000180, G-Loss:-0.000154

Epoch:0, Step:200, C-Loss:-0.000305, G-Loss:-0.000005

Epoch:0, Step:250, C-Loss:0.000056, G-Loss:-0.000116
Epoch:0, Step:300, C-Loss:0.000084, G-Loss:-0.000012
Epoch:0, Step:350, C-Loss:0.000041, G-Loss:-0.000040
Epoch:0, Step:400, C-Loss:0.000042, G-Loss:-0.000031
Epoch:0, Step:450, C-Loss:0.000030, G-Loss:0.000044
.....
.....
.....
.....
.....
.....
.....
.....
Epoch:40, Step:0, C-Loss:0.000033, G-Loss:0.001238
Epoch:40, Step:50, C-Loss:0.000022, G-Loss:0.001640
Epoch:40, Step:100, C-Loss:-0.000004, G-Loss:0.001384
Epoch:40, Step:150, C-Loss:0.000045, G-Loss:0.001990
Epoch:40, Step:200, C-Loss:-0.000280, G-Loss:0.002620
Epoch:40, Step:250, C-Loss:-0.000134, G-Loss:0.002226
Epoch:40, Step:300, C-Loss:-0.000143, G-Loss:0.002225
Epoch:40, Step:350, C-Loss:0.000007, G-Loss:0.001512
Epoch:40, Step:400, C-Loss:0.000020, G-Loss:0.001927
Epoch:40, Step:450, C-Loss:0.000034, G-Loss:0.001545

We have now successfully trained our WGAN model, let's check out the results.

Step 10: Results

Figure 7.3 shows the results of our WGAN generator network. We can see that the network learns very fast and starts generating results just after 5 epochs of training. The results however, are not very good quality. But the key point here is that using Wasserstein loss function, we have made the training process very smooth and stable.

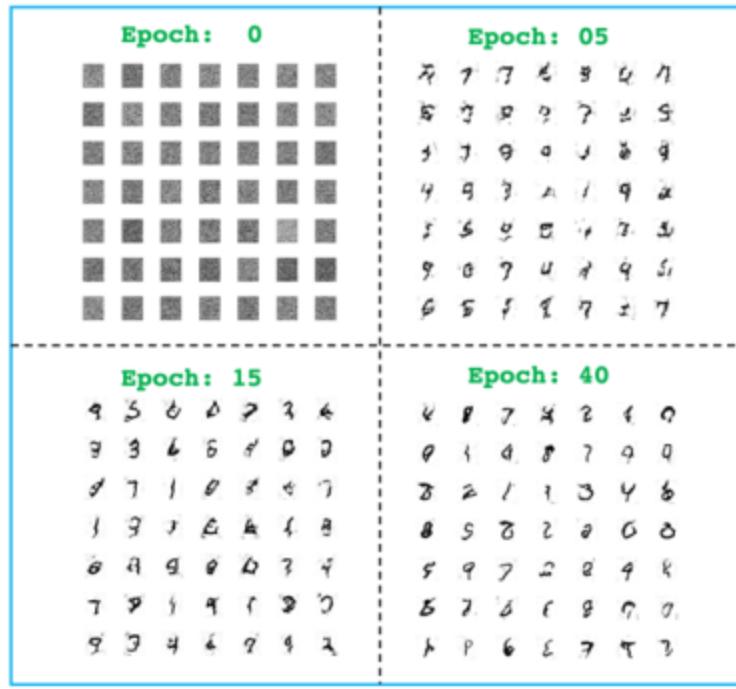


Figure 7.3: Handwritten Digit images generated using WGAN

Next, we will learn about LSGAN.

1. What is Least Squares GAN (LSGAN)?

Generative Adversarial Networks, or GANs for short, have achieved great success when it comes to generating content such as images, music and so on. One issue with traditional GANs is that they use sigmoid cross entropy as a loss function in the discriminator network, and it may lead to vanishing gradient problem during training.

Least Squares Generative Adversarial Networks, or LSGANs for short, were developed to overcome this problem. LSGAN adopts the least squares loss function for the discriminator, and minimizing it yields minimizing the Pearson divergence.

LSGAN has the following two main advantages over the traditional GAN:

- LSGAN is stable in training process as this new loss function solves the potential vanishing gradient issue.
- Quality of generated images is significantly improved over the regular GANs.

The Least Squares Generative Adversarial Network, or LSGAN for short, is a variant of Generative Adversarial Networks that was proposed by *Xudong Mao et. al.* in their 2016 paper titled '*Least Squares Generative Adversarial Networks*'. In their experiments, they claim that LSGAN is quite stable during training and it generates higher quality images as compared to the traditional GAN. We will find it out soon during the experimentation phase. Let's now look at the implementation details of LSGAN.

1. Implementation Details of LSGAN

LSGAN is quite straight forward to implement and it optimizes the model with least squares or mean squared error (MSE) loss function (L2 loss). The output layer of the discriminator network must be linear activation function. Real images are passed with target value of 1 and fake images are passed with 0, as the target value. Authors also recommend using VGG like architectures in both the generator and the discriminator networks as shown in the Figure 7.4.

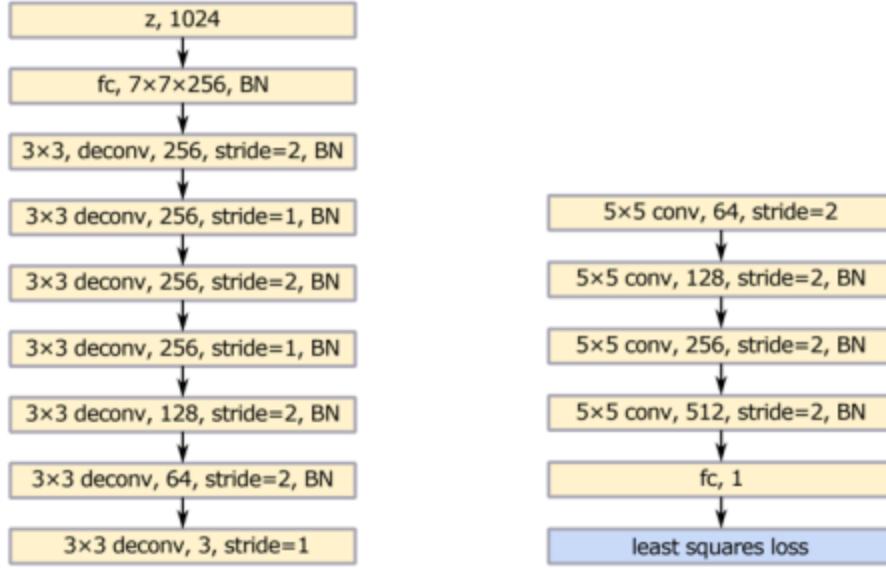


Figure 7.4: LSGAN generator (left) and discriminator (right) architectures as per the paper

Let's now look at some of the impressive results of LSGAN as shown in the paper.

3.2 Results of LSGAN

The experiments from the paper claim that LSGAN has the ability to generate higher quality images than regular GANs. Secondly, LSGAN also performs well when number of classes is high in a dataset. For example: LSGAN performs quite well on the handwritten Chinese characters dataset with 3740 classes.

Figure 7.5 shows the comparison of results between deep convolutional GAN (DCGAN) and LSGAN on LSUN-bedroom dataset. We can clearly see that the images generated by LSGAN look better quality and more realistic as compared to the bedroom images generated by the DCGAN and Energy-based GAN (EBGAN).



(a) Generated by LSGANs.



(b) Generated by DCGANs (Reported in [11]).



(c) Generated by EBGANs (Reported in [26]).

Figure 7.5: LSGAN results (top), comparison with DCGAN (mid) and EBGAN (bottom) on LSUN

Now that we have a good understanding of the Least Squares Generative Adversarial Network or LSGAN, we should now be confident about trying it out ourselves. Let's now develop a LSGAN from scratch and train it on MNIST handwritten digits dataset.

The code samples shown in this skill can be downloaded from the following Github location:

<https://github.com/kartikgill/The-GAN-Book/tree/main/Skill-07>

Let's get started.

LSGAN FOR MNIST HANDWRITTEN DIGITS

Objective

In this experiment, we will implement and train LSGAN on MNIST Handwritten Digits dataset.

This experiment has the following steps:

- Importing Libraries
- Download and Show Data
- Data Normalization
- Define Generator Network
- Define Discriminator Network
- Define combined model: LS-GAN
- Define Utility Functions
- Training LS-GAN
- Results

Let's get started.

Step 1: Importing Libraries

Open a Jupyter Notebook with python kernel and import useful python packages in a cell. In this experiment, we will use ‘*numpy*’ for data manipulation, ‘*matplotlib*’ for plotting images and graphs, and *TensorFlow2* for implementing the model architecture.

Checkout the following snippet for importing libraries:

```
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
from tqdm import tqdm_notebook  
%matplotlib inline  
import tensorflow  
print(tensorflow.__version__)
```

Out [1]: 2.4.1

Let's now download the dataset.

Step 2: Download and Show Data

In this step, we will download and show the MNIST Handwritten Digits dataset. The following python code loads the data and plots a few samples for reference:

```
from tensorflow.keras.datasets import fashion_mnist, mnist  
(trainX, trainY), (testX, testY) = mnist.load_data()  
print('Training data shapes: X=%s, y=%s' % (trainX.shape, trainY.shape))  
  
print('Testing data shapes: X=%s, y=%s' % (testX.shape, testY.shape))
```

```
for k in range(9):
```

```
    plt.figure(figsize=(10, 7))  
    for j in range(9):  
        i = np.random.randint(0, 10000)  
        plt.subplot(990 + 1 + j)  
        plt.imshow(trainX[i], cmap='gray_r')  
        #plt.title(trainY[i])  
        plt.axis('off')  
    plt.show()
```

Figure 7.6 shows a few samples from the dataset. MNIST Handwritten Digits dataset has 60k training images and 10k test images of size 28 x 28, and each image is gray scale and comes with a label value indicating the digit class.

Out [2]:

```

Training data shapes: X=(60000, 28, 28), y=(60000,)
Testing data shapes: X=(10000, 28, 28), y=(10000,)

 3 9 3 7 7 1 7 1 1
 8 3 8 1 2 8 4 2 4
 1 9 3 2 9 6 0 9 0
 1 3 9 0 1 0 0 9 5
 6 6 8 5 3 3 4 7 4
 8 9 7 1 5 7 6 4 7
 3 7 9 5 6 3 5 7 9
 2 4 4 3 7 9 0 8 3
 0 5 2 3 7 0 6 8 8

```

Figure 7.6: A few samples from MNIST Handwritten Digits dataset

Let's prepare data for model.

Step 3: Data Normalization

In this step, we will prepare the dataset for the model. We will first normalize pixel values of each image and restrict them to range [-1, 1] by subtracting and dividing with 127.5. We will also add a channel dimension to each image, as expected by the convolutional neural network layers.

The following python snippet prepares the dataset for model:

```

trainX = [(image-127.5)/127.5 for image in trainX]
testX = [(image-127.5)/127.5 for image in testX]
trainX = np.reshape(trainX, (60000, 28, 28, 1))
testX = np.reshape(testX, (10000, 28, 28, 1))
print (trainX.shape, testX.shape, trainY.shape, testY.shape)

```

Out [3]: (60000, 28, 28, 1) (10000, 28, 28, 1) (60000,) (10000,)

Our dataset is now ready. Let's define the model architecture now.

Step 4: Define Generator Network

Although LSGAN paper suggests using a VGG like architecture for both generator and the discriminator networks. We will keep things simple here work with a small and simple architecture. In our setup, the generator network accepts a noise vector of size 100 as input and then passes it through multiple layers of *Dense*, *LeakyReLU* activation and Batch Normalization with a momentum value of 0.8. The final layer has a '*tanh*' activation function and the output is reshaped into a three-dimensional vector to represent the generated image.

The following python code implements the generator network for our LSGAN model:

```
random_input = tensorflow.keras.layers.Input(shape = 100)
x = tensorflow.keras.layers.Dense(256)(random_input)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Dense(512)(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Dense(1024)(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Dense(28*28*1, activation='tanh')(x)
generated_image = tensorflow.keras.layers.Reshape((28, 28, 1))(x)
generator_network = tensorflow.keras.models.Model(inputs=random_input, outputs=generated_image)
generator_network.summary()
```

Following is the summary of the generator network. It has roughly 1.5M trainable parameters.

Out [4]: Model: "model"

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 100)]	0
dense (Dense)	(None, 256)	25856
leaky_re_lu (LeakyReLU)	(None, 256)	0
batch_normalization (BatchNo)	(None, 256)	1024
dense_1 (Dense)	(None, 512)	131584
leaky_re_lu_1 (LeakyReLU)	(None, 512)	0
batch_normalization_1 (Batch)	(None, 512)	2048
dense_2 (Dense)	(None, 1024)	525312
leaky_re_lu_2 (LeakyReLU)	(None, 1024)	0
batch_normalization_2 (Batch)	(None, 1024)	4096
dense_3 (Dense)	(None, 784)	803600
reshape (Reshape)	(None, 28, 28, 1)	0

Total params: 1,493,520
Trainable params: 1,489,936
Non-trainable params: 3,584

Let's define the discriminator network now.

Step 5: Define Discriminator Network

The discriminator network is also kept very simple in our experiment. In our setup, the network accepts an image of size 28 x 28 x 1, as input and then passes it through multiple layers of *Dense*, and *LeakyReLU* activation with an alpha value of 0.2. Final layer is also linear with just a single neuron.

The following python code implements the discriminator network for our LSGAN setup:

```
image_input = tensorflow.keras.layers.Input(shape=(28, 28, 1))
x = tensorflow.keras.layers.Flatten()(image_input)
x = tensorflow.keras.layers.Dense(512)(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.Dense(256)(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
#Linear activation
d_out = tensorflow.keras.layers.Dense(1)(x)
discriminator_network = tensorflow.keras.models.Model(inputs=image_i
nput, outputs=d_out)
discriminator_network.summary()
```

Following is the summary of the discriminator network. It has about 530k trainable parameters.

Out [5]: Model: "model_1"

Layer (type)	Output Shape	Param #
--------------	--------------	---------

input_2 (InputLayer)	[(None, 28, 28, 1)]	0
----------------------	---------------------	---

flatten (Flatten)	(None, 784)	0
-------------------	-------------	---

```
dense_4 (Dense) (None, 512) 401920
```

```
leaky_re_lu_3 (LeakyReLU) (None, 512) 0
```

```
dense_5 (Dense) (None, 256) 131328
```

```
leaky_re_lu_4 (LeakyReLU) (None, 256) 0
```

```
dense_6 (Dense) (None, 1) 257
```

```
=====
```

```
Total params: 533,505
```

```
Trainable params: 533,505
```

```
Non-trainable params: 0
```

```
None
```

We can now compile the discriminator network with ‘*mean square error (mse)*’ loss function as described in the paper. We will use Adam optimizer with a learning rate of 0.0002 and beta_1 value of 0.5 for updating the weights of our model.

The following python code compiles the discriminator network:

```
adam_optimizer = tensorflow.keras.optimizers.Adam(lr=0.0002, beta_1=0.5)
```

```
discriminator_network.compile(loss='mse', optimizer=adam_optimizer, metrics=['accuracy'])
```

Let’s now define the combined model.

Step 6: Define combined model: LS-GAN

In this step, we will define the combined LSGAN model. Very similar to the traditional GANs, we combine both networks by passing the output of the generator network as input to the discriminator network. Again, the weights

of the discriminator network are kept frozen for this combined setup as this setup will only be used for updating the weights of the generator model.

The following python snippet defines the combined LSGAN:

```
discriminator_network.trainable=False  
g_output = generator_network(random_input)  
d_output = discriminator_network(g_output)  
ls_gan_model = tensorflow.keras.models.Model(inputs = random_input, o  
utputs = d_output)  
ls_gan_model.summary()
```

Following is the summary of the combined LSGAN:

Out [6]: Model: "model_2"

Layer (type) Output Shape Param #

input_1 (InputLayer) [(None, 100)] 0

model (Functional) (None, 28, 28, 1) 1493520

model_1 (Functional) (None, 1) 533505

Total params: 2,027,025

Trainable params: 1,489,936

Non-trainable params: 537,089

We can now compile the LSGAN with similar parameters that we used for compiling the discriminator network earlier.

```
ls_gan_model.compile(loss='mse', optimizer=adam_optimizer)
```

LSGAN is now all set. Let's work on setting up the training now.

Step 7: Define Utility Functions

In this step, we will define some utility functions that will help us in creating data batches for the model training. Specifically, we will define the utility functions for: generating random noise input batches, generating batches of fake samples, and generating batches of real samples. In addition to these, we will also define a utility function for plotting the generator model results.

The following python code implements the utility functions for training:

```
indices = [i for i in range(0, len(trainX))]

def get_random_noise(batch_size, noise_size):
    random_values = np.random.randn(batch_size*noise_size)
    random_noise_batches = np.reshape(random_values, (batch_size, noise_size))
    return random_noise_batches

def get_fake_samples(generator_network, batch_size, noise_size):
    random_noise_batches = get_random_noise(batch_size, noise_size)
    fake_samples = generator_network.predict_on_batch(random_noise_batches)
    return fake_samples

def get_real_samples(batch_size):
    random_indices = np.random.choice(indices, size=batch_size)
    real_images = trainX[np.array(random_indices),:]
    return real_images

def show_generator_results(generator_network):
    for k in range(9):
        plt.figure(figsize=(10, 7))
        random_noise_batches = get_random_noise(10, noise_size)
        fake_samples = generator_network.predict_on_batch(random_noise_batches)
        for j in range(9):
            i = j
```

```

plt.subplot(990 + 1 + j)
plt.imshow(((fake_samples[i,:,:,-1])/2.0)+0.5, cmap='gray_r')
plt.axis('off')
plt.show()
return

```

We are now all set to start training our LSGAN.

Step 8: Training LSGAN

In this step, we will write the training iteration loop for our LSGAN model. We plan to train our model for 500 epochs, with a batch size of 100, and noise vector size of 100. In each iteration, we will first update the weights of the discriminator network with a mix batch of real and fake samples. Then, we will freeze the discriminator network weights and update the weights of the generator network using the noise inputs. Just like traditional GANs, we will pass the inverted labels (1 instead of 0) of the fake samples while updating the generator weights, to make the discriminator believe that these are real samples and calculate the loss value.

The following python code implements the training loop for our LSGAN model:

```

epochs = 500
batch_size = 100
steps = 500
noise_size = 100
for i in range(0, epochs):
    if (i%1 == 0):
        op = show_generator_results(generator_network)
        #print (op)
    for j in range(steps):
        fake_samples = get_fake_samples(generator_network, batch_size//2, noise_size)
        real_samples = get_real_samples(batch_size=batch_size//2)

```

```

fake_y = np.zeros((batch_size//2, 1))
real_y = np.ones((batch_size//2, 1))
# Updating Discriminator weights
discriminator_network.trainable=True
loss_d_real = discriminator_network.train_on_batch(real_samples, real_y)
loss_d_fake = discriminator_network.train_on_batch(fake_samples, fake_y)
loss_d = np.add(loss_d_real, loss_d_fake)/2.0
noise_batches = get_random_noise(batch_size, noise_size)
ls_gan_input = noise_batches
# Make the Discriminator believe that these are real samples and calculate
loss to train the generator
ls_gan_output = np.ones((batch_size, 1))
# Updating Generator weights
discriminator_network.trainable=False
loss_g = ls_gan_model.train_on_batch(ls_gan_input, ls_gan_output)
if j%50 == 0:
    print ("Epoch:%.0f, Step:%.0f, D-Loss:%.3f, D-Acc:%.3f, G-
Loss:%.3f"\n
          (i,j,loss_d[0],loss_d[1]*100,loss_g))

```

Following are the training logs:

Out [8]: Epoch:0, Step:0, D-Loss:0.662, D-Acc:52.000, G-Loss:1.984

Epoch:0, Step:50, D-Loss:0.078, D-Acc:96.000, G-Loss:0.913

Epoch:0, Step:100, D-Loss:0.095, D-Acc:92.000, G-Loss:0.935

Epoch:0, Step:150, D-Loss:0.028, D-Acc:99.000, G-Loss:0.966

Epoch:0, Step:200, D-Loss:0.055, D-Acc:96.000, G-Loss:0.894

Epoch:0, Step:250, D-Loss:0.111, D-Acc:84.000, G-Loss:0.712

Epoch:0, Step:300, D-Loss:0.101, D-Acc:85.000, G-Loss:0.897

Epoch:0, Step:350, D-Loss:0.191, D-Acc:68.000, G-Loss:0.518

Epoch:0, Step:400, D-Loss:0.091, D-Acc:86.000, G-Loss:0.963

Epoch:0, Step:450, D-Loss:0.069, D-Acc:88.000, G-Loss:0.980

.....

.....

.....

.....

.....

Epoch:50, Step:0, D-Loss:0.257, D-Acc:52.000, G-Loss:0.304

Epoch:50, Step:50, D-Loss:0.252, D-Acc:53.000, G-Loss:0.310

Epoch:50, Step:100, D-Loss:0.237, D-Acc:62.000, G-Loss:0.330

Epoch:50, Step:150, D-Loss:0.235, D-Acc:60.000, G-Loss:0.324

Epoch:50, Step:200, D-Loss:0.252, D-Acc:51.000, G-Loss:0.332

Epoch:50, Step:250, D-Loss:0.250, D-Acc:55.000, G-Loss:0.307

Epoch:50, Step:300, D-Loss:0.246, D-Acc:54.000, G-Loss:0.355

Epoch:50, Step:350, D-Loss:0.228, D-Acc:62.000, G-Loss:0.337

Epoch:50, Step:400, D-Loss:0.238, D-Acc:55.000, G-Loss:0.311

Epoch:50, Step:450, D-Loss:0.245, D-Acc:57.000, G-Loss:0.314

We have now trained our LSGAN model. Let's check out the results.

Step 9: Results

Figure 7.7 shows the results of our LSGAN generator network. We can see that the network learns very fast and starts generating results just after 5 epochs of training. At epoch 20, the results start looking good. At epoch 50, we can see some good quality results from our small LSGAN model.

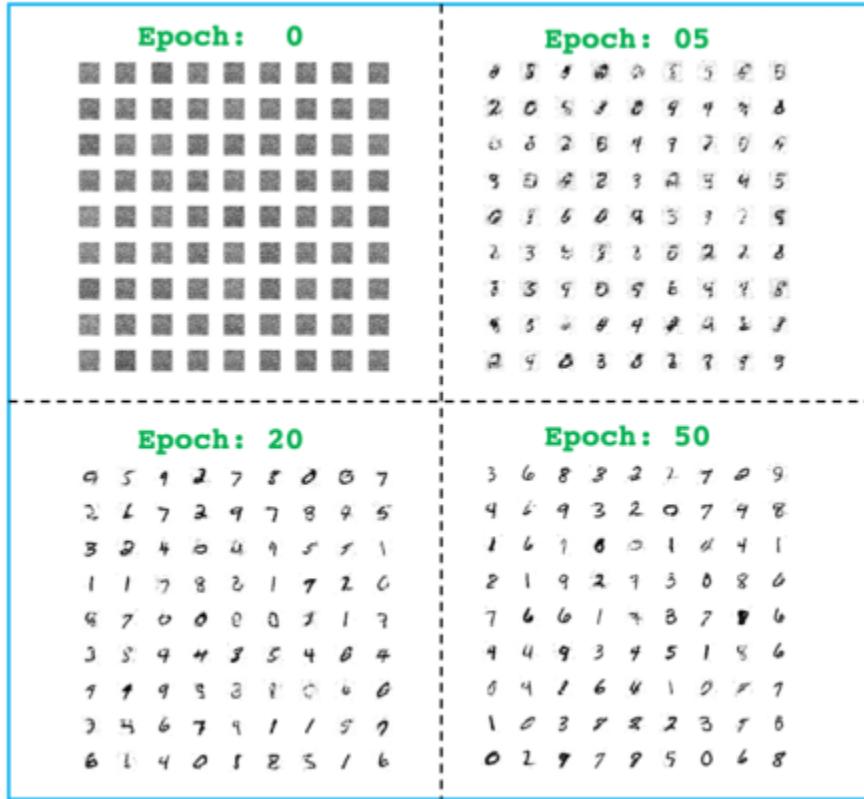


Figure 7.7: Handwritten Digit images generated by a LSGAN model

Next, we will learn about improving the Wasserstein GAN using gradient penalty method. Let's get started.

1. Improving WGAN using gradient penalty (WGAN-GP)

Wasserstein GAN, or WGAN, was proposed by *Martin Arjovsky et. al.* in 2017, to solve the training stability related issues present in regular GANs. But sometimes even WGAN fails to converge or doesn't produce high quality results. To overcome these issues with WGAN, a new extension of WGAN was developed called Wasserstein GAN with Gradient Penalty or WGAN-GP.

WGAN-GP was proposed by *Ishaan Gulrajani et. al.* in their 2017 paper titled '*Improved Training of Wasserstein GANs*'. This paper claims that the undesired behavior or issues with WGAN mainly come from the application of weight clipping to enforce a Lipschitz constraint. This paper further proposes a new method of enforcing the Lipschitz constraint without relying on weight clipping.

Following are the major improvements proposed by this paper for WGANs:

- Replace weight clipping in WGAN by gradient penalty to achieve stable training without issues.
- WGAN-GP works well even when deeper networks are used as generator and critic.
- WGAN-GP performs better than standard WGAN, in terms of quality of results without even needing to tune the hyperparameters.

Let's get into the implementation details now.

4.1 Implementation Details of WGAN-GP

WGAN-GP improves the standard WGAN architecture by removing weight clipping from the critic network and instead penalizes the norm of the gradient of critic with respect to its input, to enforce the Lipschitz constraint. Figure 7.8 shows the implementation details of WGAN-GP, with hyperparameter values that were used in the experiments of the original paper.

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

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

Require: initial critic parameters w_0 , initial generator parameters θ_0 .

```
1: while  $\theta$  has not converged do
2:   for  $t = 1, \dots, n_{\text{critic}}$  do
3:     for  $i = 1, \dots, m$  do
4:       Sample real data  $\mathbf{x} \sim \mathbb{P}_r$ , latent variable  $\mathbf{z} \sim p(\mathbf{z})$ , a random number  $\epsilon \sim U[0, 1]$ .
5:        $\tilde{\mathbf{x}} \leftarrow G_\theta(\mathbf{z})$ 
6:        $\hat{\mathbf{x}} \leftarrow \epsilon \mathbf{x} + (1 - \epsilon) \tilde{\mathbf{x}}$ 
7:        $L^{(i)} \leftarrow D_w(\tilde{\mathbf{x}}) - D_w(\mathbf{x}) + \lambda(\|\nabla_{\hat{\mathbf{x}}} D_w(\hat{\mathbf{x}})\|_2 - 1)^2$ 
8:     end for
9:      $w \leftarrow \text{Adam}(\nabla_w \frac{1}{m} \sum_{i=1}^m L^{(i)}, w, \alpha, \beta_1, \beta_2)$ 
10:   end for
11:   Sample a batch of latent variables  $\{\mathbf{z}^{(i)}\}_{i=1}^m \sim p(\mathbf{z})$ .
12:    $\theta \leftarrow \text{Adam}(\nabla_\theta \frac{1}{m} \sum_{i=1}^m -D_w(G_\theta(\mathbf{z})), \theta, \alpha, \beta_1, \beta_2)$ 
13: end while
```

Figure 7.8: Implementing training loop of Wasserstein GAN with gradient penalty (WGAN-GP)

Now we can go ahead and develop a WGAN-GP for ourselves. In the following experiment, we will develop a WGAN-GP model for Fashion MNIST dataset. Let get started.

The code samples shown in this skill can be downloaded from the following Github location:

<https://github.com/kartikgill/The-GAN-Book/tree/main/Skill-07>

Let's get started.

WGAN-GP FOR FASHION MNIST DATA

Objective

In this experiment, we will implement and train a WGAN-GP model on Fashion MNIST dataset and verify the results.

This experiment has the following steps:

- Importing Libraries
- Download and Show Data
- Data Normalization
- Define Generator Network
- Define Critic Network
- Define Wasserstein and Gradient Penalty Loss
- Define Final Critic and WGAN-GP models
- Define Utility Functions
- Training WGAN-GP
- Results

Let's get started.

Step 1: Importing Libraries

Open a Jupyter Notebook with python kernel and import useful python packages in a cell. In this experiment, we will use '*numpy*' for data manipulation, '*matplotlib*' for plotting images and graphs, and *TensorFlow version 1.14* for implementing the model architecture.

Checkout the following snippet for importing libraries:

```
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
from tqdm import tqdm_notebook  
%matplotlib inline  
import tensorflow as tf  
import keras
```

```
print(tf.__version__)
print(keras.__version__)
Out [1]: Using TensorFlow backend.
```

1.14.0

2.3.0

Let's now download the dataset.

Step 2: Download and Show Data

The following python code downloads the Fashion MNIST dataset and plots few sample images for reference:

```
from tensorflow.keras.datasets import fashion_mnist, mnist
(trainX, trainY), (testX, testY) = fashion_mnist.load_data()
print('Training data shapes: X=%s, y=%s' % (trainX.shape, trainY.shape))
print('Testing data shapes: X=%s, y=%s' % (testX.shape, testY.shape))
```

for k in range(9):

```
    plt.figure(figsize=(10, 7))
    for j in range(9):
        i = np.random.randint(0, 10000)
        plt.subplot(990 + 1 + j)
        plt.imshow(trainX[i], cmap='gray_r')
        #plt.title(trainY[i])
        plt.axis('off')
    plt.show()
```

Out [2]:



Figure 7.9: Few samples from the Fashion MNIST dataset

Figure 7.9 shows some samples from the dataset. Fashion MNIST dataset has 60k training images and 10k test images of size 28 x 28, and each sample is gray scale image and comes with a label value indicating the class value.

Let's prepare data for model.

Step 3: Data Normalization

In this step, we will prepare the dataset for the model. We will first normalize pixel values of each image and restrict them to range [-1, 1] by subtracting and dividing with 127.5. We will also add a channel dimension to each image, as expected by the convolutional neural network layers.

The following python snippet prepares the dataset for model:

```
trainX = [(image-127.5)/127.5 for image in trainX]
testX = [(image-127.5)/127.5 for image in testX]
trainX = np.reshape(trainX, (60000, 28, 28, 1))
testX = np.reshape(testX, (10000, 28, 28, 1))
print (trainX.shape, testX.shape, trainY.shape, testY.shape)
Out [3]: (60000, 28, 28, 1) (10000, 28, 28, 1) (60000,) (10000,)
```

Our dataset is now ready. Let's define the model.

Step 4: Define Generator Network

The following python function defines the generator network. It takes a random noise vector as input and then converts it into a three-dimensional vector using *Reshape* layers. This input is then passed through multiple layers of *Conv2DTranspose*, Batch Normalization with a momentum of 0.8, and *ReLU* activation. The final layer has ‘*tanh*’ activation to restrict the pixel values into range [-1, 1], for the generated images, just like the training images. See the following code:

```
def make_generator(random_input):
    x = keras.layers.Dense(7*7*128)(random_input)
    x = keras.layers.Reshape((7, 7, 128))(x)
    x = keras.layers.Conv2DTranspose(filters=128, kernel_size=(4,4), strides=2, padding='same')(x)
    x = keras.layers.BatchNormalization(momentum=0.8)(x)
    x = keras.layers.Activation('relu')(x)
    x = keras.layers.Conv2DTranspose(filters=64, kernel_size=(4,4), strides=2, padding='same')(x)
    x = keras.layers.BatchNormalization(momentum=0.8)(x)
    x = keras.layers.Activation('relu')(x)
    x = keras.layers.Conv2DTranspose(filters=1, kernel_size=(4,4), padding='same')(x)
    generated_image = keras.layers.Activation('tanh')(x)
    generator_network = keras.models.Model(inputs=random_input, outputs=generated_image)
    print(generator_network.summary())
    return generator_network
```

Let’s now define the critic network.

Step 5: Define Critic Network

The following python function defines the critic network. It accepts an image of size 28 x 28 x 1, as input and passes it through multiple layers of *Conv2D*, Batch Normalization, *LeakyReLU* activation, and Dropouts. The

final layer only has single neuron (linear activation). Final layer is kept linear as the critic will generate a score for the realness or fakeness of the input images. See the following code:

```
def make_critic():
    image_input = keras.layers.Input(shape=(28, 28, 1))
    x = keras.layers.Conv2D(filters=16, kernel_size=
(3,3), strides=2, padding='same')(image_input)
    x = keras.layers.LeakyReLU(alpha=0.2)(x)
    x = keras.layers.Dropout(0.25)(x)
    x = keras.layers.Conv2D(filters=32, kernel_size=
(3,3), strides=2, padding='same')(x)
    x = keras.layers.BatchNormalization(momentum=0.8)(x)
    x = keras.layers.LeakyReLU(alpha=0.2)(x)
    x = keras.layers.Dropout(0.25)(x)
    x = keras.layers.Conv2D(filters=64, kernel_size=
(3,3), strides=2, padding='same')(x)
    x = keras.layers.BatchNormalization(momentum=0.8)(x)
    x = keras.layers.LeakyReLU(alpha=0.2)(x)
    x = keras.layers.Dropout(0.25)(x)
    x = keras.layers.Conv2D(filters=128, kernel_size=(3,3), padding='same')
(x)
    x = keras.layers.BatchNormalization(momentum=0.8)(x)
    x = keras.layers.LeakyReLU(alpha=0.2)(x)
    x = keras.layers.Dropout(0.25)(x)
    x = keras.layers.Flatten()(x)
    c_out = keras.layers.Dense(1)(x)
    critic_network = keras.models.Model(inputs=image_input, outputs=c_out)

    print(critic_network.summary())
    return critic_network
```

Let's now define the custom loss function for our WGAN-GP setup.

Step 6: Define Wasserstein and Gradient Penalty Loss

In this step, we will define the custom Wasserstein loss with gradient penalty. The following python code defines the gradient penalty loss for our WGAN-GP model. This code has been taken from the awesome Github repository at: https://github.com/eriklindernoren/Keras-GAN/blob/master/wgan_gp/wgan_gp.py

```
# This cell code is taken from: https://github.com/eriklindernoren/Keras-GAN/blob/master/wgan_gp/wgan_gp.py
```

```
import keras
from functools import partial
import keras.backend as K
from keras.layers.merge import _Merge
class RandomWeightedAverage(_Merge):
    """Provides a (random) weighted average between real and generated image samples"""
    def _merge_function(self, inputs):
        alpha = K.random_uniform((32, 1, 1, 1))
        return (alpha * inputs[0]) + ((1 - alpha) * inputs[1])
    def gradient_penalty_loss(y_true, y_pred, averaged_samples):
        """Computes gradient penalty based on prediction and weighted real / fake samples
        """
        gradients = K.gradients(y_pred, averaged_samples)[0]
        # compute the euclidean norm by squaring ...
        gradients_sqr = K.square(gradients)
        # ... summing over the rows ...
        gradients_sqr_sum = K.sum(gradients_sqr,

```

Computes gradient penalty based on prediction and weighted real / fake samples

```
....
```

```
gradients = K.gradients(y_pred, averaged_samples)[0]
# compute the euclidean norm by squaring ...
gradients_sqr = K.square(gradients)
# ... summing over the rows ...
gradients_sqr_sum = K.sum(gradients_sqr,
axis=np.arange(1, len(gradients_sqr.shape)))
```

```

# ... and sqrt
gradient_l2_norm = K.sqrt(gradients_sqr_sum)
# compute lambda * (1 - ||grad||)^2 still for each single sample
gradient_penalty = K.square(1 - gradient_l2_norm)
# return the mean as loss over all the batch samples
return K.mean(gradient_penalty)

```

Let's now define the final critic and WGAN-GP model.

Step 7: Define Final Critic and WGAN-GP models

The following python code defines the final critic network and the combined WGAN-GP model:

```

def make_critic_model():
    real_img = keras.Input(shape=(28, 28, 1))
    random_input = keras.layers.Input(shape=(100,))
    generator_network = make_generator(random_input)
    critic_network = make_critic()
    generator_network.trainable=False
    critic_network.trainable=True
    fake_img = generator_network(random_input)
    interpolated_img = RandomWeightedAverage()([real_img, fake_img])
    c_out_interpolated = critic_network(interpolated_img)
    partial_gp_loss = partial(gradient_penalty_loss,
        averaged_samples=interpolated_img)
    partial_gp_loss.__name__ = 'gradient_penalty' # Keras requires function names
    fake = critic_network(fake_img)
    real = critic_network(real_img)
    critic_model = keras.models.Model(inputs=[real_img, random_input], outputs=[real, fake, c_out_interpolated])
    # compiling model

```

```

optimizer = keras.optimizers.RMSprop(lr=0.00005)
critic_model.compile(loss=
[wasserstein_loss, wasserstein_loss, partial_gp_loss], \
optimizer=optimizer, metrics=['accuracy'], loss_weights=[1,1,10])
print (critic_model.summary())
return critic_model, critic_network, generator_network
def make_wgan_gp(critic_network, generator_network):
#Defining Combined Model
critic_network.trainable=False
generator_network.trainable=True
noise_input = keras.layers.Input(shape=(100,))
g_output = generator_network(noise_input)
c_output = critic_network(g_output)
wgan_gp_model = keras.models.Model(inputs = noise_input, outputs = c_
output)
# compiling model
optimizer = keras.optimizers.RMSprop(lr=0.00005)
wgan_gp_model.compile(loss=wasserstein_loss, optimizer=optimizer)
print (wgan_gp_model.summary())
return wgan_gp_model

```

Following is the summary of all three models in the setup:

Out [7]: Model: "model_1"

Layer (type) Output Shape Param #

=====

input_2 (InputLayer) (None, 100) 0

dense_1 (Dense) (None, 6272) 633472

reshape_1 (Reshape) (None, 7, 7, 128) 0

conv2d_transpose_1 (Conv2DTr (None, 14, 14, 128) 262272

batch_normalization_1 (Batch (None, 14, 14, 128) 512

activation_1 (Activation) (None, 14, 14, 128) 0

conv2d_transpose_2 (Conv2DTr (None, 28, 28, 64) 131136

batch_normalization_2 (Batch (None, 28, 28, 64) 256

activation_2 (Activation) (None, 28, 28, 64) 0

conv2d_transpose_3 (Conv2DTr (None, 28, 28, 1) 1025

activation_3 (Activation) (None, 28, 28, 1) 0

=====

Total params: 1,028,673

Trainable params: 1,028,289

Non-trainable params: 384

None

Model: "model_2"

Layer (type) Output Shape Param #

=====

input_3 (InputLayer) (None, 28, 28, 1) 0

conv2d_1 (Conv2D) (None, 14, 14, 16) 160

leaky_re_lu_1 (LeakyReLU) (None, 14, 14, 16) 0

dropout_1 (Dropout) (None, 14, 14, 16) 0

conv2d_2 (Conv2D) (None, 7, 7, 32) 4640

batch_normalization_3 (Batch (None, 7, 7, 32) 128

leaky_re_lu_2 (LeakyReLU) (None, 7, 7, 32) 0

dropout_2 (Dropout) (None, 7, 7, 32) 0

conv2d_3 (Conv2D) (None, 4, 4, 64) 18496

batch_normalization_4 (Batch (None, 4, 4, 64) 256

leaky_re_lu_3 (LeakyReLU) (None, 4, 4, 64) 0

dropout_3 (Dropout) (None, 4, 4, 64) 0

conv2d_4 (Conv2D) (None, 4, 4, 128) 73856

batch_normalization_5 (Batch (None, 4, 4, 128) 512

leaky_re_lu_4 (LeakyReLU) (None, 4, 4, 128) 0

dropout_4 (Dropout) (None, 4, 4, 128) 0

```
flatten_1 (Flatten) (None, 2048) 0
```

```
dense_2 (Dense) (None, 1) 2049
```

```
=====
```

```
Total params: 100,097
```

```
Trainable params: 99,649
```

```
Non-trainable params: 448
```

```
None
```

```
Model: "model_4"
```

```
Layer (type) Output Shape Param #
```

```
=====
```

```
input_4 (InputLayer) (None, 100) 0
```

```
model_1 (Model) (None, 28, 28, 1) 1028673
```

```
model_2 (Model) (None, 1) 100097
```

```
=====
```

```
Total params: 1,128,770
```

```
Trainable params: 1,028,289
```

```
Non-trainable params: 100,481
```

```
None
```

Our WGAN-GP is now all set. Let's work on the training setup now.

Step 8: Define Utility Functions

In this step, we will define some utility functions that will help us in creating data batches for the model training. Specifically, we will define the utility functions for: generating random noise input batches, generating batches of fake samples, and generating batches of real samples. In addition to these, we will also define a utility function for plotting the generator model results.

The following python code implements the utility functions for training:

```
indices = [i for i in range(0, len(trainX))]

def get_random_noise(batch_size, noise_size):
    random_noise_batches = np.random.normal(0, 1, (batch_size, noise_size))

    return random_noise_batches

def get_fake_samples(generator_network, batch_size, noise_size):
    random_noise_batches = get_random_noise(batch_size, noise_size)
    fake_samples = generator_network.predict_on_batch(random_noise_batches)

    return fake_samples

def get_real_samples(batch_size):
    random_indices = np.random.choice(indices, size=batch_size)
    real_images = trainX[np.array(random_indices),:]

    return real_images

def show_generator_results(generator_network):
    for k in range(9):
        plt.figure(figsize=(9, 6))
        random_noise_batches = get_random_noise(10, noise_size)
        fake_samples = generator_network.predict_on_batch(random_noise_batches)

        for j in range(9):
            i = j
            plt.subplot(990 + 1 + j)
```

```

plt.imshow(fake_samples[i,:,:,-1], cmap='gray_r')
plt.axis('off')
plt.show()
return

```

Let's now train the WGAN-GP model.

Step 9: Training WGAN-GP

In this step, we will write the training iteration loop for our WGAN-GP model. We plan to train our model for 500 epochs, with a batch size of 32, and noise vector size of 100. In each iteration, we will first update the weights of the critic network with a mix batch of real and fake samples. Then, we will freeze the critic network weights and update the weights of the generator network using the noise inputs. Just like traditional GANs, we will pass the inverted labels (-1 instead of 1) of the fake samples while updating the generator weights, to make the critic believe that these are real samples and calculate the loss value.

The following python code implements the training loop for our WGAN-GP model:

```

epochs = 500
batch_size = 32
steps = 500
noise_size = 100
for i in range(0, epochs):
    for j in range(steps):
        # With Number of Critics=5
        for _ in range(5):
            random_noise_batch = get_random_noise(batch_size, noise_size)
            real_samples = get_real_samples(batch_size=batch_size)
            fake_y = np.ones((batch_size, 1))
            real_y = -np.ones((batch_size, 1))
            dummy = np.zeros((batch_size, 1))

```

```

# Updating Critic weights
loss_c = critic_model.train_on_batch([real_samples,random_noise_batch],
[real_y, fake_y, dummy])
noise_batch = get_random_noise(batch_size, noise_size)
wgan_input = noise_batch
# Make the Discriminator believe that these are real samples and calculate loss to train the generator
wgan_output = -np.ones((batch_size, 1))
# Updating Generator weights
loss_g = wgan_gp_model.train_on_batch(wgan_input, wgan_output)
if j%10 == 0:
    print ("Epoch:%.0f, Step:%.0f, C-Loss:%.3f G-Loss:%.3f"%(i,j,loss_c[0],loss_g))
    if j%100==0:
        show_generator_results(generator_network)
Following are the training logs for our WGAN-GP setup:
Out [9]: Epoch:0, Step:0, C-Loss:15.124 G-Loss:-0.189

```

Epoch:0, Step:10, C-Loss:1.847 G-Loss:0.010
 Epoch:0, Step:20, C-Loss:-0.361 G-Loss:-0.065
 Epoch:0, Step:30, C-Loss:-1.830 G-Loss:-0.226
 Epoch:0, Step:40, C-Loss:-2.138 G-Loss:-0.436
 Epoch:0, Step:50, C-Loss:-1.353 G-Loss:-0.262
 Epoch:0, Step:60, C-Loss:-2.981 G-Loss:-0.225
 Epoch:0, Step:70, C-Loss:-2.414 G-Loss:0.341
 Epoch:0, Step:80, C-Loss:-2.423 G-Loss:0.593
 Epoch:0, Step:90, C-Loss:-2.528 G-Loss:0.973
 Epoch:0, Step:100, C-Loss:-2.562 G-Loss:1.088
 . . .
 . . .
 . . .

. . . .

. . . .

Epoch:22, Step:110, C-Loss:-0.613 G-Loss:-0.182

Epoch:22, Step:120, C-Loss:-1.030 G-Loss:-0.201

Epoch:22, Step:130, C-Loss:-0.203 G-Loss:0.552

Epoch:22, Step:140, C-Loss:-1.111 G-Loss:0.107

Epoch:22, Step:150, C-Loss:-0.917 G-Loss:0.017

Epoch:22, Step:160, C-Loss:-0.964 G-Loss:0.498

Epoch:22, Step:170, C-Loss:-0.558 G-Loss:-0.111

Epoch:22, Step:180, C-Loss:-0.983 G-Loss:0.530

Epoch:22, Step:190, C-Loss:-0.116 G-Loss:-0.365

Epoch:22, Step:200, C-Loss:0.358 G-Loss:-0.217

Now that our WGAN-GP is trained. Let's check out some results.

Step 10: Results

Figure 7.10 shows the results of our WGAN-GP generator network. We can see that the network learns very fast and starts generating results just after 1 epoch of training. At epoch 5, the results start looking good. At epoch 20, the results start to look very realistic and hard to distinguish from the real ones.

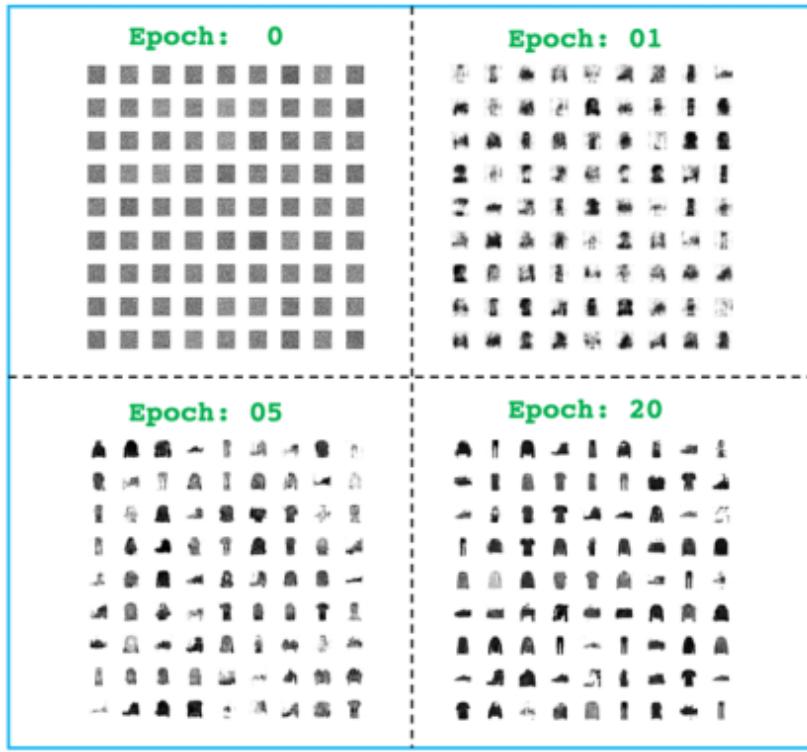


Figure 7.10: Fashion MNIST images generated through WGAN-GP model

We just saw that the gradient penalty greatly improves the results of WGAN setup.

1. Conclusion

In this skill, we discussed the common challenges with regular GAN training due to the nature of its loss function. To overcome these challenges, researchers have proposed some new loss functions that make GAN training stable and also improve the overall quality of results.

In this skill, we learned about three such methods that stabilize the GAN training and also improve overall results using new loss functions. Specifically, we learned about three approaches: Wasserstein GAN or WGAN, Least Squares GAN or LSGAN, and improved WGAN using gradient penalty or WGAN-GP. During the experiments, we saw how these methods make the

GAN training process more stable and also how they improve the overall quality of the results.

Similarly, there are other research works as well, where the researchers have tried to improve GAN performance using some tweaks to the objective function. *Mario Lucic et. al.* study about the different variants of GANs in their 2018 paper titled ‘Are GANs Created Equal? A Large-Scale study’. In this paper, the authors have also created a table to compare the objective functions, an image of that table is shown in Figure 7.11.

GAN	DISCRIMINATOR LOSS	GENERATOR LOSS
MM GAN	$\mathcal{L}_D^{GAN} = -\mathbb{E}_{x \sim p_d} [\log(D(x))] - \mathbb{E}_{\hat{x} \sim p_g} [\log(1 - D(\hat{x}))]$	$\mathcal{L}_G^{GAN} = \mathbb{E}_{\hat{x} \sim p_g} [\log(1 - D(\hat{x}))]$
NS GAN	$\mathcal{L}_D^{NSGAN} = -\mathbb{E}_{x \sim p_d} [\log(D(x))] - \mathbb{E}_{\hat{x} \sim p_g} [\log(1 - D(\hat{x}))]$	$\mathcal{L}_G^{NSGAN} = -\mathbb{E}_{\hat{x} \sim p_g} [\log(D(\hat{x}))]$
WGAN	$\mathcal{L}_D^{WGAN} = -\mathbb{E}_{x \sim p_d} [D(x)] + \mathbb{E}_{\hat{x} \sim p_g} [D(\hat{x})]$	$\mathcal{L}_G^{WGAN} = -\mathbb{E}_{\hat{x} \sim p_g} [D(\hat{x})]$
WGAN GP	$\mathcal{L}_D^{WGANGP} = \mathcal{L}_D^{WGAN} + \lambda \mathbb{E}_{\hat{x} \sim p_g} [(\nabla D(\alpha x + (1 - \alpha \hat{x})) _2 - 1)^2]$	$\mathcal{L}_G^{WGANGP} = -\mathbb{E}_{\hat{x} \sim p_g} [D(\hat{x})]$
LS GAN	$\mathcal{L}_D^{LSGAN} = -\mathbb{E}_{x \sim p_d} [(D(x) - 1)^2] + \mathbb{E}_{\hat{x} \sim p_g} [D(\hat{x})^2]$	$\mathcal{L}_G^{LSGAN} = -\mathbb{E}_{\hat{x} \sim p_g} [(D(\hat{x} - 1))^2]$
DRAGAN	$\mathcal{L}_D^{DRAGAN} = \mathcal{L}_D^{GAN} + \lambda \mathbb{E}_{\hat{x} \sim p_d + \mathcal{N}(0, c)} [(\nabla D(\hat{x}) _2 - 1)^2]$	$\mathcal{L}_G^{DRAGAN} = \mathbb{E}_{\hat{x} \sim p_g} [\log(1 - D(\hat{x}))]$
BEGAN	$\mathcal{L}_D^{BEGAN} = \mathbb{E}_{x \sim p_d} [x - AE(x) _1] - k_t \mathbb{E}_{\hat{x} \sim p_g} [\hat{x} - AE(\hat{x}) _1]$	$\mathcal{L}_G^{BEGAN} = \mathbb{E}_{\hat{x} \sim p_g} [\hat{x} - AE(\hat{x}) _1]$

Figure 7.11: GAN loss function comparison table, taken from the research paper by Mario Lucic et. al. titled ‘Are GANs Created Equal? A Large-Scale study’.

In the next skill, we will apply GAN framework for Image-to-Image translation tasks.

End of Skill-07

Skill 8

Image-to-Image Translation

Image-to-Image translation is the task of picking images from one domain and transforming them into a target domain such that the style or characteristics of the input images are transformed into the target domain while the basic content remains same. Image-to-Image translation has drawn increasing attention and made tremendous progress in recent years because of its wide range of applications in computer vision such as style transfer, object transfiguration, black-n-white to color photos, season transfer, photograph enhancement and so on.

In this skill, we will learn about how Generative Adversarial Networks, or GANs for short, can be leveraged for the task of supervised as well as unsupervised image-to-image translation. Specifically, we will learn about Pix2Pix-GAN and Cycle-GAN for paired as well as unpaired image-to-image translation.

This skill covers the following main topics:

- Image-to-Image Translation with GANs
- Supervised Image-to-Image Translation
- Unsupervised Image-to-Image Translation
- Experiments
- Conclusion

Let's get started.

1. Image-to-Image Translation with GANs

Image-to-Image translation aims to transfer images from a source domain to a target domain while preserving the basic representations of content. Because

the aim of Image-to-Image translation is learning a mapping function between two domains, it is very much related to how GANs work.

We know that the generator part of GAN, learns to generate images of a target domain from gaussian distribution (or noise). Similarly, in the task of image-to-image translation, the generator network can be used to convert an image from source domain to target domain while the discriminator network can be utilized to identify whether a given image comes from the target domain or from the generator network.

Based on the nature of available training data, Image-to-Image translation problem has the following two main types:

- Supervised Image-to-Image Translation
- Unsupervised Image-to-Image Translation

Let's learn more about these two approaches.

1. Supervised Image-to-Image Translation

Supervised Image-to-Image Translation aims to translate images from a source domain into a target domain using: many paired examples of the source domain and the target domain as training data.

Just like GANs learn a generative model of the data, the conditional GANs learn a conditional generative model. A conditional GAN is ideal for the problem of paired image-to-image translation, and we can condition it on input image and train it to generate the corresponding output image.

Pix2Pix Generative Adversarial Network, or Pix2Pix GAN for short, was introduced by *Phillip Isola, et al.* in their 2016 paper titled “*Image-to-Image Translation with Conditional Adversarial Networks*” as a general-purpose framework for the paired image-to-image translation tasks.

Let's learn about Pix2Pix GAN in details.

2.1 Pix2Pix GAN

Pix2Pix is a GAN designed as a general-purpose solution for the task of paired Image-to-Image translation. Pix2Pix generator model is provided with an image of source domain as input and it generates the translated version of it that belongs to the target domain. The discriminator model is presented with the pair of input source images and their translated version, and it is supposed to correctly distinguish the real translation from the synthetic translated version that comes from the generator network.

The generator network is trained via adversarial loss and learns to generate plausible synthetic images of the target domain. The discriminator network is updated directly whereas the generator model is updated through the discriminator network, similar to the traditional GAN training.

During the training process, the generator aims to get better at fooling the discriminator whereas the discriminator seeks to get better at its job of correctly identifying real vs. fake translation of the given input image and thus, not to be fooled by the generator. Both networks keep getting better at their job during the training process.

Let's look at the generator network's design of a Pix2Pix GAN.

2.2 Pix2Pix Generator Design

Pix2Pix GAN generator network must take a high-resolution image as input (an image from the source domain) and generate another high-resolution image as output (the translated version of input image that belongs to the target domain). Another point to keep in mind while designing the pix2pix generator is that, the input and translated image must have same underlying structure but they can differ in surface, appearance and so on to match the characteristics of the target domain.

Encoder-Decoder networks have been applied previously for such scenarios where both inputs as well outputs are high-resolution images. In these networks, input images are passed through multiple down-sampling layers, until a bottleneck and then progressively up-sampled to achieve the target resolution of output images.

One important consideration for the task of image-translation is that the underlying structure of the input images (such as edges) must remain unchanged, whereas in case of encoder-decoder networks, all the information

flows through all the layers including bottleneck layer and this might result in the loss of low-level information.

With these considerations, encoder-decoder network is augmented with skip-connections, which allows the layers to share low-level information directly. So, the Pix2Pix generator network ultimately achieves the general shape of a ‘U-Net’ architecture.

Let’s look at the discriminator design now.

2.3 Pix2Pix Discriminator Design (*Patch GAN*)

The task of a Pix2Pix discriminator network is to correctly distinguish the synthetic translation from the real ones, for a given source input image. The discriminator network of a Pix2Pix GAN, is a deep convolutional neural network that performs conditional image classification. It takes both the source image and the target image as input and predicts the likelihood of whether the target image is a real translation or fake translation of the given source image.

One known fact is that the L1 and L2 loss, often generate blurry images when applied for the task of image generation. It is because, L1 loss fails to capture the high frequency crispness but does a great job at capturing the low frequency details. In order to model the high frequency details (along with the low frequency details coming from L1), the original paper introduces a new design of the discriminator network, also termed as: *Patch GAN*.

Patch GAN only penalizes the structure at the scale of patches. Thus, the task of the discriminator network reduces to classify each $N \times N$ patch as real vs. fake instead of entire image at once and the final output is provided by averaging all the patch responses. We will see the implementation details of Patch GAN later in the experiments.

The paper also demonstrates that N can be much smaller than the original size of the image and still produce high quality result. This is advantageous because with smaller N , the *Patch GAN* has fewer parameters and runs faster, even for very high-resolution input images. This *Patch GAN* discriminator is also known as a type of texture or style loss.

Figure 8.1 shows some impressive results achieved by Pix2Pix GAN for the task of paired image-to-image translation. We can see that the model is

capable of converting the segmentation maps into realistic scenes, filling colors in black and white images, changing a day light picture to night picture, and converting a sketch into a realistic photograph. We will try to replicate similar results in our experiments, but before that we will study the unpaired image-to-image translation.

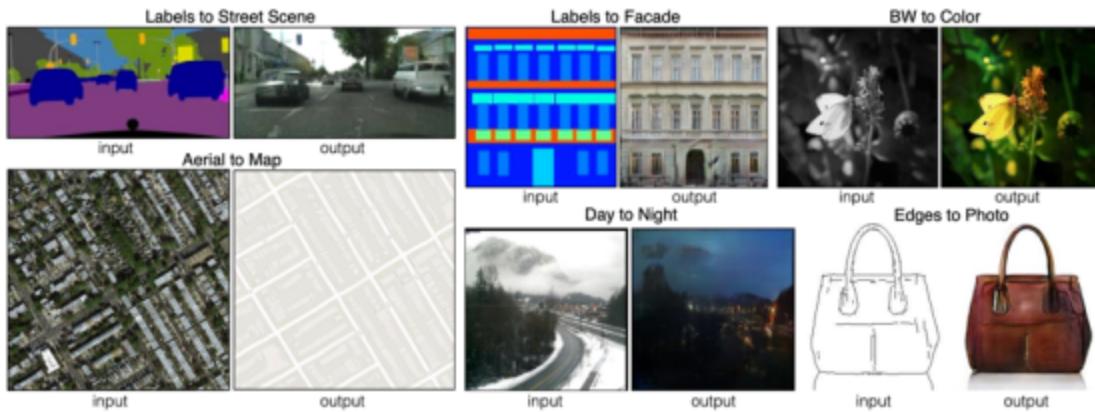


Figure 8.1: Examples of paired image-to-image translation taken from original pix2pix paper

Some of the common application areas of the Pix2Pix GAN are as follows:

- Converting Day photographs to Night light photographs
- Converting Black-n-white to color photographs
- Converting Sketch to photographs
- Translating the aerial view to Google Maps view

Let's now learn about the unpaired image-to-image translation.

1. Unsupervised Image-to-Image Translation

Unsupervised Image-to-Image translation aims to translate the images from a source domain to a target domain where the paired examples are not available. This was an important problem to solve, as for many tasks, the paired training data is close to impossible to create. Some of these datasets can be difficult and expensive to prepare and, in some cases, even impossible such as ancient style paintings.

Cycle GAN is a technique that makes it possible to train image-to-image translation models without the need of paired training examples. The model is trained in an unsupervised manner using a collection of images from the source and the target domains. It means that you only need to collect two sets of examples from two different domains and those two sets are not required to be related in any manner.

Cycle GAN is a simple technique that is quite powerful and achieves visually impressive results for many unsupervised image-to-image translation tasks. Cycle GAN was introduced by *Jun-Yan Zhu, et al.* in 2017 in their paper titled “*Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*”.

Let’s learn about the architectural details of Cycle GAN.

3.1 Cycle GAN architecture design

Cycle GAN is an extension of the GAN architecture and it involves the simultaneous training of two generator models and two discriminator models. One generator is trained to translate the images from first domain to the second domain, while the second generator does the opposite and seeks to translate the images from second domain to the first domain. Two Discriminator models are designed and trained to classify real vs. fake translated images, one for each domain.

One important feature of this architecture is ‘**cycle-consistency**’ and, it does all the magic. The idea is to pass the output of the first generator as input to the second generator, and the output of the second generator should match the original image. The reverse is also true. Cycle consistency is a concept taken from the task of machine translation where a sentence translated from English to French should also translate from French to English and be identical.

Cycle consistency is enforced on the Cycle GAN architecture in form of a loss function called ‘**cycle consistency loss**’ this is added to the adversarial losses of the GAN. With this trick, Cycle GAN achieves encouraging results for many unpaired image-to-image translation tasks. Let’s see how it works.

Suppose, we have a collection of images from two different domains: Domain A and Domain B.

Here,

Dataset A: is a collection of images from domain A

Dataset B: is a collection of images from domain B

Generator A: translates images from domain A to domain B

Generator B: translates images from domain B to domain A

Discriminator A: classifies real vs. fake translated images from domain A

Discriminator B: classifies real vs. fake translated images from domain B

In this case, we can define the cycle consistency loss in two ways, one for each domain. Following are the two variants of cycle consistency loss that can be defined from training a Cycle GAN:

- Forward Cycle Consistency Loss
- Backward Cycle Consistency Loss

Let's see how these loss functions work.

Forward Cycle Consistency loss

Forward cycle consistency loss has the following steps:

- Generator A takes an image from domain A as input
- Generator A outputs an image of domain B
- Generator B takes, the output of Generator A as input
- Generator B translates is back to domain A
- Compare original input image with Generator B output

Similarly, we can define the backward cycle consistency loss.

Backward Cycle Consistency loss

The backward cycle consistency loss is exactly reverse setting from the forward cycle consistency loss. See the following steps:

- Generator B takes an image from domain B as input
- Generator B outputs an image of domain A

- Generator A takes, the output of Generator B as input
- Generator A translates is back to domain B
- Compare original input image with Generator A output

The architectural designs of the generators and the discriminators in a Cycle GAN are quite similar to that of Pix2Pix GAN. The Generators are U-Net like architectures and the discriminators are Patch GANs. We will soon implement a Cycle GAN in the experiments section of this skill.

Figure 8.2 shows some impressive results of Cycle GAN. We can see that Cycle GAN does a good job at style transfer kind of applications such as changing a photograph from summer to winter. Interestingly, Cycle GAN doesn't even require labelled (paired) training dataset for learning image-to-image translation.

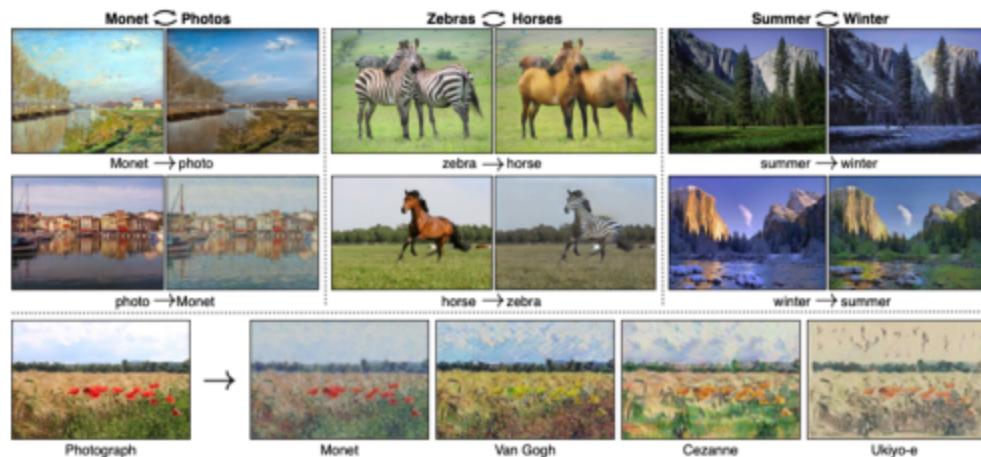


Figure 8.2: Examples of un-paired image-to-image translation taken from Original cycleGAN paper

Some common applications of Cycle GAN are:

- Style Transfer – changing styles of the photographs into ancient paintings.
- Object Transfiguration – such as translating horse images into zebras and vice versa.

- Season Transfer – changing winter style images into summer style images.
- Photograph Enhancement by improving the depth of the field.

Let's now jump into experiments.

1. Experiments

In this skill, we will perform four experiments and verify the capabilities of image-to-image translations models. We will first experiment with Pix2Pix GAN for paired image-to-image translation, and then move to Cycle GAN for unsupervised image translation experimentation.

Specifically, we will perform the following four experiments:

- Pix2Pix GAN for converting black and white images to color images.
- Pix2Pix GAN for converting aerial street view to Google maps style view.
- Cycle GAN for converting apples to oranges and vice versa.
- Cycle GAN for converting horses to zebra and vice versa.

The code samples shown in this skill can be downloaded from the following Github location:

<https://github.com/kartikgill/The-GAN-Book/tree/main/Skill-08>

Let's get started.

PIX2PIX FOR BLACK-N-WHITE TO COLOR IMAGES

Objective

In this experiment, we will develop a Pix2Pix GAN and verify its image-to-image translation capabilities for the use case of converting: black and white images into colorful images. Also, the filled color should be somewhat meaningful.

This experiment has the following steps:

- Importing Libraries
- Download and Unzip Data
- Check few Samples
- Define Generator Network
- Define Discriminator Network
- Define Pix2Pix GAN
- Define Utility Functions
- Training Pix2Pix
- Results

Let's get started.

Step 1: Importing Libraries

Open a Jupyter Notebook with python kernel and import useful python packages in a cell. In this experiment, we will use '*numpy*' for data manipulation, '*matplotlib*' for plotting images and graphs, and *TensorFlow2* for implementing the model architecture.

Checkout the following snippet for importing libraries:

```
import pandas as pd  
import numpy as np  
import glob  
import matplotlib.pyplot as plt  
from tqdm import tqdm_notebook
```

```
%matplotlib inline  
import tensorflow  
print(tensorflow.__version__)  
Output: 2.4.0
```

Let's get the dataset now.

Step 2: Download and Unzip Data

In this step, we will first download the Apple2Orange dataset from the URL given in following python snippet. Apple2Orange is publicly available dataset that was published by the authors of Cycle GAN paper.

The following snippet first extracts the dataset and then reads the file names of train and test partitions into two lists:

```
!wget https://people.eecs.berkeley.edu/~taesung_park/  
CycleGAN/datasets/apple2orange.zip  
!unzip apple2orange.zip  
!ls apple2orange  
apples = glob.glob('apple2orange/trainA/*.jpg')  
oranges = glob.glob('apple2orange/trainB/*.jpg')  
train = apples + oranges  
test = glob.glob('apple2orange/testA/*.jpg') +  
glob.glob('apple2orange/testB/*.jpg')  
len(train), len(test)
```

Output: (2014, 514)

As we see from the above output, the dataset has 2014 training images and 514 test images. Let's check few samples to the data.

Step 3: Check few Samples

In this example, we will choose three random images from the dataset and display their 'black and white' as well as color version. Not that we are converting the colorful images into black and white using OpenCV library.

The following python code shows three samples from the dataset:

```
images = []
for j in range(3):
    file = np.random.choice(train)
    img = cv2.imread(file)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    images.append(img)
    print ('Input Images')
    plt.figure(figsize=(13, 13))
    for j, img in enumerate(images):
        img = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
        plt.subplot(330 + 1 + j)
        plt.imshow(img, cmap='gray')
        plt.axis('off')
        #plt.title(trainY[i])
        plt.show()
    print ('Output Images')
    plt.figure(figsize=(13, 13))
    for j, img in enumerate(images):
        plt.subplot(330 + 1 + j)
        plt.imshow(img)
        plt.axis('off')
        #plt.title(trainY[i])
        plt.show()
```

Output:



Figure 8.3: Few samples from Apple2Orange dataset and their black and white versions.

Figure 8.3 displays three samples from the Apple2Orange dataset along with their black and white versions that we ourselves created using OpenCV library.

Our dataset is now all set. Let's define the model architecture now.

Step 4: Define Generator Network

In this step, we will define the generator network architecture for our Pix2Pix GAN setup. As suggested in the Pix2Pix GAN paper, we will design a *UNet* like architecture for our generator network with skip connections. In our setup, the generator network accepts an image of size 256 x 256 x 1 as input (single channel black and white input image), and passes it through multiple layers of strided *Conv2D* (for down-sampling), *LeakyReLU* activation, and Batch Normalization with a momentum of 0.8. Then the resulting features are passed through a bottleneck layer that is a combination of a *Conv2D* layer and *ReLU* activation. The output of the bottleneck layer along with the outputs of previous encoder layers are passed through the decoder network. Each decoder block passes the input features through the layers of *Conv2DTranspose*, *ReLU* activation, Batch Normalization with a momentum of 0.8, and finally creates a *UNet* like skip connection using a concatenation layer (see the following python code). The final decoder layer,

converts the decoded feature vector into an output image of size 256 x 256 x 3, and uses a ‘*tanh*’ layer to restrict the pixel values of the translated image into a range of [-1, 1].

The following python code define the generator architecture for our setup as discussed:

```
def encoder_layer(input_layer, filters, bn=True):
    x = tensorflow.keras.layers.Conv2D(filters, kernel_size=(4,4), strides=(2,2), padding='same')(input_layer)
    x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
    if bn:
        x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
    return x

def decoder_layer(input_layer, skip_input, filters):
    x = tensorflow.keras.layers.Conv2DTranspose(filters, kernel_size=(4,4), strides=(2,2), padding='same')(input_layer)
    x = tensorflow.keras.layers.Activation('relu')(x)
    x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
    x = tensorflow.keras.layers.concatenate([x, skip_input])
    return x

source_image_input = tensorflow.keras.layers.Input(shape=(256, 256, 1))
e1 = encoder_layer(source_image_input, 64, bn=False)#64
e2 = encoder_layer(e1, 64)#128
e3 = encoder_layer(e2, 64)#256
e4 = encoder_layer(e3, 128)#512
e5 = encoder_layer(e4, 128)#512
e6 = encoder_layer(e5, 128)#512
e7 = encoder_layer(e6, 128)#512
bottle_neck = tensorflow.keras.layers.Conv2D(128, (4,4), strides=(2,2), padding='same')(e7)
b = tensorflow.keras.layers.Activation('relu')(bottle_neck)
```

```

d1 = decoder_layer(b, e7, 128)#512
d2 = decoder_layer(d1, e6, 128)#512
d3 = decoder_layer(d2, e5, 128)#512
d4 = decoder_layer(d3, e4, 128)#512
d5 = decoder_layer(d4, e3, 64)#256
d6 = decoder_layer(d5, e2, 64)#128
d7 = decoder_layer(d6, e1, 64)#64
decoded = tensorflow.keras.layers.Conv2DTranspose(3, kernel_size=
(4,4), strides=(2,2), padding='same')(d7)
translated_image = tensorflow.keras.layers.Activation('tanh')(decoded)
generator_network =
tensorflow.keras.models.Model(inputs=source_image_input,
outputs=translated_image)
print (generator_network.summary())

```

Following is the summary of the generator network and it roughly has 3.7M trainable parameters.

Output:

Model: "model"

Layer (type) Output Shape Param # Connected to

input_1 (InputLayer) [(None, 256, 256, 1) 0]

conv2d (Conv2D) (None, 128, 128, 64) 1088 input_1[0][0]

leaky_re_lu (LeakyReLU) (None, 128, 128, 64) 0 conv2d[0][0]

conv2d_1 (Conv2D) (None, 64, 64, 64) 65600 leaky_re_lu[0][0]

leaky_re_lu_1 (LeakyReLU) (None, 64, 64, 64) 0 conv2d_1[0][0]

batch_normalization (BatchNorma (None, 64, 64, 64) 256 leaky_re_lu_1[0][0]

conv2d_2 (Conv2D) (None, 32, 32, 64) 65600 batch_normalization[0][0]

leaky_re_lu_2 (LeakyReLU) (None, 32, 32, 64) 0 conv2d_2[0][0]

batch_normalization_1 (BatchNor (None, 32, 32, 64) 256 leaky_re_lu_2[0][0]

conv2d_3 (Conv2D) (None, 16, 16, 128) 131200 batch_normalization_1[0][0]

leaky_re_lu_3 (LeakyReLU) (None, 16, 16, 128) 0 conv2d_3[0][0]

batch_normalization_2 (BatchNor (None, 16, 16, 128) 512 leaky_re_lu_3[0][0]

conv2d_4 (Conv2D) (None, 8, 8, 128) 262272 batch_normalization_2[0][0]

leaky_re_lu_4 (LeakyReLU) (None, 8, 8, 128) 0 conv2d_4[0][0]

batch_normalization_3 (BatchNor (None, 8, 8, 128) 512 leaky_re_lu_4[0][0]

conv2d_5 (Conv2D) (None, 4, 4, 128) 262272 batch_normalization_3[0][0]

leaky_re_lu_5 (LeakyReLU) (None, 4, 4, 128) 0 conv2d_5[0][0]

batch_normalization_4 (BatchNor (None, 4, 4, 128) 512 leaky_re_lu_5[0][0]

conv2d_6 (Conv2D) (None, 2, 2, 128) 262272 batch_normalization_4[0][0]

leaky_re_lu_6 (LeakyReLU) (None, 2, 2, 128) 0 conv2d_6[0][0]

batch_normalization_5 (BatchNor (None, 2, 2, 128) 512 leaky_re_lu_6[0][0]

conv2d_7 (Conv2D) (None, 1, 1, 128) 262272 batch_normalization_5[0][0]

activation (Activation) (None, 1, 1, 128) 0 conv2d_7[0][0]

conv2d_transpose (Conv2DTranspo (None, 2, 2, 128) 262272 activation[0][0]

activation_1 (Activation) (None, 2, 2, 128) 0 conv2d_transpose[0][0]

batch_normalization_6 (BatchNor (None, 2, 2, 128) 512 activation_1[0][0]

concatenate (Concatenate) (None, 2, 2, 256) 0 batch_normalization_6[0][0]

batch_normalization_5[0][0]

conv2d_transpose_1 (Conv2DTrans (None, 4, 4, 128) 524416 concatenate[0][0]

activation_2 (Activation) (None, 4, 4, 128) 0 conv2d_transpose_1[0][0]

batch_normalization_7 (BatchNor (None, 4, 4, 128) 512 activation_2[0][0]

concatenate_1 (Concatenate) (None, 4, 4, 256) 0 batch_normalization_7[0][0]

batch_normalization_4[0][0]

conv2d_transpose_2 (Conv2DTrans (None, 8, 8, 128) 524416 concatenate_1[0][0]

activation_3 (Activation) (None, 8, 8, 128) 0 conv2d_transpose_2[0][0]

batch_normalization_8 (BatchNor (None, 8, 8, 128) 512 activation_3[0][0]

concatenate_2 (Concatenate) (None, 8, 8, 256) 0 batch_normalization_8[0][0]

batch_normalization_3[0][0]

conv2d_transpose_3 (Conv2DTrans (None, 16, 16, 128) 524416 concatenate_2[0][0]

activation_4 (Activation) (None, 16, 16, 128) 0 conv2d_transpose_3[0][0]

batch_normalization_9 (BatchNor (None, 16, 16, 128) 512 activation_4[0][0]

concatenate_3 (Concatenate) (None, 16, 16, 256) 0 batch_normalization_9[0][0]

batch_normalization_2[0][0]

conv2d_transpose_4 (Conv2DTrans (None, 32, 32, 64) 262208 concatenate_3[0][0]

activation_5 (Activation) (None, 32, 32, 64) 0 conv2d_transpose_4[0][0]

batch_normalization_10 (BatchNo (None, 32, 32, 64) 256 activation_5[0][0]

concatenate_4 (Concatenate) (None, 32, 32, 128) 0 batch_normalization_10[0][0]

batch_normalization_1[0][0]

conv2d_transpose_5 (Conv2DTrans (None, 64, 64, 64) 131136 concatenate_4[0][0]

activation_6 (Activation) (None, 64, 64, 64) 0 conv2d_transpose_5[0][0]

batch_normalization_11 (BatchNo (None, 64, 64, 64) 256 activation_6[0][0]

concatenate_5 (Concatenate) (None, 64, 64, 128) 0 batch_normalization_11[0][0]

batch_normalization[0][0]

conv2d_transpose_6 (Conv2DTrans (None, 128, 128, 64) 131136 concatenate_5[0][0]

activation_7 (Activation) (None, 128, 128, 64) 0 conv2d_transpose_6[0][0]

batch_normalization_12 (BatchNo (None, 128, 128, 64) 256 activation_7[0][0]

concatenate_6 (Concatenate) (None, 128, 128, 128 0 batch_normalization_12[0][0]

leaky_re_lu[0][0]

conv2d_transpose_7 (Conv2DTrans (None, 256, 256, 3) 6147 concatenate_6[0][0]

activation_8 (Activation) (None, 256, 256, 3) 0 conv2d_transpose_7[0][0]

Total params: 3,684,099

Trainable params: 3,681,411

Non-trainable params: 2,688

None

Our generator network is ready now. Let's define the discriminator network for our Pix2Pix GAN.

Step 5: Define Discriminator Network

In this step, we will define the discriminator network for our Pix2Pix GAN setup. In our setup, the discriminator network accepts two images as input: the source domain image and the target domain image. In our case, first image would have dimensions: 256 x 256 x 1 (as it is a black and white image), and the second image would have dimensions: 256 x 256 x 3 (as this is the translated and colorful version). The first layer of the network combines these two input images using a concatenation layer and then passes them through multiple layers of strided *Conv2D*, *LeakyReLU* activation with an alpha value of 0.2, and a Batch Normalization layers (except for the first layer) with a momentum value of 0.8. The final output of our discriminator network is a patch of size 16 x 16 x 1, as it follows a Patch GAN architecture (as discussed earlier in this skill).

The following python snippet define the discriminator network for our Pix2Pix GAN:

```
def my_conv_layer(input_layer, filters, bn=True):
    x = tensorflow.keras.layers.Conv2D(filters, kernel_size=(4,4), strides=(2,2), padding='same')(input_layer)
    x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
    if bn:
        x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
    return x

source_image_input = tensorflow.keras.layers.Input(shape=(256, 256, 1))
target_image_input = tensorflow.keras.layers.Input(shape=(256, 256, 3))
combined = tensorflow.keras.layers.concatenate([source_image_input, target_image_input])
```

```

x = my_conv_layer(combined, 64, bn=False)#64
x = my_conv_layer(x, 64)#128
x = my_conv_layer(x, 128)#256
x = my_conv_layer(x, 256)#512
patch_features = tensorflow.keras.layers.Conv2D(1, kernel_size=(4,4),
strides=(1,1), padding='same')(x)
discriminator_network = tensorflow.keras.models.Model(inputs=
[source_image_input, target_image_input], outputs=patch_features)
print (discriminator_network.summary())
adam_optimizer =
tensorflow.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5)
discriminator_network.compile(loss='mse', optimizer=adam_optimizer,
metrics=['accuracy'])

```

Following is the summary of the Patch GAN discriminator network. It has roughly 730k trainable parameters:

Output:

Model: "model_2"

Layer (type) Output Shape Param # Connected to

input_4 (InputLayer) [(None, 256, 256, 1) 0

input_5 (InputLayer) [(None, 256, 256, 3) 0

concatenate_8 (Concatenate) (None, 256, 256, 4) 0 input_4[0][0]

input_5[0][0]

conv2d_13 (Conv2D) (None, 128, 128, 64) 4160 concatenate_8[0][0]

leaky_re_lu_11 (LeakyReLU) (None, 128, 128, 64) 0 conv2d_13[0][0]

conv2d_14 (Conv2D) (None, 64, 64, 64) 65600 leaky_re_lu_11[0][0]

leaky_re_lu_12 (LeakyReLU) (None, 64, 64, 64) 0 conv2d_14[0][0]

batch_normalization_16 (BatchNo (None, 64, 64, 64) 256 leaky_re_lu_12[0][0]

conv2d_15 (Conv2D) (None, 32, 32, 128) 131200 batch_normalization_16[0][0]

leaky_re_lu_13 (LeakyReLU) (None, 32, 32, 128) 0 conv2d_15[0][0]

batch_normalization_17 (BatchNo (None, 32, 32, 128) 512 leaky_re_lu_13[0][0]

conv2d_16 (Conv2D) (None, 16, 16, 256) 524544 batch_normalization_17[0][0]

leaky_re_lu_14 (LeakyReLU) (None, 16, 16, 256) 0 conv2d_16[0][0]

batch_normalization_18 (BatchNo (None, 16, 16, 256) 1024 leaky_re_lu_14[0][0]

conv2d_17 (Conv2D) (None, 16, 16, 1) 4097 batch_normalization_18[0][0]

Total params: 731,393

Trainable params: 730,497

Non-trainable params: 896

None

Our discriminator network is now ready. Let's define the combined Pix2Pix model now.

Step 6: Define Pix2Pix GAN

Pix2Pix GAN model is a combination of the generator and the discriminator network defined earlier. Here, we combine both models by passing the output of the generator network as input to the discriminator network along with the source input image. The weights of the discriminator network are kept frozen in the combined setup, as the goal of this combined setup is to only update the weights of the generator network. The gradient however flows through the frozen discriminator network first and then updates the generator weights.

Final Pix2Pix GAN has two outputs: validity output from the Patch GAN discriminator (real vs. fake), and the output of the generator network. The output of the generator network here would be used for calculating pixel-wise loss with the real translated image. For Patch GAN output, we can use '*mean squared error*' as loss function or L2 loss and for the translation output we will use '*mean absolute error*' or L1 loss. The translation output would have higher priority and thus it has a loss-weight of 100.

The following python code defines the Pix2Pix GAN setup and also compiles the model:

```
discriminator_network.trainable=False  
g_output = generator_network(source_image_input)  
d_output = discriminator_network([source_image_input, g_output])  
pix2pix = tensorflow.keras.models.Model(inputs=source_image_input,  
outputs=[d_output, g_output])  
pix2pix.summary()  
# Compiling Model  
pix2pix.compile(loss=['mse', 'mae'], optimizer=adam_optimizer,  
loss_weights=[1, 100])
```

Following is the summary of the Pix2Pix GAN:

Output:

Model: "model_3"

Layer (type) Output Shape Param # Connected to

=====

=====

input_4 (InputLayer) [(None, 256, 256, 1) 0

model (Functional) (None, 256, 256, 3) 3684099 input_4[0][0]

```
model_2 (Functional) (None, 16, 16, 1) 731393 input_4[0][0]
```

```
model[0][0]
```

```
Total params: 4,415,492
```

```
Trainable params: 3,681,411
```

```
Non-trainable params: 734,081
```

Our Pix2Pix GAN setup is ready now. Let's work on the training piece now.

Step 7: Define Utility Functions

In this step, we will define some utility functions that will help us in getting data for the training setup. Specifically, we will define utility functions for: generating the generator outputs, getting a random batch of data consisting of pairwise input and output samples, and a function to display the results of the generator model along with the real translation version for randomly chosen images from the test dataset. Note that the function that generates the batches of the dataset; also normalizes the pixel values into a range [-1, 1] of all the images, as expected by the network.

See the following python code for utility functions:

```
def get_predictions(input_sample, generator_network):
    input_sample = np.expand_dims(input_sample, axis=0)
    output_sample = generator_network.predict_on_batch(input_sample)
    return output_sample

def get_generated_samples(generator_network, imgs_input):
```

```
generated_samples = generator_network.predict_on_batch(imgs_input)
return generated_samples

def get_img_samples(batch_size):
    random_files = np.random.choice(train, size=batch_size)
    imgs_input = []
    imgs_output = []
    for file in random_files:
        img = cv2.imread(file)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        black_and_white = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
        black_and_white = np.expand_dims(black_and_white, axis=-1)
        imgs_input.append((black_and_white-127.5)/127.5)
        imgs_output.append((img-127.5)/127.5)
    imgs_input = np.array(imgs_input)
    imgs_output = np.array(imgs_output)
    return imgs_input, imgs_output

def show_generator_results(generator_network):
    images = []
    for j in range(3):
        file = np.random.choice(test)
        img = cv2.imread(file)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        images.append(img)
    print ('Input Images')
    plt.figure(figsize=(13, 13))
    for j, img in enumerate(images):
        black_and_white = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
        plt.subplot(330 + 1 + j)
```

```

plt.imshow(black_and_white, cmap='gray')
plt.axis('off')
#plt.title(trainY[i])
plt.show()
print ('Predicted Output Images')
plt.figure(figsize=(13, 13))
for j, img in enumerate(images):
    black_and_white = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    black_and_white = (black_and_white-127.5)/127.5
    output = get_predictions(black_and_white, generator_network)[0]
    output = (output+1.0)/2.0
    plt.subplot(330 + 1 + j)
    plt.imshow(output)
    plt.axis('off')
    #plt.title(trainY[i])
    plt.show()
print ('Real Output Images')
plt.figure(figsize=(13, 13))
for j, img in enumerate(images):
    plt.subplot(330 + 1 + j)
    plt.imshow(img)
    plt.axis('off')
    #plt.title(trainY[i])
    plt.show()

```

Our utility functions are ready now. Let's start with the training process.

Step 8: Training Pix2Pix

In this step, we will write the training iteration loop for our Pix2Pix GAN. In each training step, we will first update the weights of the discriminator network using two separate batches of the real translation outputs and the

fake translation outputs. It is advised to train the discriminator on separate batches of real and fake samples. The discriminator output is a patch of size 16 x 16 x 1, and it is compared with a similar patch of similar size. The patch of the real images is filled with ones, while the patch of the fake images is filled with zeroes. Then, we freeze the weights of the discriminator network and update the weights of the generator network by passing a batch of input images, output images and a patch of real images (filled with ones). Here, we pass a patch of real images to make the discriminator believe that it is the real translation of the input image and calculate the loss value.

The following python code defines the training setup for Pix2Pix GAN:

```
for i in range(0, epochs):
    if (i%10 == 0):
        op = show_generator_results(generator_network)
    for j in range(steps):
        imgs_input, imgs_output = get_img_samples(batch_size=batch_size//2)
        generated_imgs_output = get_generated_samples(generator_network,
        imgs_input)
        fake_patch = np.zeros((batch_size//2, 16, 16, 1))
        real_patch = np.ones((batch_size//2, 16, 16, 1))
        # Updating Discriminator weights
        discriminator_network.trainable=True
        loss_d1 = discriminator_network.train_on_batch([imgs_input,
        imgs_output], real_patch)
        loss_d2 = discriminator_network.train_on_batch([imgs_input,
        generated_imgs_output], fake_patch)
        loss_d = (np.add(loss_d1, loss_d2))/2.0

        imgs_input, imgs_output = get_img_samples(batch_size=batch_size)
        # Make the Discriminator believe that these are real samples and calculate
        loss to train the generator
```

```

real_patch = np.ones((batch_size, 16, 16, 1))
# Updating Generator weights
discriminator_network.trainable=False
loss_g, _, _ = pix2pix.train_on_batch(imgs_input, [real_patch,
imgs_output])
if j%10 == 0:
    print ("Epoch:%.0f, Step:%.0f, D-Loss:%.3f, D-Acc:%.3f, G-
Loss:%.3f"%(i,j,loss_d[0],loss_d[1]*100,loss_g))
    generator_network.save("/content/gdrive/MyDrive/GAN_datasets/black_
n_white_to_color")

```

Following are the training logs:

Output:

```

Epoch:0, Step:0, D-Loss:2.810, D-Acc:47.664, G-Loss:65.759
Epoch:0, Step:10, D-Loss:0.626, D-Acc:53.652, G-Loss:27.292
Epoch:1, Step:0, D-Loss:0.424, D-Acc:52.875, G-Loss:22.069
Epoch:1, Step:10, D-Loss:0.375, D-Acc:51.496, G-Loss:22.323
Epoch:2, Step:0, D-Loss:0.360, D-Acc:49.445, G-Loss:20.862
Epoch:2, Step:10, D-Loss:0.331, D-Acc:49.316, G-Loss:19.165
Epoch:3, Step:0, D-Loss:0.310, D-Acc:49.117, G-Loss:18.850
Epoch:3, Step:10, D-Loss:0.297, D-Acc:49.539, G-Loss:18.094
Epoch:4, Step:0, D-Loss:0.300, D-Acc:48.461, G-Loss:17.526
. . .
. . .
. . .
. . .

Epoch:41, Step:0, D-Loss:0.020, D-Acc:99.789, G-Loss:7.595
Epoch:41, Step:10, D-Loss:0.026, D-Acc:99.836, G-Loss:7.754
Epoch:42, Step:0, D-Loss:0.018, D-Acc:99.910, G-Loss:7.205
Epoch:42, Step:10, D-Loss:0.013, D-Acc:99.957, G-Loss:6.543

```

Epoch:43, Step:0, D-Loss:0.024, D-Acc:99.980, G-Loss:7.678

Our Pix2Pix GAN is successfully trained now. Let's check out some results.

Step 9: Results

In this step, we will check some of the generated translation results from our Pix2Pix GAN model. We can utilize the '*show_generator_results*' function for checking the generator results on some random test images. See the following code:

```
for i in range(5):  
    show_generator_results(generator_network)
```

Figure 8.4 shows the results of the Pix2Pix GAN generator. The top row represents the input images that we passed to the model. Middle row shows the generator outputs of our Pix2Pix GAN. The bottom row shows the real translation of the input images. We can clearly see that the generator network is successfully able to fill colors in black and white images. Important thing to notice here is that it fills red color in case of apples but orange color in case of the oranges. So, the network is also able to understand the shapes and properties of different types of images. As it a pretty big model, we did not train it for long otherwise the results would have been even better.

Output:

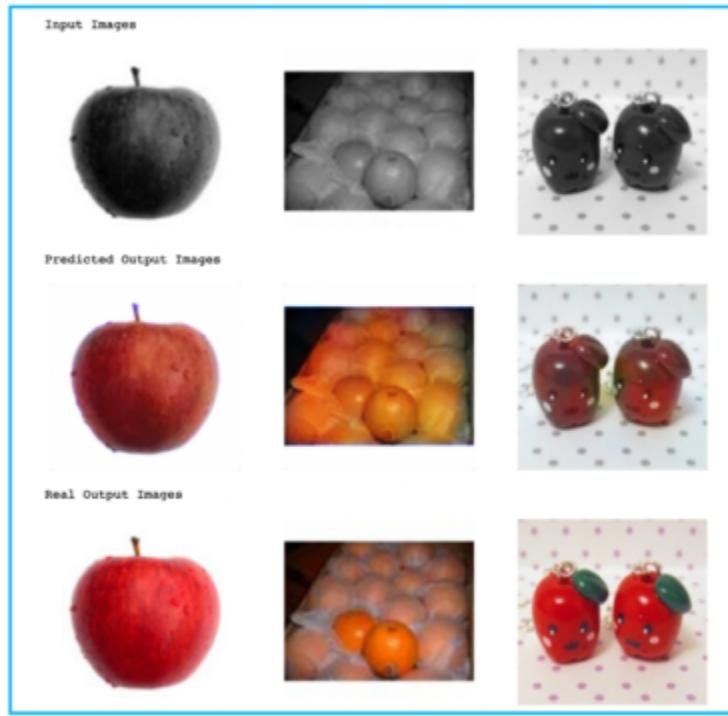


Figure 8.4: Pix2Pix GAN results. (Top) input source images. (Middle) Images translated by Pix2Pix generator, and (Bottom) Real translated version of the images.

We just saw that the Pix2Pix GAN is capable of converting the black and white images into colorful images. Let's do one more experiment to verify their results.

PIX2PIX FOR GOOGLE MAPS EXPERIMENT

Objective

In this experiment, we will train a Pix2Pix GAN for another image-to-image translation task of converting the aerial street view to Google Maps style street view.

This experiment has the following steps:

- Importing Libraries
- Download and Unzip Data
- Check few Samples
- Define Generator Network
- Define Discriminator Network
- Define Pix2Pix GAN
- Define Utility Functions
- Training Pix2Pix
- Results

Let's get started.

Step 1: Importing Libraries

Open a Jupyter Notebook with python kernel and import useful python packages in a cell. In this experiment, we will use '*numpy*' for data manipulation, '*matplotlib*' for plotting images and graphs, and *TensorFlow2* for implementing the model architecture later.

Checkout the following snippet for importing libraries:

```
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
from tqdm import tqdm_notebook  
%matplotlib inline  
import tensorflow  
print(tensorflow.__version__)
```

Output: 2.4.0

Let's get the dataset now.

Step 2: Download and Unzip Data

In this step, we will first download the maps dataset from the URL given in following python snippet. Maps dataset is a publicly available dataset that was published by the authors of Pix2Pix GAN paper.

The following snippet first extracts the dataset and then reads the file names of train and test partitions into two lists:

```
!wget http://efrosgans.eecs.berkeley.edu/pix2pix/datasets/maps.tar.gz  
!tar -zxf maps.tar.gz  
!ls  
import glob  
train_files = glob.glob('maps/train/*.jpg')  
test_files = glob.glob('maps/val/*.jpg')  
len(train_files), len(test_files)  
Output: (1096, 1098)
```

We can see that the dataset has 1096 training images and 1098 test images. Let's check out some samples from the dataset.

Step 3: Check few Samples

In this example, we will choose three random images from the dataset and display their 'aerial view' as well as the Google Maps style version. Note that the source and target images come concatenated in a single image, so we need to cut each image from the middle such that left part represents the aerial view and the right part presents the Google Maps style output of the same.

The following python code shows three samples from the dataset:

```
maps = []  
for j in range(3):  
    file = np.random.choice(train_files)  
    map = cv2.imread(file)
```

```

map = cv2.cvtColor(map, cv2.COLOR_BGR2RGB)
maps.append(map)
print ('Input Images')
plt.figure(figsize=(15, 15))
for j, map in enumerate(maps):
    map1 = map[:, :map.shape[1]//2]
    plt.subplot(330 + 1 + j)
    plt.imshow(map1)
    plt.axis('off')
plt.show()
print ('Output Images')
plt.figure(figsize=(15, 15))
for j, map in enumerate(maps):
    map2 = map[:, map.shape[1]//2:]
    plt.subplot(330 + 1 + j)
    plt.imshow(map2)
    plt.axis('off')
plt.show()

```

Figure 8.5 shows three samples from the Maps dataset. The first row shows the aerial street view images and the second row shows the corresponding Google Maps style street view. Our job is to convert the style of the images from first row into a style similar to as shown in the second row.

Output:

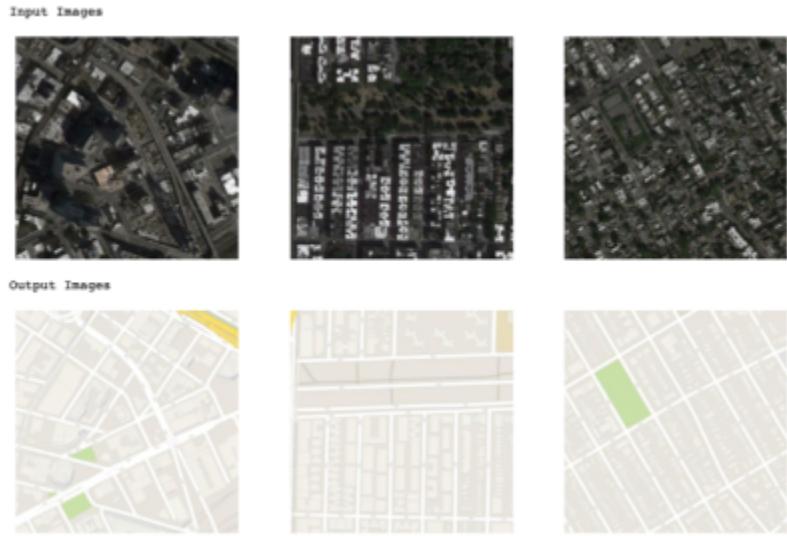


Figure 8.5: Three samples from the Maps Dataset of Pix2Pix GAN experiments

Our dataset is all set. Let's define the model architecture now.

Step 4: Define Generator Network

In this step, we will define the generator network architecture for our Pix2Pix GAN setup. As suggested in the Pix2Pix GAN paper, we will design a *UNet* like architecture for our generator network with skip connections. In our setup, the generator network accepts an image of size $256 \times 256 \times 3$ as input (an image from the source domain), and passes it through multiple layers of strided *Conv2D* (for down-sampling), *LeakyReLU* activation with an alpha value of 0.2, and Instance Normalization. Then the resulting features are passed through a bottleneck layer that is a combination of a *Conv2D* layer and *ReLU* activation. The output of the bottleneck layer along with the outputs of previous encoder layers are passed through the decoder network. Each decoder block passes the input features through the layers of *Conv2DTranspose*, *ReLU* activation, Instance Normalization, and finally creates a *UNet* like skip connection using a concatenation layer (see the following python code). The final decoder layer, converts the decoded feature vector into an output image of size $256 \times 256 \times 3$, and uses a '*tanh*' layer to restrict the pixel values of the translated image into a range of [-1, 1].

The following python code define the generator architecture for our setup as discussed:

```
import tensorflow_addons as tfa

def encoder_layer(input_layer, filters, bn=True):
    x = tensorflow.keras.layers.Conv2D(filters, kernel_size=(4,4),\
        strides=(2,2), padding='same')(input_layer)
    x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
    if bn:
        x = tfa.layers.InstanceNormalization()(x)
    return x

def decoder_layer(input_layer, skip_input, filters):
    x = tensorflow.keras.layers.Conv2DTranspose(filters, \
        kernel_size=(4,4), strides=(2,2), padding='same')(input_layer)
    x = tensorflow.keras.layers.Activation('relu')(x)
    x = tfa.layers.InstanceNormalization()(x)
    x = tensorflow.keras.layers.Concatenate()([x, skip_input])
    return x

source_image_input = tensorflow.keras.layers.Input(shape=(256, 256, 3))
e1 = encoder_layer(source_image_input, 64, bn=False)#64
e2 = encoder_layer(e1, 128)#128
e3 = encoder_layer(e2, 256)#256
e4 = encoder_layer(e3, 512)#512
e5 = encoder_layer(e4, 512)#512
e6 = encoder_layer(e5, 512)#512
e7 = encoder_layer(e6, 512)#512
bottle_neck = tensorflow.keras.layers.Conv2D(512, \
    (4,4), strides=(2,2), padding='same')(e7)
b = tensorflow.keras.layers.Activation('relu')(bottle_neck)
d1 = decoder_layer(b, e7, 512)#512
```

```
d2 = decoder_layer(d1, e6, 512)#512
d3 = decoder_layer(d2, e5, 512)#512
d4 = decoder_layer(d3, e4, 512)#512
d5 = decoder_layer(d4, e3, 256)#256
d6 = decoder_layer(d5, e2, 128)#128
d7 = decoder_layer(d6, e1, 64)#64
decoded = tensorflow.keras.layers.Conv2DTranspose(3, \
kernel_size=(4,4), strides=(2,2), padding='same')(d7)
translated_image = tensorflow.keras.layers.Activation('tanh')(decoded)
generator_network = tensorflow.keras.models.Model(\n
inputs=source_image_input, outputs=translated_image)
print (generator_network.summary())
```

Following is the summary of the generator network. It roughly has 54M trainable parameters.

Output:

Model: "model"

Layer (type) Output Shape Param # Connected to

input_1 (InputLayer) [(None, 256, 256, 3) 0

conv2d (Conv2D) (None, 128, 128, 64) 3136 input_1[0][0]

leaky_re_lu (LeakyReLU) (None, 128, 128, 64) 0 conv2d[0][0]

conv2d_1 (Conv2D) (None, 64, 64, 128) 131200 leaky_re_lu[0][0]

leaky_re_lu_1 (LeakyReLU) (None, 64, 64, 128) 0 conv2d_1[0][0]

instance_normalization (Instanc (None, 64, 64, 128) 256 leaky_re_lu_1[0][0]

conv2d_2 (Conv2D) (None, 32, 32, 256) 524544 instance_normalization[0][0]

leaky_re_lu_2 (LeakyReLU) (None, 32, 32, 256) 0 conv2d_2[0][0]

instance_normalization_1 (Insta (None, 32, 32, 256) 512 leaky_re_lu_2[0][0]

conv2d_3 (Conv2D) (None, 16, 16, 512) 2097664 instance_normalization_1[0][0]

leaky_re_lu_3 (LeakyReLU) (None, 16, 16, 512) 0 conv2d_3[0][0]

instance_normalization_2 (Insta (None, 16, 16, 512) 1024 leaky_re_lu_3[0][0]

conv2d_4 (Conv2D) (None, 8, 8, 512) 4194816 instance_normalization_2[0][0]

leaky_re_lu_4 (LeakyReLU) (None, 8, 8, 512) 0 conv2d_4[0][0]

instance_normalization_3 (Insta (None, 8, 8, 512) 1024 leaky_re_lu_4[0][0])

conv2d_5 (Conv2D) (None, 4, 4, 512) 4194816 instance_normalization_3[0][0]

leaky_re_lu_5 (LeakyReLU) (None, 4, 4, 512) 0 conv2d_5[0][0]

instance_normalization_4 (Insta (None, 4, 4, 512) 1024 leaky_re_lu_5[0][0])

conv2d_6 (Conv2D) (None, 2, 2, 512) 4194816 instance_normalization_4[0][0]

leaky_re_lu_6 (LeakyReLU) (None, 2, 2, 512) 0 conv2d_6[0][0]

instance_normalization_5 (Insta (None, 2, 2, 512) 1024 leaky_re_lu_6[0][0])

conv2d_7 (Conv2D) (None, 1, 1, 512) 4194816 instance_normalization_5[0][0]

activation (Activation) (None, 1, 1, 512) 0 conv2d_7[0][0]

conv2d_transpose (Conv2DTranspo (None, 2, 2, 512) 4194816 activation[0][0]

activation_1 (Activation) (None, 2, 2, 512) 0 conv2d_transpose[0][0]

instance_normalization_6 (Insta (None, 2, 2, 512) 1024 activation_1[0][0]

concatenate (Concatenate) (None, 2, 2, 1024) 0 instance_normalization_6[0][0]

instance_normalization_5[0][0]

conv2d_transpose_1 (Conv2DTrans (None, 4, 4, 512) 8389120 concatenate[0][0]

activation_2 (Activation) (None, 4, 4, 512) 0 conv2d_transpose_1[0][0]

instance_normalization_7 (Insta (None, 4, 4, 512) 1024 activation_2[0][0]

concatenate_1 (Concatenate) (None, 4, 4, 1024) 0 instance_normalization_7[0][0]
instance_normalization_4[0][0]

conv2d_transpose_2 (Conv2DTrans (None, 8, 8, 512) 8389120 concatenate_1[0][0]

activation_3 (Activation) (None, 8, 8, 512) 0 conv2d_transpose_2[0][0]

instance_normalization_8 (Insta (None, 8, 8, 512) 1024 activation_3[0][0]

concatenate_2 (Concatenate) (None, 8, 8, 1024) 0 instance_normalization_8[0][0]
instance_normalization_3[0][0]

conv2d_transpose_3 (Conv2DTrans (None, 16, 16, 512) 8389120 concatenate_2[0][0]

activation_4 (Activation) (None, 16, 16, 512) 0 conv2d_transpose_3[0][0]

instance_normalization_9 (Insta (None, 16, 16, 512) 1024 activation_4[0][0]

concatenate_3 (Concatenate) (None, 16, 16, 1024) 0 instance_normalization_9[0][0]

instance_normalization_2[0][0]

conv2d_transpose_4 (Conv2DTrans (None, 32, 32, 256) 4194560 concatenate_3[0][0]

activation_5 (Activation) (None, 32, 32, 256) 0 conv2d_transpose_4[0][0]

instance_normalization_10 (Inst (None, 32, 32, 256) 512 activation_5[0][0]

concatenate_4 (Concatenate) (None, 32, 32, 512) 0 instance_normalization_10[0][0]

instance_normalization_1[0][0]

conv2d_transpose_5 (Conv2DTrans (None, 64, 64, 128) 1048704 concatenate_4[0][0]

activation_6 (Activation) (None, 64, 64, 128) 0 conv2d_transpose_5[0][0]

instance_normalization_11 (Inst (None, 64, 64, 128) 256 activation_6[0][0]

concatenate_5 (Concatenate) (None, 64, 64, 256) 0 instance_normalization_11[0][0]

instance_normalization[0][0]

conv2d_transpose_6 (Conv2DTrans (None, 128, 128, 64) 262208 concatenate_5[0][0]

activation_7 (Activation) (None, 128, 128, 64) 0 conv2d_transpose_6[0][0]

instance_normalization_12 (Inst (None, 128, 128, 64) 128 activation_7[0][0]

concatenate_6 (Concatenate) (None, 128, 128, 128 0 instance_normalization_12[0][0]

leaky_re_lu[0][0]

conv2d_transpose_7 (Conv2DTrans (None, 256, 256, 3) 6147 concatenate_6[0][0]

activation_8 (Activation) (None, 256, 256, 3) 0 conv2d_transpose_7[0][0]

Total params: 54,419,459

Trainable params: 54,419,459

Non-trainable params: 0

None

Let's define the discriminator network now.

Step 5: Define Discriminator Network

In this step, we will define the discriminator network for our Pix2Pix GAN setup. In our setup, the discriminator network accepts two images as input: the source domain image and the target domain image. In our case, both input images have dimensions: 256 x 256 x 3. The first layer of the network combines these two input images using a concatenation layer and then passes them through multiple layers of strided *Conv2D*, *LeakyReLU* activation with an alpha value of 0.2, and an Instance Normalization layer (except for the first layer). The final output of our discriminator network is a patch of size 16 x 16 x 1, as it follows a Patch GAN architecture (as discussed earlier in this skill).

The following python snippet defines the discriminator network for our Pix2Pix GAN:

```
def my_conv_layer(input_layer, filters, bn=True):
    x = tensorflow.keras.layers.Conv2D(filters, kernel_size=(4,4),\
        strides=(2,2), padding='same')(input_layer)
    x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
    if bn:
        x = tfa.layers.InstanceNormalization()(x)
    return x

source_image_input = tensorflow.keras.layers.Input(shape=(256, 256, 3))
target_image_input = tensorflow.keras.layers.Input(shape=(256, 256, 3))
combined = tensorflow.keras.layers.concatenate([source_image_input, target_image_input])
x = my_conv_layer(combined, 64, bn=False)#64
x = my_conv_layer(x, 128)#128
x = my_conv_layer(x, 256)#256
x = my_conv_layer(x, 512)#512
```

```
patch_features = tensorflow.keras.layers.Conv2D(1,\n    kernel_size=(4,4), strides=(1,1), padding='same')(x)\n\ndiscriminator_network = tensorflow.keras.models.Model(inputs\\n    =[source_image_input, target_image_input], outputs=patch_features)\nprint (discriminator_network.summary())\nadam_optimizer = tensorflow.keras.optimizers.Adam(\n    learning_rate=0.0002, beta_1=0.5)\ndiscriminator_network.compile(loss='mse', \\n    optimizer=adam_optimizer, metrics=['accuracy'])
```

Following is the summary of the discriminator network. It roughly has 2.7M trainable parameters.

Output:

Model: "model_1"

Layer (type) Output Shape Param # Connected to

input_2 (InputLayer) [(None, 256, 256, 3) 0]

input_3 (InputLayer) [(None, 256, 256, 3) 0]

concatenate_7 (Concatenate) (None, 256, 256, 6) 0 input_2[0][0]

input_3[0][0]

conv2d_8 (Conv2D) (None, 128, 128, 64) 6208 concatenate_7[0][0]

leaky_re_lu_7 (LeakyReLU) (None, 128, 128, 64) 0 conv2d_8[0][0]

conv2d_9 (Conv2D) (None, 64, 64, 128) 131200 leaky_re_lu_7[0][0]

leaky_re_lu_8 (LeakyReLU) (None, 64, 64, 128) 0 conv2d_9[0][0]

instance_normalization_13 (Inst (None, 64, 64, 128) 256 leaky_re_lu_8[0][0])

conv2d_10 (Conv2D) (None, 32, 32, 256) 524544 instance_normalization_13[0][0]

leaky_re_lu_9 (LeakyReLU) (None, 32, 32, 256) 0 conv2d_10[0][0]

instance_normalization_14 (Inst (None, 32, 32, 256) 512 leaky_re_lu_9[0][0])

conv2d_11 (Conv2D) (None, 16, 16, 512) 2097664 instance_normalization_14[0][0]

leaky_re_lu_10 (LeakyReLU) (None, 16, 16, 512) 0 conv2d_11[0][0]

```
instance_normalization_15 (Inst (None, 16, 16, 512) 1024 leaky_re_lu_10[0][0]
```

```
conv2d_12 (Conv2D) (None, 16, 16, 1) 8193 instance_normalization_15[0][0]
```

Total params: 2,769,601

Trainable params: 2,769,601

Non-trainable params: 0

None

Our discriminator network is ready now. Let's define the Pix2Pix GAN.

Step 6: Define Pix2Pix GAN

Pix2Pix GAN model is a combination of the generator and the discriminator network defined earlier. Here, we combine both models by passing the output of the generator network as input to the discriminator network along with the source input image. The weights of the discriminator network are kept frozen in the combined setup, as the goal of this combined setup is to only update the weights of the generator network. The gradient, however, flows through the frozen discriminator network first and then updates the generator weights.

Final Pix2Pix GAN model has two outputs: validity output from the Patch GAN discriminator (real vs. fake), and the output of the generator network. The output of the generator network here would be used for calculating pixel-wise loss with the real translated image. For Patch GAN output, we can use '*mean squared error*' as loss function or L2 loss and for the translation output

we will use ‘*mean absolute error*’ or L1 loss. The translation output would have higher priority and thus it has a loss-weight of 100.

The following python code defines the Pix2Pix GAN setup and also compiles the model:

```
discriminator_network.trainable=False
g_output = generator_network(source_image_input)
d_output = discriminator_network([source_image_input, g_output])
pix2pix = tensorflow.keras.models.Model(
    inputs=source_image_input, outputs=[d_output, g_output])
pix2pix.summary()
# Compiling Pix2Pix
pix2pix.compile(loss=['mse', 'mae'], \
optimizer=adam_optimizer, loss_weights=[1, 100])
```

Following is the summary of the combined Pix2Pix GAN:

Output:

Model: "model_2"

Layer (type) Output Shape Param # Connected to

=====

=====

input_2 (InputLayer) [(None, 256, 256, 3) 0

model (Functional) (None, 256, 256, 3) 54419459 input_2[0][0]

model_1 (Functional) (None, 16, 16, 1) 2769601 input_2[0][0]

```
model[0][0]
=====
Total params: 57,189,060
Trainable params: 54,419,459
Non-trainable params: 2,769,601
```

Our Pix2Pix GAN setup is ready now. Let's work on the training piece.

Step 7: Define Utility Functions

In this step, we will define some utility functions that will help us in getting data for the training setup. Specifically, we will define utility functions for: generating the generator outputs, getting a random batch of data consisting of pairwise input and output samples, and a function to display the results of the generator model along with the real translation version for randomly chosen images from the test dataset. Note that the function that generates the batches of the dataset; also normalizes the pixel values into a range [-1, 1] of all the images, as expected by the network.

See the following python code for utility functions:

```
def get_predictions(input_sample, generator_network):
    input_sample = np.expand_dims(input_sample, axis=0)
    output_sample = generator_network.predict_on_batch(input_sample)
    return output_sample

def get_generated_samples(generator_network, maps_input):
    generated_samples = generator_network.predict_on_batch(maps_input)
    return generated_samples

def get_map_samples(batch_size):
    random_files = np.random.choice(train_files, size=batch_size)
    maps_input = []
```

```
maps_output = []
for file in random_files:
    map = cv2.imread(file)
    map = cv2.cvtColor(map, cv2.COLOR_BGR2RGB)
    map1 = map[:, :map.shape[1]//2]
    map2 = map[:, map.shape[1]//2:]
    map1 = cv2.resize(map1, (256, 256))
    map2 = cv2.resize(map2, (256, 256))
    maps_input.append((map1-127.5)/127.5)
    maps_output.append((map2-127.5)/127.5)
maps_input = np.array(maps_input)
maps_output = np.array(maps_output)
return maps_input, maps_output

def show_generator_results(generator_network):
    maps = []
    for j in range(3):
        file = np.random.choice(test_files)
        map = cv2.imread(file)
        map = cv2.cvtColor(map, cv2.COLOR_BGR2RGB)
        maps.append(map)
        print ('Input Images')
        plt.figure(figsize=(13, 13))
        for j, map in enumerate(maps):
            map1 = map[:, :map.shape[1]//2]
            map1 = cv2.resize(map1, (256, 256))
            plt.subplot(330 + 1 + j)
            plt.imshow(map1)
            plt.axis('off')
```

```

# plt.title(trainY[i])
plt.show()
print ('Predicted Output Images')
plt.figure(figsize=(13, 13))
for j, map in enumerate(maps):
    map1 = map[:, :map.shape[1]//2]
    map1 = cv2.resize(map1, (256, 256))
    map1 = (map1-127.5)/127.5
    output = get_predictions(map1, generator_network)[0]
    output = (output+1.0)/2.0
    plt.subplot(330 + 1 + j)
    plt.imshow(output)
    plt.axis('off')
# plt.title(trainY[i])
plt.show()
print ('Real Output Images')
plt.figure(figsize=(13, 13))
for j, map in enumerate(maps):
    map2 = map[:, map.shape[1]//2:]
    map2 = cv2.resize(map2, (256, 256))
    plt.subplot(330 + 1 + j)
    plt.imshow(map2)
    plt.axis('off')
# plt.title(trainY[i])
plt.show()

```

We are now all set to start training our model.

Step 8: Training Pix2Pix

In this step, we will write the training iteration loop for our Pix2Pix GAN. In each training step, we will first update the weights of the discriminator network using two separate batches of the real translation outputs and the fake translation outputs. It is advised to train the discriminator on separate batches of real and fake samples. The discriminator output is a patch of size 16 x 16 x 1, and it is compared with a similar patch of similar size. The patch of the real images is filled with ones, while the patch of the fake images is filled with zeroes. Then, we freeze the weights of the discriminator network and update the weights of the generator network by passing a batch of input images, output images and a patch of real images (filled with ones). Here, we pass a patch of real images to make the discriminator believe that it is the real translation of the input image and calculate the loss value.

The following python code defines the training setup for Pix2Pix GAN:

```
epochs = 500
batch_size = 1
steps = 1096
for i in range(0, epochs):
    if (i%5 == 0):
        show_generator_results(generator_network)
    for j in range(steps):
        maps_input, maps_output = get_map_samples(batch_size=batch_size)
        generated_maps_output = get_generated_samples(generator_network,
                                                       maps_input)
        fake_patch = np.zeros((batch_size, 16, 16, 1))
        real_patch = np.ones((batch_size, 16, 16, 1))
        # Updating Discriminator weights
        discriminator_network.trainable=True
        loss_d1 = discriminator_network.train_on_batch(\n            [maps_input, maps_output], real_patch)
        loss_d2 = discriminator_network.train_on_batch([maps_input,\n            generated_maps_output], fake_patch)
```

```

loss_d = (np.add(loss_d1, loss_d2))/2.0
maps_input, maps_output = get_map_samples(batch_size=batch_size)
# Make the Discriminator believe that these are real
#samples and calculate loss to train the generator
real_patch = np.ones((batch_size, 16, 16, 1))
# Updating Generator weights
discriminator_network.trainable=False
loss_g, _, _ = pix2pix.train_on_batch(maps_input, \
[real_patch, maps_output])
if j%100 == 0:
    print ("Epoch:%.0f, Step:%.0f, D-Loss:%.3f, D-Acc:%.3f,\n"
    "G-Loss:%.3f"%(i,j,loss_d[0],loss_d[1]*100,loss_g))

```

Following are the training logs:

Output:

Epoch:0, Step:0, D-Loss:4.919, D-Acc:47.070, G-Loss:76.036

Epoch:0, Step:100, D-Loss:0.392, D-Acc:45.312, G-Loss:12.424

Epoch:0, Step:200, D-Loss:0.241, D-Acc:71.094, G-Loss:8.243

Epoch:0, Step:300, D-Loss:0.718, D-Acc:73.633, G-Loss:8.938

Epoch:0, Step:400, D-Loss:0.188, D-Acc:74.609, G-Loss:21.487

Epoch:0, Step:500, D-Loss:0.135, D-Acc:89.062, G-Loss:8.117

Epoch:0, Step:600, D-Loss:0.097, D-Acc:88.086, G-Loss:9.455

Epoch:0, Step:700, D-Loss:0.377, D-Acc:54.297, G-Loss:10.299

Epoch:0, Step:800, D-Loss:0.029, D-Acc:99.805, G-Loss:14.020

Epoch:0, Step:900, D-Loss:0.137, D-Acc:84.961, G-Loss:19.451

Epoch:0, Step:1000, D-Loss:0.131, D-Acc:83.789, G-Loss:8.633

Epoch:1, Step:0, D-Loss:0.510, D-Acc:52.734, G-Loss:13.977

.....

.....

.....

.....

Our Pix2Pix GAN network is quite big, it trains really slow. So, we have only trained it for very small number of steps. Let's now check out some results.

Step 9: Results

In this step, we will check some of the generated translation results from our Pix2Pix GAN model. We can utilize the '*show_generator_results*' function for checking the generator results on some random test images. See the following code:

```
for i in range(5):  
    show_generator_results(generator_network)
```

Output:

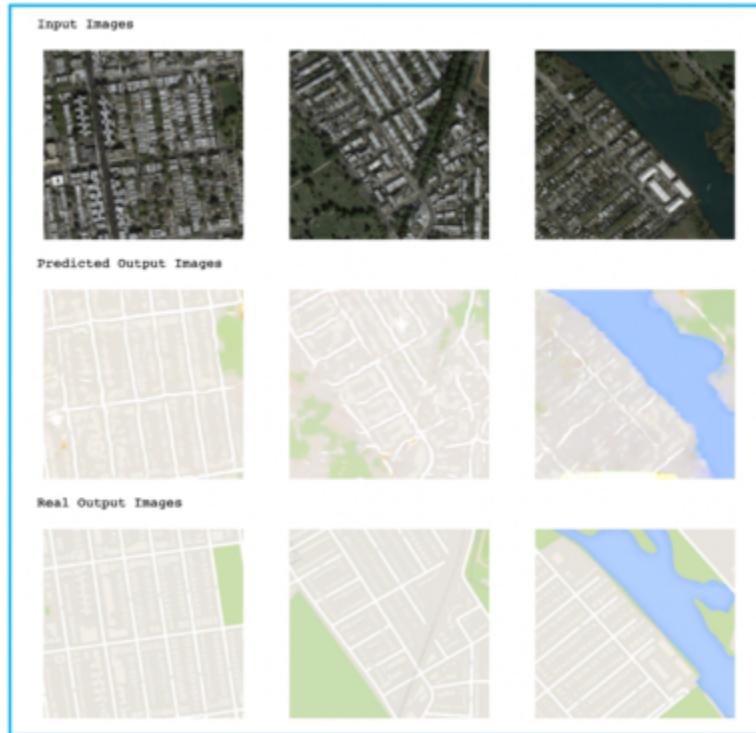


Figure 8.6: Pix2Pix GAN results on Maps dataset. (Top) Input source images. (Middle) Images translated by Pix2Pix generator, and (Bottom) Real translated version of the images.

Figure 8.6 shows the results of the Pix2Pix GAN generator. The top row represents the input domain images that we passed to the model. Middle row shows the generator outputs of our Pix2Pix GAN. The bottom row shows the real translation of the input images (from the second domain). We can clearly see that the generator network is able to convert the source domain images into the target domain. As it was a very large model, the training was really slow. So, we have only trained it for a very small number of steps. Still, we can see that the results are starting to look really good.

We have just seen the capabilities of the Pix2Pix GAN for the task of image-to-image translation. But this model, however, requires paired training data. Let's now experiment with Cycle GAN model, that works with unpaired data.

CYCLE GAN FOR APPLES TO ORANGES TRANSLATION

Objective

In this experiment, we will implement and train a Cycle GAN model and verify its results for unpaired image-to-image translation.

This experiment has the following steps:

- Importing Libraries
- Download and Unzip Data
- Check few Samples
- Define Generator Network
- Define Discriminator Network
- Define Cycle-GAN
- Define Utility Functions
- Training Cycle-GAN
- Results

Let's get started.

Step 1: Importing Libraries

Open a Jupyter Notebook with python kernel and import useful python packages in a cell. In this experiment, we will use ‘*numpy*’ for data manipulation, ‘*matplotlib*’ for plotting images and graphs, and *TensorFlow2* for implementing the model architecture later.

Checkout the following snippet for importing libraries:

```
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
from tqdm import tqdm_notebook  
%matplotlib inline  
import tensorflow  
print(tensorflow.__version__)
```

Output: 2.4.0

Let's get the data now.

Step 2: Download and Unzip Data

In this step, we will first download the Apple2Orange dataset from the URL given in following python snippet. Apple2Orange is publicly available dataset that was published by the authors of Cycle GAN paper.

The following snippet first extracts the dataset and then reads the file names of train and test partitions into four lists, two for apples and two for oranges:

```
!wget https://people.eecs.berkeley.edu/~taesung_park\  
/CycleGAN/datasets/apple2orange.zip  
!unzip apple2orange.zip  
!ls apple2orange  
import glob  
apples_train = glob.glob('apple2orange/trainA/*.jpg')  
oranges_train = glob.glob('apple2orange/trainB/*.jpg')  
apples_test = glob.glob('apple2orange/testA/*.jpg')  
oranges_test = glob.glob('apple2orange/testB/*.jpg')  
len(apples_train), len(oranges_train), \  
len(apples_test), len(oranges_test)
```

Overall, we have 995 training and 266 test images for apples, and 1019 training and 248 test images for oranges.

Output: (995, 1019, 266, 248)

Let's check out some data samples.

Step 3: Check few Samples

In this step, we will display a few samples from both the domains. The following python code first plots some random samples from the 'Apples' domain, and then plots some random samples from the 'Oranges' domain as well:

```
print ("Apples")
```

```
for k in range(5):
    plt.figure(figsize=(13, 13))
    for j in range(9):
        file = np.random.choice(apples_train)
        img = cv2.imread(file)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        plt.subplot(990 + 1 + j)
        plt.imshow(img)
        plt.axis('off')
    plt.show()
    print ("-*80)
    print ("Oranges")
for k in range(5):
    plt.figure(figsize=(13, 13))
    for j in range(9):
        file = np.random.choice(oranges_train)
        img = cv2.imread(file)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        plt.subplot(990 + 1 + j)
        plt.imshow(img)
        plt.axis('off')
    plt.show()
```

Figure 8.7 shows some samples from the Apple2Orange dataset. First two rows show the images of apples and last rows show the images of oranges.

Output:



Figure 8.7: Few samples from the Apple2Orange dataset

Let's now work on the model architecture.

Step 4: Define Generator Network

In this step, we will define the generator network architecture for our Cycle GAN setup. As suggested in the Cycle GAN paper, we will design a *UNet* like architecture for our generator network with skip connections. In our setup, the generator network accepts an image of size 256 x 256 x 3 as input (an image from the source domain), and passes it through multiple layers of strided *Conv2D* (for down-sampling), *LeakyReLU* activation with an alpha value of 0.2, and Instance Normalization. Then the resulting features are passed through a bottleneck layer that is a combination of a *Conv2D* layer and *ReLU* activation. The output of the bottleneck layer along with the outputs of previous encoder layers are passed through the decoder network. Each decoder block passes the input features through the layers of *Conv2DTranspose*, *ReLU* activation, Instance Normalization, and finally creates a *UNet* like skip connection using a concatenation layer (see the following python code). The final decoder layer, converts the decoded feature vector into an output image of size 256 x 256 x 3, and uses a '*tanh*' activation layer to restrict the pixel values of the translated image into a range of [-1, 1].

Using these design guidelines, we will define two exactly similar generator networks; one for each domain. The generator network AB, will convert the images from domain A to domain B. While, the generator network BA, will convert the images from domain B to domain A. The inherent architecture is exactly same.

The following python code defines the generators for our Cycle GAN setup as discussed:

```
import tensorflow_addons as tfa

def encoder_layer(input_layer, filters, bn=True):
    x = tensorflow.keras.layers.Conv2D(filters, kernel_size\ 
=(4,4), strides=(2,2), padding='same')(input_layer)
    x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
    if bn:
        x = tfa.layers.InstanceNormalization()(x)
    return x

def decoder_layer(input_layer, skip_input, filters):
    x = tensorflow.keras.layers.Conv2DTranspose(filters, \
kernel_size=(4,4), strides=(2,2), \
padding='same')(input_layer)
    x = tensorflow.keras.layers.Activation('relu')(x)
    x = tfa.layers.InstanceNormalization()(x)
    x = tensorflow.keras.layers.Concatenate()([x, skip_input])
    return x

source_image_A = tensorflow.keras.layers.Input(shape=(256, 256, 3))
source_image_B = tensorflow.keras.layers.Input(shape=(256, 256, 3))

def make_generator():
    source_image = tensorflow.keras.layers.Input(shape=(256, 256, 3))
    e1 = encoder_layer(source_image, 32, bn=False)
    e2 = encoder_layer(e1, 64)
```

```

e3 = encoder_layer(e2, 128)
e4 = encoder_layer(e3, 256)
e5 = encoder_layer(e4, 256)
e6 = encoder_layer(e5, 256)
e7 = encoder_layer(e6, 256)
bottle_neck = tensorflow.keras.layers.Conv2D(512,\n
(4,4), strides=(2,2), padding='same')(e7)
b = tensorflow.keras.layers.Activation('relu')(bottle_neck)
d1 = decoder_layer(b, e7, 256)
d2 = decoder_layer(d1, e6, 256)
d3 = decoder_layer(d2, e5, 256)
d4 = decoder_layer(d3, e4, 256)
d5 = decoder_layer(d4, e3, 256)
d6 = decoder_layer(d5, e2, 64)
d7 = decoder_layer(d6, e1, 32)
decoded = tensorflow.keras.layers.Conv2DTranspose(3,\n
kernel_size=(4,4), strides=(2,2), padding='same')(d7)
translated_image = tensorflow.keras.layers.Activation('tanh')(decoded)
return source_image, translated_image
source_image, translated_image = make_generator()
generator_network_AB = tensorflow.keras.models.Model(\n
inputs=source_image, outputs=translated_image)
source_image, translated_image = make_generator()
generator_network_BA = tensorflow.keras.models.Model(\n
inputs=source_image, outputs=translated_image)
print(generator_network_AB.summary())

```

Following is the summary of one of the generator networks. It roughly has 17M trainable parameters:

Output:

Model: "model"

Layer (type) Output Shape Param # Connected to

input_3 (InputLayer) [(None, 256, 256, 3) 0]

conv2d (Conv2D) (None, 128, 128, 32) 1568 input_3[0][0]

leaky_re_lu (LeakyReLU) (None, 128, 128, 32) 0 conv2d[0][0]

conv2d_1 (Conv2D) (None, 64, 64, 64) 32832 leaky_re_lu[0][0]

leaky_re_lu_1 (LeakyReLU) (None, 64, 64, 64) 0 conv2d_1[0][0]

instance_normalization (Instanc (None, 64, 64, 64) 128 leaky_re_lu_1[0][0]

conv2d_2 (Conv2D) (None, 32, 32, 128) 131200 instance_normalization[0][0]

leaky_re_lu_2 (LeakyReLU) (None, 32, 32, 128) 0 conv2d_2[0][0]

instance_normalization_1 (Insta (None, 32, 32, 128) 256 leaky_re_lu_2[0][0]

conv2d_3 (Conv2D) (None, 16, 16, 256) 524544 instance_normalization_1[0][0]

leaky_re_lu_3 (LeakyReLU) (None, 16, 16, 256) 0 conv2d_3[0][0]

instance_normalization_2 (Insta (None, 16, 16, 256) 512 leaky_re_lu_3[0][0]

conv2d_4 (Conv2D) (None, 8, 8, 256) 1048832 instance_normalization_2[0][0]

leaky_re_lu_4 (LeakyReLU) (None, 8, 8, 256) 0 conv2d_4[0][0]

instance_normalization_3 (Insta (None, 8, 8, 256) 512 leaky_re_lu_4[0][0]

conv2d_5 (Conv2D) (None, 4, 4, 256) 1048832 instance_normalization_3[0][0]

leaky_re_lu_5 (LeakyReLU) (None, 4, 4, 256) 0 conv2d_5[0][0]

instance_normalization_4 (Insta (None, 4, 4, 256) 512 leaky_re_lu_5[0][0]

conv2d_6 (Conv2D) (None, 2, 2, 256) 1048832 instance_normalization_4[0][0]

leaky_re_lu_6 (LeakyReLU) (None, 2, 2, 256) 0 conv2d_6[0][0]

instance_normalization_5 (Insta (None, 2, 2, 256) 512 leaky_re_lu_6[0][0]

conv2d_7 (Conv2D) (None, 1, 1, 512) 2097664 instance_normalization_5[0][0]

activation (Activation) (None, 1, 1, 512) 0 conv2d_7[0][0]

conv2d_transpose (Conv2DTranspo (None, 2, 2, 256) 2097408 activation[0][0]

activation_1 (Activation) (None, 2, 2, 256) 0 conv2d_transpose[0][0]

instance_normalization_6 (Insta (None, 2, 2, 256) 512 activation_1[0][0]

concatenate (Concatenate) (None, 2, 2, 512) 0 instance_normalization_6[0][0]

instance_normalization_5[0][0]

conv2d_transpose_1 (Conv2DTrans (None, 4, 4, 256) 2097408 concatenate[0][0]

activation_2 (Activation) (None, 4, 4, 256) 0 conv2d_transpose_1[0][0]

instance_normalization_7 (Insta (None, 4, 4, 256) 512 activation_2[0][0]

concatenate_1 (Concatenate) (None, 4, 4, 512) 0 instance_normalization_7[0][0]

instance_normalization_4[0][0]

conv2d_transpose_2 (Conv2DTrans (None, 8, 8, 256) 2097408 concatenate_1[0][0]

activation_3 (Activation) (None, 8, 8, 256) 0 conv2d_transpose_2[0][0]

instance_normalization_8 (Insta (None, 8, 8, 256) 512 activation_3[0][0]

concatenate_2 (Concatenate) (None, 8, 8, 512) 0 instance_normalization_8[0][0]

instance_normalization_3[0][0]

conv2d_transpose_3 (Conv2DTrans (None, 16, 16, 256) 2097408 concatenate_2[0][0]

activation_4 (Activation) (None, 16, 16, 256) 0 conv2d_transpose_3[0][0]

instance_normalization_9 (Insta (None, 16, 16, 256) 512 activation_4[0][0]

concatenate_3 (Concatenate) (None, 16, 16, 512) 0 instance_normalization_9[0][0]

instance_normalization_2[0][0]

conv2d_transpose_4 (Conv2DTrans (None, 32, 32, 256) 2097408 concatenate_3[0][0]

activation_5 (Activation) (None, 32, 32, 256) 0 conv2d_transpose_4[0][0]

instance_normalization_10 (Inst (None, 32, 32, 256) 512 activation_5[0][0]

concatenate_4 (Concatenate) (None, 32, 32, 384) 0 instance_normalization_10[0][0]

instance_normalization_1[0][0]

conv2d_transpose_5 (Conv2DTrans (None, 64, 64, 64) 393280 concatenate_4[0][0]

activation_6 (Activation) (None, 64, 64, 64) 0 conv2d_transpose_5[0][0]

instance_normalization_11 (Inst (None, 64, 64, 64) 128 activation_6[0][0]

concatenate_5 (Concatenate) (None, 64, 64, 128) 0 instance_normalization_11[0][0]

instance_normalization[0][0]

conv2d_transpose_6 (Conv2DTrans (None, 128, 128, 32) 65568 concatenate_5[0][0]

activation_7 (Activation) (None, 128, 128, 32) 0 conv2d_transpose_6[0][0]

instance_normalization_12 (Inst (None, 128, 128, 32) 64 activation_7[0][0]

concatenate_6 (Concatenate) (None, 128, 128, 64) 0 instance_normalization_12[0][0]

leaky_re_lu[0][0]

```
conv2d_transpose_7 (Conv2DTrans (None, 256, 256, 3) 3075 concatenate_6[0][0]
```

```
activation_8 (Activation) (None, 256, 256, 3) 0 conv2d_transpose_7[0][0]
```

Total params: 16,888,451

Trainable params: 16,888,451

Non-trainable params: 0

None

Let's define the discriminator network now.

Step 5: Define Discriminator Network

In this step, we will define the discriminator network for our Cycle GAN setup. In our setup, the discriminator network accepts an image of size 256 x 256 x 3 as input (the translated image). This input image is then passed through multiple layers of strided *Conv2D*, *LeakyReLU* activation with an alpha value of 0.2, and an Instance Normalization layer (except for the first layer). The final output of our discriminator network is a patch of size 8 x 8 x 1, as it follows a Patch GAN architecture (as discussed earlier in this skill).

In our Cycle GAN setup, we will require two exactly same discriminators; one for each domain. The discriminator network A, will verify the translated version of domain A, while the discriminator network B, will verify the translated version of domain B. As both the discriminators are Patch GAN based, we can compile them using '*mean squared error (mse)*' loss function.

The following python snippet defines the discriminators for our Cycle GAN setup and compiles them:

```
def my_conv_layer(input_layer, filters, bn=True):
    x = tensorflow.keras.layers.Conv2D(filters, kernel_size=\
(4,4), strides=(2,2), padding='same')(input_layer)
    x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
    if bn:
        x = tfa.layers.InstanceNormalization()(x)
    return x

def make_discriminator():
    target_image_input = tensorflow.keras.layers.Input(shape=(256, 256, 3))
    x = my_conv_layer(target_image_input, 64, bn=False)
    x = my_conv_layer(x, 128)
    x = my_conv_layer(x, 256)
    x = my_conv_layer(x, 512)
    x = my_conv_layer(x, 512)
    patch_features = tensorflow.keras.layers.Conv2D(1,\ 
kernel_size=(4,4), strides=(1,1), padding='same')(x)
    return target_image_input, patch_features

target_image_input, patch_features = make_discriminator()
discriminator_network_A = tensorflow.keras.models.Model(\ 
inputs=target_image_input, outputs=patch_features)
target_image_input, patch_features = make_discriminator()
discriminator_network_B = tensorflow.keras.models.Model(\ 
inputs=target_image_input, outputs=patch_features)
print (discriminator_network_A.summary())
```

```
adam_optimizer =  
tensorflow.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5)  
discriminator_network_A.compile(loss='mse', \  
optimizer=adam_optimizer, metrics=['accuracy'])  
discriminator_network_B.compile(loss='mse', \  
optimizer=adam_optimizer, metrics=['accuracy'])
```

Following is the summary of one of the discriminator networks:

Output:

Model: "model_2"

Layer (type) Output Shape Param #

input_5 (InputLayer) [(None, 256, 256, 3)] 0

conv2d_16 (Conv2D) (None, 128, 128, 64) 3136

leaky_re_lu_14 (LeakyReLU) (None, 128, 128, 64) 0

conv2d_17 (Conv2D) (None, 64, 64, 128) 131200

leaky_re_lu_15 (LeakyReLU) (None, 64, 64, 128) 0

instance_normalization_26 (I (None, 64, 64, 128) 256

conv2d_18 (Conv2D) (None, 32, 32, 256) 524544

leaky_re_lu_16 (LeakyReLU) (None, 32, 32, 256) 0

instance_normalization_27 (I (None, 32, 32, 256) 512

```
conv2d_19 (Conv2D) (None, 16, 16, 512) 2097664
```

```
leaky_re_lu_17 (LeakyReLU) (None, 16, 16, 512) 0
```

```
instance_normalization_28 (I (None, 16, 16, 512) 1024
```

```
conv2d_20 (Conv2D) (None, 8, 8, 512) 4194816
```

```
leaky_re_lu_18 (LeakyReLU) (None, 8, 8, 512) 0
```

```
instance_normalization_29 (I (None, 8, 8, 512) 1024
```

```
conv2d_21 (Conv2D) (None, 8, 8, 1) 8193
```

```
=====
```

```
Total params: 6,962,369
```

```
Trainable params: 6,962,369
```

```
Non-trainable params: 0
```

```
None
```

The discriminator networks are all set now. Let's define the final combined Cycle GAN.

Step 6: Define Cycle-GAN

In this step, we will define the final combined Cycle GAN architecture. The combined Cycle GAN model is a combination of four models: two generator networks and two discriminator networks. The generator network AB, translates images from domain A to domain B, and the discriminator B validates these translated images. Similarly, the generator BA, translates images from domain B to domain A, and the discriminator A validates these translated images. The discriminator networks are Patch GAN based, so they

basically validate if the given image is real or fake representation of the given domain.

In the combined model: both the discriminator networks are kept frozen as the idea of the combined model is only to update the weights of the generator networks using the frozen discriminators as a gradient supplier. Combined Cycle GAN model has 6 outputs: two validity outputs (one for each domain), two cycle consistency outputs (one for each domain), and two identity outputs (one for each domain). Validity outputs come from the discriminator networks indicating the real vs. fake feedback.

The cycle consistency output is generated using both the generators. The idea is simple: one source image from domain A is converted to domain B using the generator AB, then the converted image of domain B is passed into the generator BA and it converts it back to domain A. Now, as per the cycle consistency, this doubly converted image which now represents domain A should be exactly similar to the original source image that also belongs to domain A. In the similar manner, the cycle consistency is checked for domain B as well.

The idea behind identity output is also simple. We know that the generator AB, converts an image from domain A to domain B. But, if it is provided with an image that already belongs to domain B as input, it should output the same image. Similarly, the identity can be verified for the generator BA as well.

For these six outputs, the model needs 6 loss functions. It uses ‘*mse*’ loss for validity outputs and ‘*mae*’ loss for image comparison outputs. Validity loss, cycle consistency loss and identify loss have loss weights of 1, 10 and 3 respectively.

The following python code defines and compiles the final Cycle GAN setup:

```
# Domain Transfer
fake_B = generator_network_AB(source_image_A)
fake_A = generator_network_BA(source_image_B)
# Restoring original Domain
get_back_A = generator_network_BA(fake_B)
```

```

get_back_B = generator_network_AB(fake_A)
# Get back Identical/Same Image
get_same_A = generator_network_BA(source_image_A)
get_same_B = generator_network_AB(source_image_B)
discriminator_network_A.trainable=False
discriminator_network_B.trainable=False
# Tell Real vs Fake, for a given domain
verify_A = discriminator_network_A(fake_A)
verify_B = discriminator_network_B(fake_B)
cycle_gan = tensorflow.keras.models.Model(inputs =
[source_image_A, source_image_B], \
outputs = [verify_A, verify_B, \
get_back_A, get_back_B, get_same_A, get_same_B])
cycle_gan.summary()
# Compiling Model
cycle_gan.compile(loss=['mse', 'mse', 'mae', 'mae', 'mae', \
'mae'], loss_weights=[1, 1, 10, 10, 3, 3], \
optimizer=adam_optimizer)

```

Following is the summary of the combined Cycle GAN model:

Output:

Model: "model_4"

Layer (type) Output Shape Param # Connected to

```
=====
=====
```

input_2 (InputLayer) [(None, 256, 256, 3) 0

input_1 (InputLayer) [(None, 256, 256, 3) 0

model_1 (Functional) (None, 256, 256, 3) 16888451 input_2[0][0]

model[0][0]

input_1[0][0]

model (Functional) (None, 256, 256, 3) 16888451 input_1[0][0]

model_1[0][0]

input_2[0][0]

model_2 (Functional) (None, 8, 8, 1) 6962369 model_1[0][0]

model_3 (Functional) (None, 8, 8, 1) 6962369 model[0][0]

Total params: 47,701,640

Trainable params: 33,776,902

Non-trainable params: 13,924,738

Our final model is ready. Let's work on the training setup now.

Step 7: Define Utility Functions

In this step, we will define some utility functions that will help us in creating batches of the data for training. Specifically, we will define the utility functions for: generating the apple to orange results from the first generator, generating the orange to apple results from the second generator, generating a real data batch from apples, generating a real data batch from oranges, and two utility functions for plotting the results of both the generators.

The following python code defines the utility functions:

```
def apples_to_oranges(apples, generator_network):
    generated_samples = generator_network.predict_on_batch(apples)
    return generated_samples

def oranges_to_apples(oranges, generator_network):
    generated_samples = generator_network.predict_on_batch(oranges)
    return generated_samples

def get_apple_samples(batch_size):
    random_files = np.random.choice(apples_train, size=batch_size)
    images = []
    for file in random_files:
        img = cv2.imread(file)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        images.append((img-127.5)/127.5)
    apple_images = np.array(images)
    return apple_images

def get_orange_samples(batch_size):
    random_files = np.random.choice(oranges_train, size=batch_size)
    images = []
    for file in random_files:
```

```
img = cv2.imread(file)
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
images.append((img-127.5)/127.5)
orange_images = np.array(images)
return orange_images

def show_generator_results_apples_to_oranges(\n    generator_network_AB, generator_network_BA):\n    images = []\n    for j in range(7):\n        file = np.random.choice(apples_test)\n        img = cv2.imread(file)\n        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)\n        images.append(img)\n        print ('Input Apple Images')\n        plt.figure(figsize=(13, 13))\n        for j, img in enumerate(images):\n            plt.subplot(770 + 1 + j)\n            plt.imshow(img)\n            plt.axis('off')\n            #plt.title(trainY[i])\n            plt.show()\n        print ('Translated (Apple -> Orange) Images')\n        translated = []\n        plt.figure(figsize=(13, 13))\n        for j, img in enumerate(images):\n            img = (img-127.5)/127.5\n            output = apples_to_oranges(np.array([img]),\n                generator_network_AB)[0]
```

```
translated.append(output)
output = (output+1.0)/2.0
plt.subplot(770 + 1 + j)
plt.imshow(output)
plt.axis('off')
#plt.title(trainY[i])
plt.show()
print ('Translated reverse ( Fake Oranges -> Fake Apples)')
plt.figure(figsize=(13, 13))
for j, img in enumerate(translated):
    output = oranges_to_apples(np.array([img]),\
generator_network_BA)[0]
    output = (output+1.0)/2.0
    plt.subplot(770 + 1 + j)
    plt.imshow(output)
    plt.axis('off')
    #plt.title(trainY[i])
    plt.show()

def show_generator_results_oranges_to_apples(\
generator_network_AB, generator_network_BA):
images = []
for j in range(7):
    file = np.random.choice(oranges_test)
    img = cv2.imread(file)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    images.append(img)
    print ('Input Orange Images')
plt.figure(figsize=(13, 13))
```

```
for j, img in enumerate(images):
    plt.subplot(770 + 1 + j)
    plt.imshow(img)
    plt.axis('off')
    #plt.title(trainY[i])
    plt.show()
    print ('Translated (Orange -> Apple) Images')
    translated = []
    plt.figure(figsize=(13, 13))
    for j, img in enumerate(images):
        img = (img-127.5)/127.5
        output = oranges_to_apples(np.array([img]),\
generator_network_BA)[0]
        translated.append(output)
        output = (output+1.0)/2.0
        plt.subplot(770 + 1 + j)
        plt.imshow(output)
        plt.axis('off')
        #plt.title(trainY[i])
        plt.show()
    print ('Translated reverse (Fake Apples -> Fake Oranges)')
    plt.figure(figsize=(13, 13))
    for j, img in enumerate(translated):
        output = apples_to_oranges(np.array([img]),\
generator_network_AB)[0]
        output = (output+1.0)/2.0
        plt.subplot(770 + 1 + j)
        plt.imshow(output)
```

```
plt.axis('off')
#plt.title(trainY[i])
plt.show()
```

We are now all set to start training our Cycle GAN network.

Step 8: Training Cycle-GAN

In this step, we will write the training iteration loop for our Cycle GAN. In each training step, we will first update the weights of the discriminator networks using separate batches of the real translation outputs and the fake translation outputs of each domain. It is advised to train the discriminators on separate batches of real and fake samples. The output of each discriminator network is a patch of size $8 \times 8 \times 1$, and it is compared with a similar patch of similar size. The patch of the real images is filled with ones, while the patch of the fake images is filled with zeroes.

Then, we freeze the weights of the discriminator networks and update the weights of the generator networks in a single pass, by passing the batches of the input images, output images and patches of real images (filled with ones). Here, we pass the patches of real images to make the discriminators believe that they are the real translations of the input images and calculate the loss values.

The following python code defines the training setup for Cycle GAN:

```
epochs = 500
batch_size = 1
steps = 1000
for i in range(0, epochs):
    for j in range(steps):
        if j%100 == 0:
            show_generator_results_apples_to_oranges(
                generator_network_AB, generator_network_BA)
            print ("-*100")
```

```
show_generator_results_oranges_to_apples(\n    generator_network_AB, generator_network_BA)\n\n# A == Apples\n# B == Oranges\n\ndomain_A_images = get_apple_samples(batch_size)\ndomain_B_images = get_orange_samples(batch_size)\nfake_patch = np.zeros((batch_size, 8, 8, 1))\nreal_patch = np.ones((batch_size, 8, 8, 1))\nfake_B_images = generator_network_AB(domain_A_images)\nfake_A_images = generator_network_BA(domain_B_images)\n\n# Updating Discriminator A weights\ndiscriminator_network_A.trainable=True\ndiscriminator_network_B.trainable=False\n\nloss_d_real_A = discriminator_network_A.train_on_batch(\n    domain_A_images, real_patch)\nloss_d_fake_A = discriminator_network_A.train_on_batch(\n    fake_A_images, fake_patch)\nloss_d_A = np.add(loss_d_real_A, loss_d_fake_A)/2.0\n\n# Updating Discriminator B weights\ndiscriminator_network_B.trainable=True\ndiscriminator_network_A.trainable=False\n\nloss_d_real_B = discriminator_network_B.train_on_batch(\n    domain_B_images, real_patch)\nloss_d_fake_B = discriminator_network_B.train_on_batch(\n    fake_B_images, fake_patch)\nloss_d_B = np.add(loss_d_real_B, loss_d_fake_B)/2.0\n\n# Make the Discriminator believe that these are real\n#samples and calculate loss to train the generator
```

```

discriminator_network_A.trainable=False
discriminator_network_B.trainable=False
# Updating Generator weights
loss_g = cycle_gan.train_on_batch(
[domain_A_images, domain_B_images],\
[real_patch, real_patch, domain_A_images,\ 
domain_B_images, domain_A_images, domain_B_images])
if j%50 == 0:
    print ("Epoch:%.0f, Step:%.0f, DA-Loss:%.3f, \
DA-Acc:%.3f, DB-Loss:%.3f, \
DB-Acc:%.3f, G-Loss:%.3f"\ \
%(i,j,loss_d_A[0],loss_d_A[1]*100,loss_d_B[0],\
loss_d_B[1]*100,loss_g[0]))

```

Following are the training logs of our Cycle GAN:

Output:

Epoch:0, Step:0, DA-Loss:2.378, DA-Acc:53.906, DB-Loss:1.842, DB-Acc:48.438, G-Loss:131.370

Epoch:0, Step:50, DA-Loss:0.343, DA-Acc:53.906, DB-Loss:0.479, DB-Acc:44.531, G-Loss:11.231

Epoch:0, Step:100, DA-Loss:0.317, DA-Acc:62.500, DB-Loss:0.332, DB-Acc:64.062, G-Loss:15.323

Epoch:0, Step:150, DA-Loss:0.325, DA-Acc:51.562, DB-Loss:0.407, DB-Acc:66.406, G-Loss:9.195

.....

.....

.....

.....

Epoch:1, Step:300, DA-Loss:0.250, DA-Acc:56.250, DB-Loss:0.597, DB-Acc:48.438, G-Loss:3.750

Epoch:1, Step:350, DA-Loss:0.175, DA-Acc:79.688, DB-Loss:0.204, DB-Acc:73.438, G-Loss:4.371

As this is a very big model. We only train it for a few steps. Let's check out the results now.

Step 9: Results

Output:

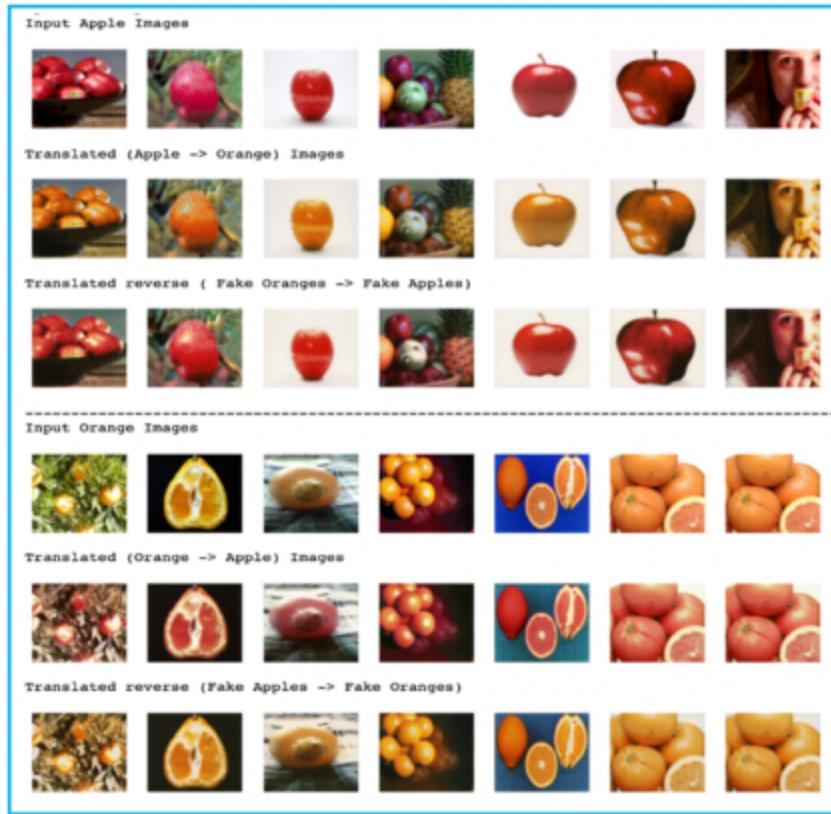


Figure 8.8: Cycle GAN results on apple2orange dataset. Top three rows represent the apple to orange translation, while the bottom three rows represent the orange to apple translation.

As shown in Figure 8.8, our Cycle GAN model is able to successfully translate apple images into oranges and vice versa. As it was a pretty big model and training was slow, we were only able to train it only for a few steps. Still, we can see that the results look really plausible. It understands that the apples are red and oranges are orange in nature. Best thing about

Cycle GAN is that it was able to solve image-to-image translation without needing the paired examples.

Next, let's train Cycle GAN on horses2zebras dataset as well.

CYCLE GAN FOR HORSES-TO-ZEBRAS EXPERIMENT

Objective

In this experiment, we will implement and train a Cycle GAN model for Horses to Zebras dataset.

This experiment has the following steps:

- Importing Libraries
- Download and Unzip Data
- Check few Samples
- Define Generator Network
- Define Discriminator Network
- Define Cycle-GAN
- Define Utility Functions
- Training Cycle-GAN
- Results

Let's get started.

Step 1: Importing Libraries

Open a Jupyter Notebook with python kernel and import useful python packages in a cell. In this experiment, we will use ‘*numpy*’ for data manipulation, ‘*matplotlib*’ for plotting images and graphs, and *TensorFlow2* for implementing the model architecture later.

Checkout the following snippet for importing libraries:

```
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
from tqdm import tqdm_notebook  
%matplotlib inline  
import tensorflow  
print(tensorflow.__version__)
```

Output: 2.4.0

Now, let's get the data.

Step 2: Download and Unzip Data

In this step, we will first download the Horse2Zebra dataset from the URL given in following python snippet. Horse2Zebra is publicly available dataset that was published by the authors of Cycle GAN paper.

The following snippet first extracts the dataset and then reads the file names of train and test partitions into four lists, two for horses and two for zebras:

```
!wget https://people.eecs.berkeley.edu/~\taesung_park/CycleGAN/datasets/horse2zebra.zip  
!unzip horse2zebra.zip  
!ls horse2zebra  
import glob  
path = ""  
horses_train = glob.glob(path + 'horse2zebra/trainA/*.jpg')  
zebras_train = glob.glob(path + 'horse2zebra/trainB/*.jpg')  
horses_test = glob.glob(path + 'horse2zebra/testA/*.jpg')  
zebras_test = glob.glob(path + 'horse2zebra/testB/*.jpg')  
len(horses_train), len(zebras_train),  
len(horses_test), len(zebras_test)
```

Overall, the dataset has 1067 training and 120 test images for horses, and 1334 training and 140 test images for zebras.

Output: (1067, 1334, 120, 140)

Let's check few samples from the dataset.

Step 3: Check few Samples

In this step, we will display a few samples from both the domains. The following python code first plots some random samples from the 'horses' domain, and then plots some random samples from the 'zebras' domain as well:

```

print ("Horses")
for k in range(2):
    plt.figure(figsize=(15, 15))
    for j in range(6):
        file = np.random.choice(horses_train)
        img = cv2.imread(file)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        plt.subplot(660 + 1 + j)
        plt.imshow(img)
        plt.axis('off')
        #plt.title(trainY[i])
    plt.show()
    print ("-*80")
print ("Zebras")
for k in range(2):
    plt.figure(figsize=(15, 15))
    for j in range(6):
        file = np.random.choice(zebras_train)
        img = cv2.imread(file)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        plt.subplot(660 + 1 + j)
        plt.imshow(img)
        plt.axis('off')
        #plt.title(trainY[i])
    plt.show()

```

Figure 8.9 shows some samples from the *horse2zebra* dataset. First two rows show the images of horses and last rows show the images of zebras.

Output:

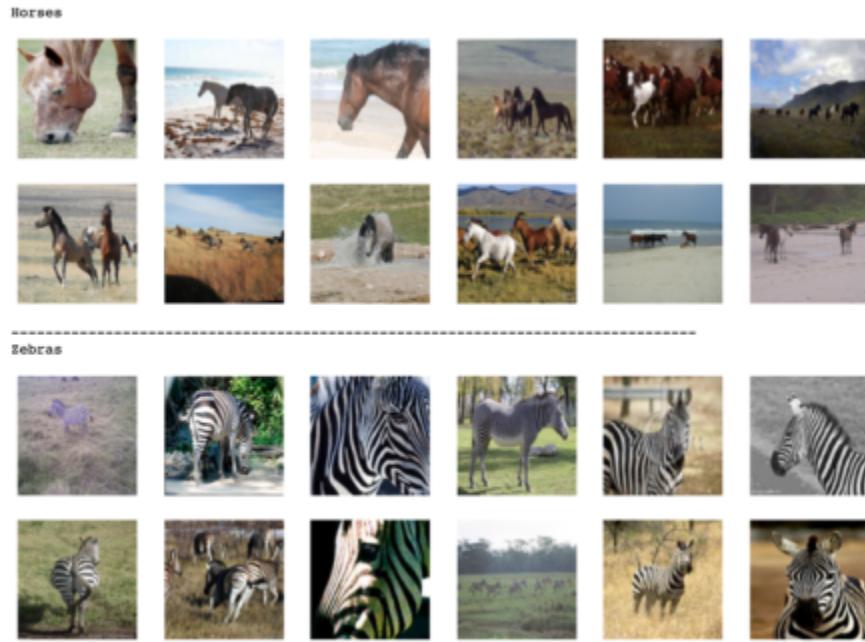


Figure 8.9: Few samples from horse2zebra dataset

Let's now define the model architecture.

Step 4: Define Generator Network

In this step, we will define the generator network architecture for our Cycle GAN setup. As suggested in the Cycle GAN paper, we will design a *ResNet* like architecture for our generator network with skip connections.

In our setup, the generator network accepts an image of size 256 x 256 x 3 as input (an image from the source domain), and passes it through multiple layers of strided *Conv2D* (for down-sampling), Instance Normalization and *ReLU* activation. Then the resulting features are passed through multiple *ResNet* like blocks. Each *ResNet* block passes the input features through the layers of *Reflection padding*, *Conv2D*, *ReLU* activation, Instance Normalization, and finally creates a skip connection using an addition layer (see the following python code). The final layer converts the decoded feature vector into an output image of size 256 x 256 x 3, and uses a '*tanh*' activation layer to restrict the pixel values of the translated image into a range of [-1, 1].

Using these design guidelines, we will define two exactly similar generator networks; one for each domain. The generator network AB, will

convert the images from domain A to domain B. While, the generator network BA, will convert the images from domain B to domain A. The inherent architecture is exactly same.

The following python code defines the generators for our Cycle GAN setup as discussed:

#Following function is taken from:

#<https://keras.io/examples/generative/cyclegan/>

class ReflectionPadding2D(tensorflow.keras.layers.Layer):

"""Implements Reflection Padding as a layer.

Args:

padding(tuple): Amount of padding for the spatial dimensions.

Returns:

A padded tensor with the same type as the input tensor.

"""

```
def __init__(self, padding=(1, 1), **kwargs):
    self.padding = tuple(padding)
    super(ReflectionPadding2D, self).__init__(**kwargs)
    def call(self, input_tensor, mask=None):
        padding_width, padding_height = self.padding
        padding_tensor = [
            [0, 0],
            [padding_height, padding_height],
            [padding_width, padding_width],
            [0, 0],
        ]
        return tensorflow.pad(input_tensor, \
            padding_tensor, mode="REFLECT")
import tensorflow_addons as tfa
```

```

# Weights initializer for the layers.
kernel_init = tensorflow.keras.initializers.RandomNormal(mean=0.0,
stddev=0.02)

# Gamma initializer for instance normalization.
gamma_init = tensorflow.keras.initializers.RandomNormal(mean=0.0,
stddev=0.02)

def custom_resnet_block(input_data, filters):
    x = ReflectionPadding2D()(input_data)
    x = tensorflow.keras.layers.Conv2D(filters, \
        kernel_size=(3,3), padding='valid',\
        kernel_initializer=kernel_init)(x)
    x = tfa.layers.InstanceNormalization()(x)
    x = tensorflow.keras.layers.Activation('relu')(x)
    x = ReflectionPadding2D()(x)
    x = tensorflow.keras.layers.Conv2D(filters, \
        kernel_size=(3,3), padding='valid',\
        kernel_initializer=kernel_init)(x)
    x = tfa.layers.InstanceNormalization()(x)
    x = tensorflow.keras.layers.Add()([x, input_data])
return x

def make_generator():
    source_image = tensorflow.keras.layers.Input(shape=(256, 256, 3))
    x = ReflectionPadding2D(padding=(3, 3))(source_image)
    x = tensorflow.keras.layers.Conv2D(64, \
        kernel_size=(7,7), kernel_initializer\
        =kernel_init, use_bias=False)(x)
    x = tfa.layers.InstanceNormalization()(x)
    x = tensorflow.keras.layers.Activation('relu')(x)
    x = tensorflow.keras.layers.Conv2D(128, \

```

```
kernel_size=(3,3), strides=(2,2), \
padding='same', kernel_initializer=kernel_init)(x)
x = tfa.layers.InstanceNormalization()(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.Conv2D(256, kernel_size=(3,3),\
strides=(2,2), padding='same', \
kernel_initializer=kernel_init)(x)
x = tfa.layers.InstanceNormalization()(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = custom_resnet_block(x, 256)
x = tensorflow.keras.layers.Conv2DTranspose(128, \
kernel_size=(3,3), strides=(2,2), \
padding='same', kernel_initializer=kernel_init)(x)
x = tfa.layers.InstanceNormalization()(x)
x = tensorflow.keras.layers.Activation('relu')(x)
x = tensorflow.keras.layers.Conv2DTranspose(64, \
kernel_size=(3,3), strides=(2,2), padding='same',\
kernel_initializer=kernel_init)(x)
x = tfa.layers.InstanceNormalization()(x)
x = tensorflow.keras.layers.Activation('relu')(x)
```

```
x = ReflectionPadding2D(padding=(3, 3))(x)
x = tensorflow.keras.layers.Conv2D(3, \
kernel_size=(7,7), padding='valid')(x)
x = tfa.layers.InstanceNormalization()(x)
translated_image = tensorflow.keras.layers.Activation('tanh')(x)
return source_image, translated_image
source_image, translated_image = make_generator()
generator_network_AB = tensorflow.keras.models.Model(\n
inputs=source_image, outputs=translated_image)
source_image, translated_image = make_generator()
generator_network_BA = tensorflow.keras.models.Model(\n
inputs=source_image, outputs=translated_image)
print(generator_network_AB.summary())
```

Following is the summary of one of the generator networks. It roughly has 11M trainable parameters:

Output:

Model: "model"

Layer (type) Output Shape Param # Connected to

input_1 (InputLayer) [(None, 256, 256, 3) 0

reflection_padding2d (Reflectio (None, 262, 262, 3) 0 input_1[0][0]

conv2d (Conv2D) (None, 256, 256, 64) 9408 reflection_padding2d[0][0]

instance_normalization (Instanc (None, 256, 256, 64) 128 conv2d[0][0]

activation (Activation) (None, 256, 256, 64) 0 instance_normalization[0][0]

conv2d_1 (Conv2D) (None, 128, 128, 128 73856 activation[0][0]

instance_normalization_1 (Insta (None, 128, 128, 128 256 conv2d_1[0][0]

activation_1 (Activation) (None, 128, 128, 128 0 instance_normalization_1[0][0]

conv2d_2 (Conv2D) (None, 64, 64, 256) 295168 activation_1[0][0]

instance_normalization_2 (Insta (None, 64, 64, 256) 512 conv2d_2[0][0]

activation_2 (Activation) (None, 64, 64, 256) 0 instance_normalization_2[0][0]

reflection_padding2d_1 (Reflect (None, 66, 66, 256) 0 activation_2[0][0]

conv2d_3 (Conv2D) (None, 64, 64, 256) 590080 reflection_padding2d_1[0][0]

instance_normalization_3 (Insta (None, 64, 64, 256) 512 conv2d_3[0][0]

activation_3 (Activation) (None, 64, 64, 256) 0 instance_normalization_3[0][0]

reflection_padding2d_2 (Reflect (None, 66, 66, 256) 0 activation_3[0][0]

conv2d_4 (Conv2D) (None, 64, 64, 256) 590080 reflection_padding2d_2[0][0]

instance_normalization_4 (Insta (None, 64, 64, 256) 512 conv2d_4[0][0]

add (Add) (None, 64, 64, 256) 0 instance_normalization_4[0][0]

activation_2[0][0]

reflection_padding2d_3 (Reflect (None, 66, 66, 256) 0 add[0][0]

conv2d_5 (Conv2D) (None, 64, 64, 256) 590080 reflection_padding2d_3[0][0]

instance_normalization_5 (Insta (None, 64, 64, 256) 512 conv2d_5[0][0]

activation_4 (Activation) (None, 64, 64, 256) 0 instance_normalization_5[0][0]

reflection_padding2d_4 (Reflect (None, 66, 66, 256) 0 activation_4[0][0]

conv2d_6 (Conv2D) (None, 64, 64, 256) 590080 reflection_padding2d_4[0][0]

instance_normalization_6 (Insta (None, 64, 64, 256) 512 conv2d_6[0][0]

add_1 (Add) (None, 64, 64, 256) 0 instance_normalization_6[0][0]

add[0][0]

reflection_padding2d_5 (Reflect (None, 66, 66, 256) 0 add_1[0][0]

conv2d_7 (Conv2D) (None, 64, 64, 256) 590080 reflection_padding2d_5[0][0]

instance_normalization_7 (Insta (None, 64, 64, 256) 512 conv2d_7[0][0]

activation_5 (Activation) (None, 64, 64, 256) 0 instance_normalization_7[0][0]

reflection_padding2d_6 (Reflect (None, 66, 66, 256) 0 activation_5[0][0]

conv2d_8 (Conv2D) (None, 64, 64, 256) 590080 reflection_padding2d_6[0][0]

instance_normalization_8 (Insta (None, 64, 64, 256) 512 conv2d_8[0][0]

add_2 (Add) (None, 64, 64, 256) 0 instance_normalization_8[0][0]

add_1[0][0]

reflection_padding2d_7 (Reflect (None, 66, 66, 256) 0 add_2[0][0]

conv2d_9 (Conv2D) (None, 64, 64, 256) 590080 reflection_padding2d_7[0][0]

instance_normalization_9 (Insta (None, 64, 64, 256) 512 conv2d_9[0][0]

activation_6 (Activation) (None, 64, 64, 256) 0 instance_normalization_9[0][0]

reflection_padding2d_8 (Reflect (None, 66, 66, 256) 0 activation_6[0][0]

conv2d_10 (Conv2D) (None, 64, 64, 256) 590080 reflection_padding2d_8[0][0]

instance_normalization_10 (Inst (None, 64, 64, 256) 512 conv2d_10[0][0]

add_3 (Add) (None, 64, 64, 256) 0 instance_normalization_10[0][0]

add_2[0][0]

reflection_padding2d_9 (Reflect (None, 66, 66, 256) 0 add_3[0][0]

conv2d_11 (Conv2D) (None, 64, 64, 256) 590080 reflection_padding2d_9[0][0]

instance_normalization_11 (Inst (None, 64, 64, 256) 512 conv2d_11[0][0]

activation_7 (Activation) (None, 64, 64, 256) 0 instance_normalization_11[0][0]

reflection_padding2d_10 (Reflec (None, 66, 66, 256) 0 activation_7[0][0]

conv2d_12 (Conv2D) (None, 64, 64, 256) 590080 reflection_padding2d_10[0][0]

instance_normalization_12 (Inst (None, 64, 64, 256) 512 conv2d_12[0][0]

add_4 (Add) (None, 64, 64, 256) 0 instance_normalization_12[0][0]

add_3[0][0]

reflection_padding2d_11 (Reflec (None, 66, 66, 256) 0 add_4[0][0]

conv2d_13 (Conv2D) (None, 64, 64, 256) 590080 reflection_padding2d_11[0][0]

instance_normalization_13 (Inst (None, 64, 64, 256) 512 conv2d_13[0][0]

activation_8 (Activation) (None, 64, 64, 256) 0 instance_normalization_13[0][0]

reflection_padding2d_12 (Reflec (None, 66, 66, 256) 0 activation_8[0][0]

conv2d_14 (Conv2D) (None, 64, 64, 256) 590080 reflection_padding2d_12[0][0]

instance_normalization_14 (Inst (None, 64, 64, 256) 512 conv2d_14[0][0]

add_5 (Add) (None, 64, 64, 256) 0 instance_normalization_14[0][0]

add_4[0][0]

reflection_padding2d_13 (Reflec (None, 66, 66, 256) 0 add_5[0][0]

conv2d_15 (Conv2D) (None, 64, 64, 256) 590080 reflection_padding2d_13[0][0]

instance_normalization_15 (Inst (None, 64, 64, 256) 512 conv2d_15[0][0]

activation_9 (Activation) (None, 64, 64, 256) 0 instance_normalization_15[0][0]

reflection_padding2d_14 (Reflec (None, 66, 66, 256) 0 activation_9[0][0]

conv2d_16 (Conv2D) (None, 64, 64, 256) 590080 reflection_padding2d_14[0][0]

instance_normalization_16 (Inst (None, 64, 64, 256) 512 conv2d_16[0][0]

add_6 (Add) (None, 64, 64, 256) 0 instance_normalization_16[0][0]

add_5[0][0]

reflection_padding2d_15 (Reflec (None, 66, 66, 256) 0 add_6[0][0]

conv2d_17 (Conv2D) (None, 64, 64, 256) 590080 reflection_padding2d_15[0][0]

instance_normalization_17 (Inst (None, 64, 64, 256) 512 conv2d_17[0][0]

activation_10 (Activation) (None, 64, 64, 256) 0 instance_normalization_17[0][0]

reflection_padding2d_16 (Reflec (None, 66, 66, 256) 0 activation_10[0][0]

conv2d_18 (Conv2D) (None, 64, 64, 256) 590080 reflection_padding2d_16[0][0]

instance_normalization_18 (Inst (None, 64, 64, 256) 512 conv2d_18[0][0]

add_7 (Add) (None, 64, 64, 256) 0 instance_normalization_18[0][0]

add_6[0][0]

reflection_padding2d_17 (Reflec (None, 66, 66, 256) 0 add_7[0][0]

conv2d_19 (Conv2D) (None, 64, 64, 256) 590080 reflection_padding2d_17[0][0]

instance_normalization_19 (Inst (None, 64, 64, 256) 512 conv2d_19[0][0]

activation_11 (Activation) (None, 64, 64, 256) 0 instance_normalization_19[0][0]

reflection_padding2d_18 (Reflec (None, 66, 66, 256) 0 activation_11[0][0]

conv2d_20 (Conv2D) (None, 64, 64, 256) 590080 reflection_padding2d_18[0][0]

instance_normalization_20 (Inst (None, 64, 64, 256) 512 conv2d_20[0][0]

add_8 (Add) (None, 64, 64, 256) 0 instance_normalization_20[0][0]

add_7[0][0]

conv2d_transpose (Conv2DTranspo (None, 128, 128, 128 295040 add_8[0][0]

instance_normalization_21 (Inst (None, 128, 128, 128 256 conv2d_transpose[0][0]

activation_12 (Activation) (None, 128, 128, 128 0 instance_normalization_21[0][0]

conv2d_transpose_1 (Conv2DTrans (None, 256, 256, 64) 73792 activation_12[0][0]

instance_normalization_22 (Inst (None, 256, 256, 64) 128 conv2d_transpose_1[0][0]

activation_13 (Activation) (None, 256, 256, 64) 0 instance_normalization_22[0][0]

reflection_padding2d_19 (Reflec (None, 262, 262, 64) 0 activation_13[0][0]

conv2d_21 (Conv2D) (None, 256, 256, 3) 9411 reflection_padding2d_19[0][0]

instance_normalization_23 (Inst (None, 256, 256, 3) 6 conv2d_21[0][0]

activation_14 (Activation) (None, 256, 256, 3) 0 instance_normalization_23[0][0]

```
=====
=====
Total params: 11,388,617
```

```
Trainable params: 11,388,617
```

```
Non-trainable params: 0
```

```
None
```

Let's now define the discriminator network.

Step 5: Define Discriminator Network

In this step, we will define the discriminator network for our Cycle GAN setup. In our setup, the discriminator network accepts an image of size 256 x 256 x 3 as input (the translated image). This input image is then passed through multiple layers of strided *Conv2D*, *LeakyReLU* activation with an alpha value of 0.2, and an Instance Normalization layer (except for the first layer). The final output of our discriminator network is a patch of size 32 x 32 x 1, as it follows a Patch GAN architecture (as discussed earlier in this skill).

In our Cycle GAN setup, we will require two exactly same discriminators; one for each domain. The discriminator network A, will verify the translated version of domain A, while the discriminator network B, will verify the translated version of domain B. As both the discriminators are Patch GAN based, we can compile them using '*mean squared error (mse)*' loss function.

The following python snippet defines the discriminators for our Cycle GAN setup and compiles them:

```
def my_conv_layer(input_layer, filters, strides, bn=True):
    x = tensorflow.keras.layers.Conv2D(filters, \
        kernel_size=(4,4), strides=strides, \
        padding='same', kernel_initializer=\\
```

```
kernel_init)(input_layer)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
if bn:
    x = tfa.layers.InstanceNormalization()(x)
return x

def make_discriminator():
    target_image_input = tensorflow.keras.layers.Input(shape=(256, 256, 3))
    x = my_conv_layer(target_image_input, 64, (2,2), bn=False)
    x = my_conv_layer(x, 128, (2,2))
    x = my_conv_layer(x, 256, (2,2))
    x = my_conv_layer(x, 512, (1,1))
    patch_features = tensorflow.keras.layers.Conv2D(1,\n        kernel_size=(4,4), padding='same')(x)
    return target_image_input, patch_features
```

```
target_image_input, patch_features = make_discriminator()
discriminator_network_A = tensorflow.keras.models.Model(\n    inputs=target_image_input, outputs=patch_features)
target_image_input, patch_features = make_discriminator()
discriminator_network_B = tensorflow.keras.models.Model(\n    inputs=target_image_input, outputs=patch_features)
print (discriminator_network_A.summary())
adam_optimizer =
tensorflow.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5)
discriminator_network_A.compile(loss='mse', \
optimizer=adam_optimizer, metrics=['accuracy'])
discriminator_network_B.compile(loss='mse', \
optimizer=adam_optimizer, metrics=['accuracy'])
```

Following is the summary of one of the discriminator networks:

Output:

Model: "model_2"

Layer (type) Output Shape Param #

input_3 (InputLayer) [(None, 256, 256, 3)] 0

conv2d_44 (Conv2D) (None, 128, 128, 64) 3136

leaky_re_lu (LeakyReLU) (None, 128, 128, 64) 0

conv2d_45 (Conv2D) (None, 64, 64, 128) 131200

leaky_re_lu_1 (LeakyReLU) (None, 64, 64, 128) 0

instance_normalization_48 (I (None, 64, 64, 128) 256

conv2d_46 (Conv2D) (None, 32, 32, 256) 524544

leaky_re_lu_2 (LeakyReLU) (None, 32, 32, 256) 0

instance_normalization_49 (I (None, 32, 32, 256) 512

conv2d_47 (Conv2D) (None, 32, 32, 512) 2097664

leaky_re_lu_3 (LeakyReLU) (None, 32, 32, 512) 0

instance_normalization_50 (I (None, 32, 32, 512) 1024

```
conv2d_48 (Conv2D) (None, 32, 32, 1) 8193
```

```
Total params: 2,766,529
```

```
Trainable params: 2,766,529
```

```
Non-trainable params: 0
```

None

The discriminators are ready now. Let's define the final Cycle GAN.

Step 6: Define Cycle-GAN

In this step, we will define the final combined Cycle GAN architecture. The combined Cycle GAN model is a combination of four models: two generator networks and two discriminator networks.

The generator network AB, translates images from domain A to domain B, and the discriminator B validates these translated images. Similarly, the generator BA, translates images from domain B to domain A, and the discriminator A validates the translated images. The discriminator networks are Patch GAN based, so they basically validate if the given image is real or fake representation of the given domain.

In the combined model: both discriminator networks are kept frozen as the idea of the combined model is to only update the weights of the generator networks using the frozen discriminators as a gradient supplier. The combined Cycle GAN model has 6 outputs: two validity outputs (one for each domain), two cycle consistency outputs (one for each domain), and two identity outputs (one for each domain). Validity outputs come from the discriminator networks, indicating the real vs. fake feedback.

The cycle consistency output is generated using both the generators. The idea is simple: one source image from domain A is converted to domain B using the generator AB, then the converted image of domain B is passed into the generator BA and it converts it back to domain A. Now, as per the cycle consistency, this doubly converted image which now represents domain A should be exactly similar to the original source image that also belongs to

domain A. In the similar manner, the cycle consistency is checked for domain B as well.

The idea behind identity output is also simple. We know that the generator AB, converts an image from domain A to domain B. But, if it is provided with an image that already belongs to domain B as input, it should output the same image. Similarly, the identity can be verified for the generator BA as well.

For these six outputs, the model needs 6 loss functions. It uses ‘mse’ loss for validity outputs and ‘mae’ loss for image comparison outputs. Validity loss, cycle consistency loss and identify loss have loss weights of 1, 10 and 5 respectively.

The following python code defines and compiles the final Cycle GAN setup:

```
source_image_A = tensorflow.keras.layers.Input(shape=(256, 256, 3))
source_image_B = tensorflow.keras.layers.Input(shape=(256, 256, 3))

# Domain Transfer
fake_B = generator_network_AB(source_image_A)
fake_A = generator_network_BA(source_image_B)

# Restoring original Domain
get_back_A = generator_network_BA(fake_B)
get_back_B = generator_network_AB(fake_A)

# Get back Identical/Same Image
get_same_A = generator_network_BA(source_image_A)
get_same_B = generator_network_AB(source_image_B)

discriminator_network_A.trainable=False
discriminator_network_B.trainable=False

# Tell Real vs Fake, for a given domain
verify_A = discriminator_network_A(fake_A)
verify_B = discriminator_network_B(fake_B)

cycle_gan = tensorflow.keras.models.Model(\
```

```
inputs = [source_image_A, source_image_B], \
outputs = [verify_A, verify_B, get_back_A,\ 
get_back_B, get_same_A, get_same_B])
cycle_gan.summary()
# Compiling Model
cycle_gan.compile(loss=['mse', 'mse', 'mae', 'mae', \
'mae', 'mae'], loss_weights=[1, 1, 10, \
10, 5, 5], optimizer=adam_optimizer)
```

Following is the summary of the final Cycle GAN model:

Output:

Model: "model_4"

Layer (type) Output Shape Param # Connected to

input_6 (InputLayer) [(None, 256, 256, 3) 0

input_5 (InputLayer) [(None, 256, 256, 3) 0

model_1 (Functional) (None, 256, 256, 3) 11388617 input_6[0][0]

model[0][0]

input_5[0][0]

```
model (Functional) (None, 256, 256, 3) 11388617 input_5[0][0]  
model_1[0][0]  
input_6[0][0]
```

```
model_2 (Functional) (None, 32, 32, 1) 2766529 model_1[0][0]
```

```
model_3 (Functional) (None, 32, 32, 1) 2766529 model[0][0]
```

```
=====
```

Total params: 28,310,292

```
Trainable params: 22,777,234
```

```
Non-trainable params: 5,533,058
```

Our Cycle GAN is now ready. Let's work on the training setup now.

Step 7: Define Utility Functions

In this step, we will define some utility functions that will help us in creating batches of the data for training. Specifically, we will define the utility functions for: generating the horse to zebra results from the first generator, generating the zebra to horse results from the second generator, generating a real data batch from horses, generating a real data batch from zebras, and two utility functions for plotting the results of both the generators.

The following python code defines the utility functions:

```
def horses_to_zebras(horses, generator_network):  
    generated_samples = generator_network.predict_on_batch(horses)
```

```
return generated_samples

def zebras_to_horses(zebras, generator_network):
    generated_samples = generator_network.predict_on_batch(zebras)
    return generated_samples

def get_horse_samples(batch_size):
    random_files = np.random.choice(horses_train, size=batch_size)
    images = []
    for file in random_files:
        img = cv2.imread(file)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        images.append((img-127.5)/127.5)
    horse_images = np.array(images)
    return horse_images

def get_zebra_samples(batch_size):
    random_files = np.random.choice(zebras_train, size=batch_size)
    images = []
    for file in random_files:
        img = cv2.imread(file)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        images.append((img-127.5)/127.5)
    zebra_images = np.array(images)
    return zebra_images

def show_generator_results_horses_to_zebras(\
generator_network_AB, generator_network_BA):
    images = []
    for j in range(5):
        file = np.random.choice(horses_test)
        img = cv2.imread(file)
```

```
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
images.append(img)
print ('Input Horse Images')
plt.figure(figsize=(13, 13))
for j, img in enumerate(images):
    plt.subplot(550 + 1 + j)
    plt.imshow(img)
    plt.axis('off')
    #plt.title(trainY[i])
    plt.show()
print ('Translated (Horse -> Zebra) Images')
translated = []
plt.figure(figsize=(13, 13))
for j, img in enumerate(images):
    img = (img-127.5)/127.5
    output = horses_to_zebras(\n        np.array([img]), generator_network_AB)[0]
    translated.append(output)
    output = (output+1.0)/2.0
    plt.subplot(550 + 1 + j)
    plt.imshow(output)
    plt.axis('off')
    #plt.title(trainY[i])
    plt.show()
print ('Translated reverse ( Fake Zebras -> Fake Horses)')
plt.figure(figsize=(13, 13))
for j, img in enumerate(translated):
    output = zebras_to_horses(\n        np.array([img]), generator_network_BA)[0]
    plt.subplot(550 + 1 + j)
    plt.imshow(output)
    plt.axis('off')
    #plt.title(trainY[i])
    plt.show()
```

```
np.array([img]), generator_network_BA)[0]
output = (output+1.0)/2.0
plt.subplot(550 + 1 + j)
plt.imshow(output)
plt.axis('off')
#plt.title(trainY[i])
plt.show()

def show_generator_results_zebras_to_horses(\n    generator_network_AB, generator_network_BA):\n    images = []\n    for j in range(5):\n        file = np.random.choice(zebras_test)\n        img = cv2.imread(file)\n        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)\n        images.append(img)\n        print ('Input Zebra Images')\n        plt.figure(figsize=(13, 13))\n        for j, img in enumerate(images):\n            plt.subplot(550 + 1 + j)\n            plt.imshow(img)\n            plt.axis('off')\n            #plt.title(trainY[i])\n            plt.show()\n        print ('Translated (Zebra -> Horse) Images')\n        translated = []\n        plt.figure(figsize=(13, 13))\n        for j, img in enumerate(images):\n            img = (img-127.5)/127.5
```

```

output = zebras_to_horses(\n
    np.array([img]), generator_network_BA)[0]\n
translated.append(output)\n
output = (output+1.0)/2.0\n
plt.subplot(550 + 1 + j)\n
plt.imshow(output)\n
plt.axis('off')\n
#plt.title(trainY[i])\n
plt.show()\n
print ('Translated reverse (Fake Horse -> Fake Zebra)')\n
plt.figure(figsize=(13, 13))\n
for j, img in enumerate(translated):\n
    output = horses_to_zebras(\n
        np.array([img]), generator_network_AB)[0]\n
    output = (output+1.0)/2.0\n
    plt.subplot(550 + 1 + j)\n
    plt.imshow(output)\n
    plt.axis('off')\n
    #plt.title(trainY[i])\n
    plt.show()

```

We are now all set to start training our Cycle GAN model.

Step 8: Training Cycle-GAN

In this step, we will write the training iteration loop for our Cycle GAN. In each training step, we will first update the weights of the discriminator networks using separate batches of the real translation outputs and the fake translation outputs of each domain. It is advised to train the discriminators on separate batches of real and fake samples. The output of each discriminator network is a patch of size 32 x 32 x 1, and it is compared with a similar patch of similar size. The patch of the real images is filled with ones, while the patch of the fake images is filled with zeroes.

Then, we freeze the weights of the discriminator networks and update the weights of both generator networks in a single pass, by passing the batches of the input images, output images and patches of real images (filled with ones). Here, we pass the patches of real images to make the discriminators believe that they are the real translations of the input images and calculate the loss values.

The following python code defines the training setup for Cycle GAN:

```
epochs = 500
batch_size = 1
steps = 1067
for i in range(0, epochs):
    if i%5 == 0:
        show_generator_results_horses_to_zebras(\n            generator_network_AB, generator_network_BA)
        print ("-*"*100)
        show_generator_results_zebras_to_horses(\n            generator_network_AB, generator_network_BA)
    for j in range(steps):
        # A == Horses
        # B == Zebras
        domain_A_images = get_horse_samples(batch_size)
        domain_B_images = get_zebra_samples(batch_size)
        fake_patch = np.zeros((batch_size, 32, 32, 1))
        real_patch = np.ones((batch_size, 32, 32, 1))
        fake_B_images = generator_network_AB(domain_A_images)
        fake_A_images = generator_network_BA(domain_B_images)
        # Updating Discriminator A weights
        discriminator_network_A.trainable=True
        discriminator_network_B.trainable=False
```

```

loss_d_real_A = discriminator_network_A.train_on_batch(\n
domain_A_images, real_patch)\n
loss_d_fake_A = discriminator_network_A.train_on_batch(\n
fake_A_images, fake_patch)\n
loss_d_A = np.add(loss_d_real_A, loss_d_fake_A)/2.0\n
# Updating Discriminator B weights\n
discriminator_network_B.trainable=True\n
discriminator_network_A.trainable=False\n
loss_d_real_B = discriminator_network_B.train_on_batch(\n
domain_B_images, real_patch)\n
loss_d_fake_B = discriminator_network_B.train_on_batch(\n
fake_B_images, fake_patch)\n
loss_d_B = np.add(loss_d_real_B, loss_d_fake_B)/2.0\n
# Make the Discriminator believe that these are real\n
#samples and calculate loss to train the generator\n
discriminator_network_A.trainable=False\n
discriminator_network_B.trainable=False\n
# Updating Generator weights\n
loss_g = cycle_gan.train_on_batch(\n
[domain_A_images, domain_B_images],\n
[real_patch, real_patch, domain_A_images,\n
domain_B_images, domain_A_images, domain_B_images])\n
if j%100 == 0:\n
    print ("Epoch:%.0f, Step:%.0f, DA-Loss:%.3f, \\\n
DA-Acc:%.3f, DB-Loss:%.3f, DB-Acc:%.3f, G-Loss:%.3f"\\\n
%(i,j,loss_d_A[0],loss_d_A[1]*100,loss_d_B[0],\\
loss_d_B[1]*100,loss_g[0]))\n

```

Following are the training logs of our Cycle GAN model:

Output:

Epoch:0, Step:0, DA-Loss:3.535, DA-Acc:49.951, DB-Loss:2.649, DB-Acc:47.412, G-Loss:36.135

Epoch:0, Step:100, DA-Loss:0.354, DA-Acc:40.967, DB-Loss:0.382, DB-Acc:48.193, G-Loss:7.942

Epoch:0, Step:200, DA-Loss:0.210, DA-Acc:74.365, DB-Loss:0.254, DB-Acc:60.791, G-Loss:12.729

Epoch:0, Step:300, DA-Loss:0.269, DA-Acc:65.039, DB-Loss:0.274, DB-Acc:61.084, G-Loss:9.904

Epoch:0, Step:400, DA-Loss:0.296, DA-Acc:56.006, DB-Loss:0.266, DB-Acc:51.953, G-Loss:7.609

Epoch:0, Step:500, DA-Loss:0.252, DA-Acc:49.854, DB-Loss:0.311, DB-Acc:54.297, G-Loss:10.891

Epoch:0, Step:600, DA-Loss:0.279, DA-Acc:46.924, DB-Loss:0.331, DB-Acc:45.605, G-Loss:5.328

Epoch:0, Step:700, DA-Loss:0.280, DA-Acc:53.125, DB-Loss:0.346, DB-Acc:41.455, G-Loss:10.363

Epoch:0, Step:800, DA-Loss:0.312, DA-Acc:51.758, DB-Loss:0.244, DB-Acc:60.156, G-Loss:8.232

Epoch:0, Step:900, DA-Loss:0.330, DA-Acc:53.809, DB-Loss:0.282, DB-Acc:53.320, G-Loss:8.059

Epoch:0, Step:1000, DA-Loss:0.485, DA-Acc:37.891, DB-Loss:0.352, DB-Acc:58.838, G-Loss:6.233

Epoch:1, Step:0, DA-Loss:0.212, DA-Acc:78.223, DB-Loss:0.242, DB-Acc:64.795, G-Loss:7.969

.....

.....

.....

.....

Let's now check the results.

Step 9: Results

As shown in Figure 8.10, our Cycle GAN model is able to successfully translate horse images into zebra images and vice versa. As it was a pretty big model and training was slow, we were only able to train it for a few steps. Still, we can see that the results look really plausible. Best thing about Cycle

GAN is that it was able to solve image-to-image translation problem without needing the paired examples.

Output:

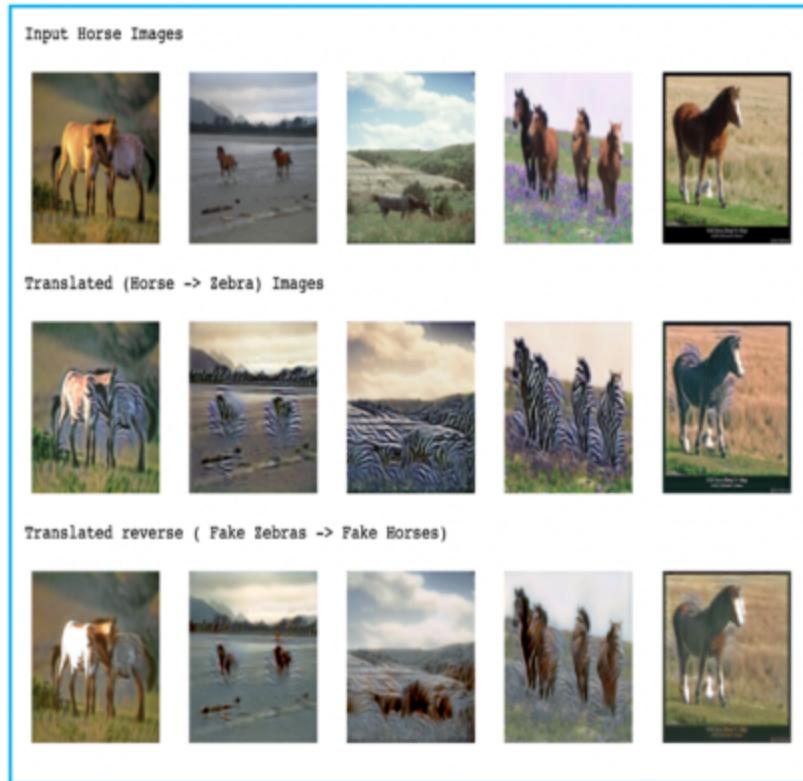


Figure 8.10: Cycle GAN results on horse2zebra dataset. First row represents the original images of horses. Second row represents the translated images into zebras. Third row represents the results of translating those fake zebras back to horse images.

We can see that Cycle GAN is a very powerful model that can learn image-to-image translation without needing the paired examples.

1. Conclusion

In this skill, we discussed the use case of image-to-image translation, in computer vision. We discovered that GANs are quite effective for the task of paired as well as unpaired image-to-image translation. Specifically, we learned about two GAN architectures capable of solving the image-to-image translation tasks: Pix2Pix GAN and Cycle GAN.

Pix2Pix GANs require paired images for solving image translation problems and have simple architecture. Cycle GANs on the other hand are capable of solving image translation in an unsupervised manner. But this super power comes at a cost. As a result Cycle GANs are very heavy models and require more compute resources and time for training. We understood how cycle consistency loss lets Cycle GAN perform unpaired image-to-image translation. We also found out that *Patch GAN* based discriminator network helps the GAN in generating high quality images.

Next, we will learn about some other common GAN based architectures.

End of Skill-08

Skill 9

Other GANs and Experiments

Generative Adversarial Networks have been successfully applied for the task of plausible image generation, image-to-image translation and many other such applications. The capabilities of the framework have been unveiled and the applications are limited to the researcher's creative mind. GANs have been successfully applied for many image-related tasks such as image super resolution, image inpainting, image style conversion and so on.

In this skill, we will learn about four popular variants of GANs that have been successfully applied in many creative image related applications. Specifically, we will learn about the following four GAN variants:

- Super Resolution GAN or SRGAN
- Disco-GAN
- Cartoon-GAN
- Context Encoder
- Conclusion

Let's get started.

1. Super Resolution GAN or SRGAN

Image super-resolution is a task of generating high resolution images from given low resolution images. Super Resolution GAN, or SRGAN for short, is a GAN based architecture for solving image super-resolution task.

SRGAN was proposed by *Christian Ledig, et al.* in 2017 in their paper titled "*Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network*".

SRGAN is capable of inferring photo-realistic natural images for $4\times$ upscaling factors. It uses a perceptual loss function which consists of an adversarial loss and a content loss.

The adversarial loss helps the model to learn to generate natural image manifold while, the content loss uses features maps to make the model invariant to the changes in the pixel space (basically, upscaling of images without changing the content).

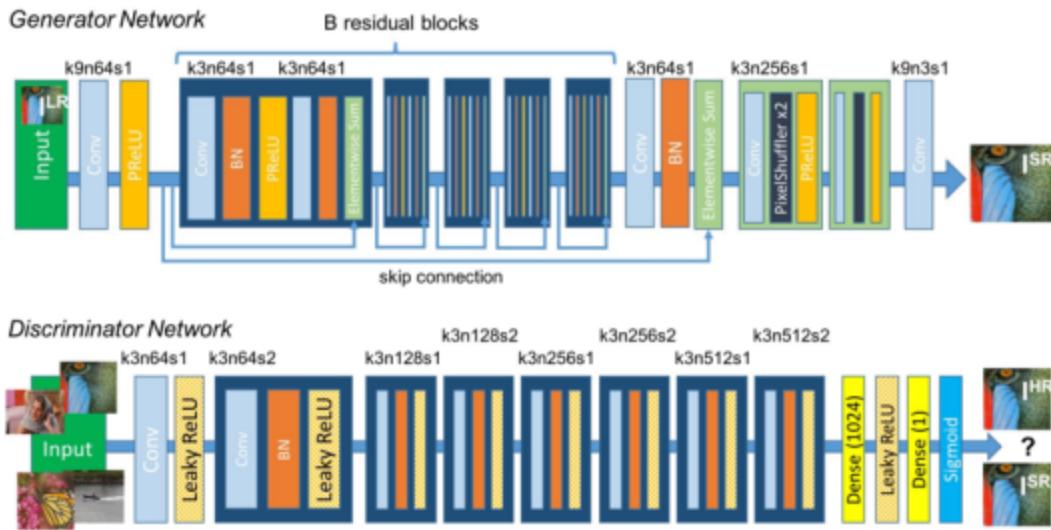


Figure 9.1: Architectures of generator and discriminator network from original SRGAN paper by Christian Ledig et al.

Figure 9.1 shows the generator and the discriminator architecture for SRGAN as shown in the paper. Let's note down the key points about both of these networks.

Let's first look the generator network.

1.1 SRGAN generator network

SRGAN generator network accepts a low-resolution image as input and converts it into a high-resolution image such that the underlying content of the image is unchanged. As shown in the Figure 9.1, SRGAN generator network is a deep convolutional neural network with residual connections (or skip connections). Each block consists of Convolutional layers with small kernel sizes (3×3), followed by the layers of Batch normalization and parametric-ReLU as the activation function.

Let's now look at the discriminator network.

1.2 SRGAN discriminator network

The discriminator network aims to distinguish the real high-resolution images from the generated super-resolution samples by the generator network. As shown in the Figure 9.1, the discriminator network follows the architecture guidelines of DCGAN discriminator such as avoiding max-pooling layers and using *LeakyReLU* (with, $\alpha = 0.2$) as the activation function.

Input images are down-sampled using multiple layers of strided convolutions and the resulting feature maps are passed into two dense layers. The resulting output is then passed through a sigmoid activation function to obtain a probability value for classification (or validity).

Let's now learn about the loss function of SRGAN model.

1.3 SRGAN Loss Functions

SRGAN uses a perceptual loss function which is a combination of adversarial loss and content loss. The perceptual loss function is critical for the performance of SRGAN as the mean squared error or MSE based pixel-wise loss fails to handle the high frequency details such as texture, and it results in an overly smooth output.

SRGAN loss function or the perceptual loss, is a weighted sum of the content loss and the adversarial loss:

$$\text{SRGAN-Loss} = \text{content-Loss} + 0.001 \times \text{adversarial-Loss}$$

Content loss: Instead of using pixel-wise MSE loss which had a problem of generating overly smooth textures, the content loss is a loss function that is closer to the perceptual similarity. In this method, a pre-trained image classification model such as *VGG19*, is used for extracting the feature representations (from an intermediate layer) of the generated high-resolution images as well as from the real high-resolution images. The content loss is then calculated as the *Euclidean* distance between the feature maps of the generated image and the real image.

Adversarial loss: In addition to the content loss, SRGAN also adds the adversarial loss, in order to encourage the generator to favor the outputs that reside on the manifold of the natural images, by trying to fool the discriminator network.

Let's now look at some of the impressive results of SRGAN from paper.

1.4 Results of SRGAN

SRGAN provides excellent upscaling (4x high-resolution) results that look very natural.



Figure 9.2: Results from original Super Resolution GAN paper by Christian Ledig et al.

Figure 9.2 shows some results from the original SRGAN paper and we can see that the output of SRGAN is way much closer to the original high-resolution image than other methods such as *bicubic interpolation* and *ResNet*.

Let's now jump into the experiment and try to replicate the amazing results of the SRGAN model.

The code samples shown in this skill can be downloaded from the following Github location:

<https://github.com/kartikgill/The-GAN-Book/tree/main/Skill-09>

Let's get started.

SUPER RESOLUTION GAN OR SR-GAN

Objective

In this experiment, we will implement and train SRGAN model and verify its results for generating high-resolution images.

This experiment has the following steps:

- Importing Libraries
- Download and Unzip Data
- Check few Samples
- Define Generator Network
- Define Discriminator Network
- Load Pre-trained VGG features
- Define SR-GAN
- Define Utility Functions
- Training SR-GAN
- Results

Let's get started.

Step 1: Importing Libraries

Open a Jupyter Notebook with python kernel and import useful python packages in a cell. In this experiment, we will use ‘*numpy*’ for data manipulation, ‘*matplotlib*’ for plotting images and graphs, and *TensorFlow2* for implementing the model architecture later.

Checkout the following snippet for importing libraries:

```
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
from tqdm import tqdm_notebook  
%matplotlib inline  
import tensorflow  
print(tensorflow.__version__)
```

Output: 2.4.0

Let's now get the dataset.

Step 2: Download and Unzip Data

We have already downloaded the face scrub dataset. In this step, we will extract the dataset and divide the images into two lists: train and test. The list 'train' stores 35k paths of training set images and the list 'test' stores about 7k paths of test set images.

See the following code:

```
!unzip /content/gdrive/MyDrive\  
/GAN_datasets/face_scrub.zip -d /  
import glob  
train = glob.glob('*_faces/*/*.jpeg')[:35000]  
test = glob.glob('*_faces/*/*.jpeg')[35000:]  
len(train), len(test)
```

Output: (35000, 7196)

Let's check few samples of the data.

Step 3: Check few Samples

In this step, we will choose few random indices and plot those images in low resolution as well as high resolution. We will create the low-res and high-res versions of the images using OpenCV library by converting them into the resolutions of 64 x 64 and 256 x 256 respectively. Similarly, we will create the dataset for the SRGAN model later.

See the following python code that shows few data samples:

```
import cv2  
files = np.random.choice(train, size=4)  
print ("Low quality Samples")  
for k in range(1):  
    plt.figure(figsize=(15, 15))  
    for j, file in enumerate(files):  
        img = cv2.imread(file)
```

```
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
img = cv2.resize(img, (64, 64))
plt.subplot(440 + 1 + j)
plt.imshow(img)
plt.axis('off')
#plt.title(trainY[i])
plt.show()
print ("-*"*100)
print ("Real High quality version")
for k in range(1):
    plt.figure(figsize=(15, 15))
    for j, file in enumerate(files):
        img = cv2.imread(file)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img = cv2.resize(img, (256, 256))
        plt.subplot(440 + 1 + j)
        plt.imshow(img)
        plt.axis('off')
        #plt.title(trainY[i])
        plt.show()
```

Figure 9.3 shows some sample images from the dataset. First row shows the low-resolution version while, the second row shows the high-resolution version of the same samples.

Output:

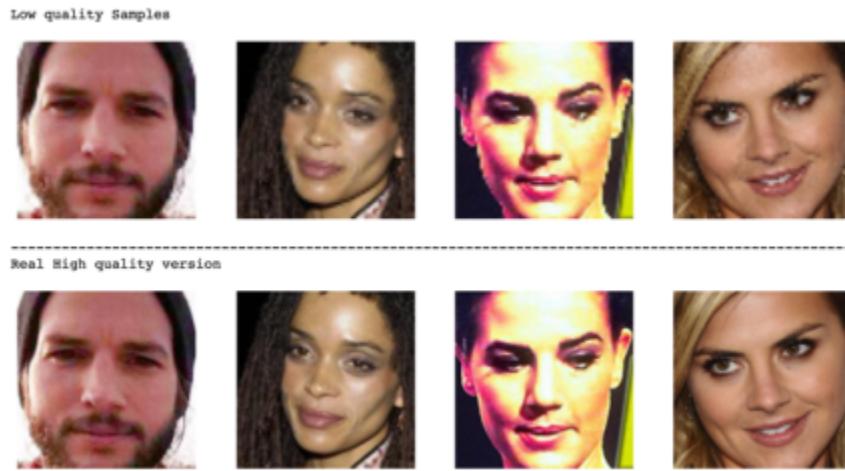


Figure 9.3: A few samples from the dataset of human faces for SRGAN.

Let's define the model architecture now.

Step 4: Define Generator Network

In this step, we will define the generator network architecture for our SRGAN setup. As suggested in the SRGAN paper, we will design a *ResNet* like architecture for our generator network with skip connections.

In our setup, the generator network accepts an image of size 64 x 64 x 3 as input (a low-resolution image), and passes it through multiple layers of strided *Conv2D* (for down-sampling), Batch Normalization with a momentum of 0.8, *ReLU* activation, Instance Normalization and, finally creates a skip connection using an addition layer (see the following python code). The final layer converts the feature vector into an output image of size 256 x 256 x 3 (a high-resolution image), and uses a '*tanh*' activation layer to restrict the pixel values of the translated image into a range of [-1, 1].

The following python code defines the generator network for our SRGAN setup:

```
import tensorflow_addons as tfa

def custom_resnet_block(input_layer, filters, \
upsample=False, resnet=True):
    x = input_layer
    if resnet==True:
```



```

x = custom_resnet_block(x, 64, False)
y = tensorflow.keras.layers.Conv2D(64, \
kernel_size=3, strides=1, padding='same')(x)
y = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(y)
y = tensorflow.keras.layers.Add()([y, first])
z = custom_resnet_block(y, 256, True, False)
z = custom_resnet_block(z, 256, True, False)
z = tensorflow.keras.layers.Conv2D(3, kernel_size=9,\ 
strides=1, padding='same')(z)
high_quality_image = tensorflow.keras.layers.Activation('tanh')(z)
generator_network = tensorflow.keras.models.Model(\ 
inputs=low_quality_image, outputs=high_quality_image)
print(generator_network.summary())

```

Following is the summary of our generator network. It roughly has 2M trainable parameters.

Output:

Model: "model"

Layer (type) Output Shape Param # Connected to

```
=====
=====
```

input_1 (InputLayer) [(None, 64, 64, 3)] 0

conv2d (Conv2D) (None, 64, 64, 64) 15616 input_1[0][0]

activation (Activation) (None, 64, 64, 64) 0 conv2d[0][0]

conv2d_1 (Conv2D) (None, 64, 64, 64) 36928 activation[0][0]

activation_1 (Activation) (None, 64, 64, 64) 0 conv2d_1[0][0]

batch_normalization (BatchNorma (None, 64, 64, 64) 256 activation_1[0][0]

conv2d_2 (Conv2D) (None, 64, 64, 64) 36928 batch_normalization[0][0]

instance_normalization (Instanc (None, 64, 64, 64) 128 conv2d_2[0][0]

add (Add) (None, 64, 64, 64) 0 instance_normalization[0][0]

activation[0][0]

conv2d_3 (Conv2D) (None, 64, 64, 64) 36928 add[0][0]

activation_2 (Activation) (None, 64, 64, 64) 0 conv2d_3[0][0]

batch_normalization_1 (BatchNor (None, 64, 64, 64) 256 activation_2[0][0]

conv2d_4 (Conv2D) (None, 64, 64, 64) 36928 batch_normalization_1[0][0]

instance_normalization_1 (Insta (None, 64, 64, 64) 128 conv2d_4[0][0]

add_1 (Add) (None, 64, 64, 64) 0 instance_normalization_1[0][0]

add[0][0]

conv2d_5 (Conv2D) (None, 64, 64, 64) 36928 add_1[0][0]

activation_3 (Activation) (None, 64, 64, 64) 0 conv2d_5[0][0]

batch_normalization_2 (BatchNor (None, 64, 64, 64) 256 activation_3[0][0]

conv2d_6 (Conv2D) (None, 64, 64, 64) 36928 batch_normalization_2[0][0]

instance_normalization_2 (Insta (None, 64, 64, 64) 128 conv2d_6[0][0]

add_2 (Add) (None, 64, 64, 64) 0 instance_normalization_2[0][0]

add_1[0][0]

conv2d_7 (Conv2D) (None, 64, 64, 64) 36928 add_2[0][0]

activation_4 (Activation) (None, 64, 64, 64) 0 conv2d_7[0][0]

batch_normalization_3 (BatchNor (None, 64, 64, 64) 256 activation_4[0][0]

conv2d_8 (Conv2D) (None, 64, 64, 64) 36928 batch_normalization_3[0][0]

instance_normalization_3 (Insta (None, 64, 64, 64) 128 conv2d_8[0][0]

add_3 (Add) (None, 64, 64, 64) 0 instance_normalization_3[0][0]

add_2[0][0]

conv2d_9 (Conv2D) (None, 64, 64, 64) 36928 add_3[0][0]

activation_5 (Activation) (None, 64, 64, 64) 0 conv2d_9[0][0]

batch_normalization_4 (BatchNor (None, 64, 64, 64) 256 activation_5[0][0]

conv2d_10 (Conv2D) (None, 64, 64, 64) 36928 batch_normalization_4[0][0]

instance_normalization_4 (Insta (None, 64, 64, 64) 128 conv2d_10[0][0]

add_4 (Add) (None, 64, 64, 64) 0 instance_normalization_4[0][0]

add_3[0][0]

conv2d_11 (Conv2D) (None, 64, 64, 64) 36928 add_4[0][0]

activation_6 (Activation) (None, 64, 64, 64) 0 conv2d_11[0][0]

batch_normalization_5 (BatchNor (None, 64, 64, 64) 256 activation_6[0][0]

conv2d_12 (Conv2D) (None, 64, 64, 64) 36928 batch_normalization_5[0][0]

instance_normalization_5 (Insta (None, 64, 64, 64) 128 conv2d_12[0][0]

add_5 (Add) (None, 64, 64, 64) 0 instance_normalization_5[0][0]

add_4[0][0]

conv2d_13 (Conv2D) (None, 64, 64, 64) 36928 add_5[0][0]

activation_7 (Activation) (None, 64, 64, 64) 0 conv2d_13[0][0]

batch_normalization_6 (BatchNor (None, 64, 64, 64) 256 activation_7[0][0]

conv2d_14 (Conv2D) (None, 64, 64, 64) 36928 batch_normalization_6[0][0]

instance_normalization_6 (Insta (None, 64, 64, 64) 128 conv2d_14[0][0]

add_6 (Add) (None, 64, 64, 64) 0 instance_normalization_6[0][0]

add_5[0][0]

conv2d_15 (Conv2D) (None, 64, 64, 64) 36928 add_6[0][0]

activation_8 (Activation) (None, 64, 64, 64) 0 conv2d_15[0][0]

batch_normalization_7 (BatchNor (None, 64, 64, 64) 256 activation_8[0][0]

conv2d_16 (Conv2D) (None, 64, 64, 64) 36928 batch_normalization_7[0][0]

instance_normalization_7 (Insta (None, 64, 64, 64) 128 conv2d_16[0][0]

add_7 (Add) (None, 64, 64, 64) 0 instance_normalization_7[0][0]

add_6[0][0]

conv2d_17 (Conv2D) (None, 64, 64, 64) 36928 add_7[0][0]

activation_9 (Activation) (None, 64, 64, 64) 0 conv2d_17[0][0]

batch_normalization_8 (BatchNor (None, 64, 64, 64) 256 activation_9[0][0]

conv2d_18 (Conv2D) (None, 64, 64, 64) 36928 batch_normalization_8[0][0]

instance_normalization_8 (Insta (None, 64, 64, 64) 128 conv2d_18[0][0]

add_8 (Add) (None, 64, 64, 64) 0 instance_normalization_8[0][0]

add_7[0][0]

conv2d_19 (Conv2D) (None, 64, 64, 64) 36928 add_8[0][0]

activation_10 (Activation) (None, 64, 64, 64) 0 conv2d_19[0][0]

batch_normalization_9 (BatchNor (None, 64, 64, 64) 256 activation_10[0][0]

conv2d_20 (Conv2D) (None, 64, 64, 64) 36928 batch_normalization_9[0][0]

instance_normalization_9 (Insta (None, 64, 64, 64) 128 conv2d_20[0][0]

add_9 (Add) (None, 64, 64, 64) 0 instance_normalization_9[0][0]

add_8[0][0]

conv2d_21 (Conv2D) (None, 64, 64, 64) 36928 add_9[0][0]

activation_11 (Activation) (None, 64, 64, 64) 0 conv2d_21[0][0]

batch_normalization_10 (BatchNo (None, 64, 64, 64) 256 activation_11[0][0]

conv2d_22 (Conv2D) (None, 64, 64, 64) 36928 batch_normalization_10[0][0]

instance_normalization_10 (Inst (None, 64, 64, 64) 128 conv2d_22[0][0]

add_10 (Add) (None, 64, 64, 64) 0 instance_normalization_10[0][0]

add_9[0][0]

conv2d_23 (Conv2D) (None, 64, 64, 64) 36928 add_10[0][0]

activation_12 (Activation) (None, 64, 64, 64) 0 conv2d_23[0][0]

batch_normalization_11 (BatchNo (None, 64, 64, 64) 256 activation_12[0][0]

conv2d_24 (Conv2D) (None, 64, 64, 64) 36928 batch_normalization_11[0][0]

instance_normalization_11 (Inst (None, 64, 64, 64) 128 conv2d_24[0][0]

add_11 (Add) (None, 64, 64, 64) 0 instance_normalization_11[0][0]

add_10[0][0]

conv2d_25 (Conv2D) (None, 64, 64, 64) 36928 add_11[0][0]

activation_13 (Activation) (None, 64, 64, 64) 0 conv2d_25[0][0]

batch_normalization_12 (BatchNo (None, 64, 64, 64) 256 activation_13[0][0]

conv2d_26 (Conv2D) (None, 64, 64, 64) 36928 batch_normalization_12[0][0]

instance_normalization_12 (Inst (None, 64, 64, 64) 128 conv2d_26[0][0]

add_12 (Add) (None, 64, 64, 64) 0 instance_normalization_12[0][0]

add_11[0][0]

conv2d_27 (Conv2D) (None, 64, 64, 64) 36928 add_12[0][0]

activation_14 (Activation) (None, 64, 64, 64) 0 conv2d_27[0][0]

batch_normalization_13 (BatchNo (None, 64, 64, 64) 256 activation_14[0][0]

conv2d_28 (Conv2D) (None, 64, 64, 64) 36928 batch_normalization_13[0][0]

instance_normalization_13 (Inst (None, 64, 64, 64) 128 conv2d_28[0][0]

add_13 (Add) (None, 64, 64, 64) 0 instance_normalization_13[0][0]

add_12[0][0]

conv2d_29 (Conv2D) (None, 64, 64, 64) 36928 add_13[0][0]

activation_15 (Activation) (None, 64, 64, 64) 0 conv2d_29[0][0]

batch_normalization_14 (BatchNo (None, 64, 64, 64) 256 activation_15[0][0]

conv2d_30 (Conv2D) (None, 64, 64, 64) 36928 batch_normalization_14[0][0]

instance_normalization_14 (Inst (None, 64, 64, 64) 128 conv2d_30[0][0]

add_14 (Add) (None, 64, 64, 64) 0 instance_normalization_14[0][0]

add_13[0][0]

conv2d_31 (Conv2D) (None, 64, 64, 64) 36928 add_14[0][0]

activation_16 (Activation) (None, 64, 64, 64) 0 conv2d_31[0][0]

batch_normalization_15 (BatchNo (None, 64, 64, 64) 256 activation_16[0][0]

conv2d_32 (Conv2D) (None, 64, 64, 64) 36928 batch_normalization_15[0][0]

instance_normalization_15 (Inst (None, 64, 64, 64) 128 conv2d_32[0][0]

add_15 (Add) (None, 64, 64, 64) 0 instance_normalization_15[0][0]

add_14[0][0]

conv2d_33 (Conv2D) (None, 64, 64, 64) 36928 add_15[0][0]

batch_normalization_16 (BatchNo (None, 64, 64, 64) 256 conv2d_33[0][0]

add_16 (Add) (None, 64, 64, 64) 0 batch_normalization_16[0][0]

activation[0][0]

up_sampling2d (UpSampling2D) (None, 128, 128, 64) 0 add_16[0][0]

conv2d_34 (Conv2D) (None, 128, 128, 256 147712 up_sampling2d[0][0]

activation_17 (Activation) (None, 128, 128, 256 0 conv2d_34[0][0]

up_sampling2d_1 (UpSampling2D) (None, 256, 256, 256 0 activation_17[0][0]

conv2d_35 (Conv2D) (None, 256, 256, 256 590080 up_sampling2d_1[0][0]

activation_18 (Activation) (None, 256, 256, 256 0 conv2d_35[0][0]

conv2d_36 (Conv2D) (None, 256, 256, 3) 62211 activation_18[0][0]

activation_19 (Activation) (None, 256, 256, 3) 0 conv2d_36[0][0]

Total params: 2,040,643

Trainable params: 2,038,467

Non-trainable params: 2,176

None

Let's now define the discriminator network.

Step 5: Define Discriminator Network

In this step, we will define the discriminator network for our SRGAN setup. In our setup, the discriminator network accepts an image of size 256 x 256 x 3 as input (a high-resolution image). This input image is then passed through multiple layers of strided *Conv2D*, *LeakyReLU* activation with an alpha value of 0.2, and an Instance Normalization layer (except for the first layer). The final output of our discriminator network is a patch of size 16 x

16×1 , as it follows a Patch GAN architecture (as discussed in the last skill). As the discriminator network is Patch GAN based, we can compile it using ‘*mean squared error*’ loss function. We will utilize Adam optimizer with a learning rate of 0.0002 and beta_1 value of 0.5 for updating the weights of the discriminator network.

The following python snippet defines the discriminator network for our SRGAN setup and compiles it:

```
def custom_d_block(input_layer, filters, strides, bn=True):
    x = tensorflow.keras.layers.Conv2D(filters, \
        kernel_size=3, strides=strides, \
        padding='same')(input_layer)
    x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
    if bn==True:
        x = tfa.layers.InstanceNormalization()(x)
    return x

high_quality_input = tensorflow.keras.layers.Input(shape=(256, 256, 3))
x = custom_d_block(high_quality_input, 64, 1, False)
x = custom_d_block(x, 64, 2, True)
x = custom_d_block(x, 128, 1, True)
x = custom_d_block(x, 128, 2, True)
x = custom_d_block(x, 256, 1, True)
x = custom_d_block(x, 256, 2, True)
x = custom_d_block(x, 512, 1, True)
x = custom_d_block(x, 512, 2, True)
x = tensorflow.keras.layers.Dense(1024)(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
real_vs_fake_patch = tensorflow.keras.layers.Dense(1, \
    activation='sigmoid')(x)
discriminator_network = tensorflow.keras.models.Model(\n    inputs=high_quality_input,\n    outputs=real_vs_fake_patch)
```

```
outputs=real_vs_fake_patch)
print (discriminator_network.summary())
adam_optimizer = tensorflow.keras.optimizers.Adam(\n    learning_rate=0.0002, beta_1=0.5)
discriminator_network.compile(loss='mse', \
optimizer=adam_optimizer, metrics=['accuracy'])
```

Following is the summary of the discriminator network. It roughly has 5M trainable parameters:

Output:

Model: "model_1"

Layer (type)	Output Shape	Param #
<hr/>		
input_2 (InputLayer)	[(None, 256, 256, 3)]	0
conv2d_37 (Conv2D)	(None, 256, 256, 64)	1792
leaky_re_lu (LeakyReLU)	(None, 256, 256, 64)	0
conv2d_38 (Conv2D)	(None, 128, 128, 64)	36928
leaky_re_lu_1 (LeakyReLU)	(None, 128, 128, 64)	0
instance_normalization_16 (I)	(None, 128, 128, 64)	128
conv2d_39 (Conv2D)	(None, 128, 128, 128)	73856
leaky_re_lu_2 (LeakyReLU)	(None, 128, 128, 128)	0
instance_normalization_17 (I)	(None, 128, 128, 128)	256
conv2d_40 (Conv2D)	(None, 64, 64, 128)	147584
leaky_re_lu_3 (LeakyReLU)	(None, 64, 64, 128)	0

instance_normalization_18 (I (None, 64, 64, 128) 256

conv2d_41 (Conv2D) (None, 64, 64, 256) 295168

leaky_re_lu_4 (LeakyReLU) (None, 64, 64, 256) 0

instance_normalization_19 (I (None, 64, 64, 256) 512

conv2d_42 (Conv2D) (None, 32, 32, 256) 590080

leaky_re_lu_5 (LeakyReLU) (None, 32, 32, 256) 0

instance_normalization_20 (I (None, 32, 32, 256) 512

conv2d_43 (Conv2D) (None, 32, 32, 512) 1180160

leaky_re_lu_6 (LeakyReLU) (None, 32, 32, 512) 0

instance_normalization_21 (I (None, 32, 32, 512) 1024

conv2d_44 (Conv2D) (None, 16, 16, 512) 2359808

leaky_re_lu_7 (LeakyReLU) (None, 16, 16, 512) 0

instance_normalization_22 (I (None, 16, 16, 512) 1024

dense (Dense) (None, 16, 16, 1024) 525312

leaky_re_lu_8 (LeakyReLU) (None, 16, 16, 1024) 0

dense_1 (Dense) (None, 16, 16, 1) 1025

Total params: 5,215,425

Trainable params: 5,215,425

Non-trainable params: 0

None

Our discriminator network is all set now. Let's download a pre-trained VGG network weights for the content loss.

Step 6: Load Pre-trained VGG features

In this step, we will load a pre-trained *VGG19* network. We will only load some intermediate weights of the model for feature calculation purpose. This pre-trained VGG will help us in comparing the contents of the generated high-resolution image with the real high-resolution image. This comparison will make sure that the high-resolution image generated by the generator network is not losing any useful information.

The following python code prepares the pre-trained *VGG19* network for feature creation:

```
image_input = tensorflow.keras.layers.Input(shape=(256, 256, 3))
pre_trained_vgg = tensorflow.keras.applications.vgg19.VGG19(\n    weights='imagenet', input_shape=(256, 256, 3), \
    include_top=False)
pre_trained_vgg_model = tensorflow.keras.models.Model(\n    inputs=pre_trained_vgg.input, \
    outputs=pre_trained_vgg.get_layer('block3_conv4').output)
pre_trained_image_feautures = pre_trained_vgg_model(image_input)
custom_vgg = tensorflow.keras.models.Model(inputs=image_input,\n    outputs=pre_trained_image_feautures)
```

Let's define the combined SRGAN model now.

Step 7: Define SR-GAN

In this step, we will combine the generator and the discriminator networks to get the final SRGAN architecture. The combined SRGAN model has two outputs: one validity output from the discriminator network, and VGG feature outputs for the high-resolution image generated by the generator network. This setup will have two inputs: one low-resolution image that the generator will convert into high-resolution and the real high-resolution version of the same image that the VGG network will use for comparing features. The discriminator network and the pre-trained VGG

network are kept frozen in this combined setup, as the idea here is to update the weights of the generator network only.

We can utilize the '*binary_crossentropy*' loss for the validity output and '*mean_squared_error*' loss for the content loss. The loss weights are kept 0.001 and 1 respectively, as suggested in the original paper.

The following python code defines and compiles the final SRGAN setup:

```
low_quality_image = tensorflow.keras.layers.Input(shape=(64, 64, 3))
high_quality_input = tensorflow.keras.layers.Input(shape=(256, 256, 3))
fake_high_quality_image = generator_network(low_quality_image)
discriminator_network.trainable=False
custom_vgg.trainable=False
d_output = discriminator_network(fake_high_quality_image)
fake_high_quality_features = custom_vgg(fake_high_quality_image)
sr_gan = tensorflow.keras.models.Model(
    inputs=[low_quality_image, high_quality_input],\
    outputs=[d_output, fake_high_quality_features])
print(sr_gan.summary())
# Compiling Models
sr_gan.compile(loss=['binary_crossentropy', 'mse'], \
    loss_weights=[0.001, 1], optimizer=adam_optimizer)
```

Following is the summary of the combined SRGAN model:

Output:

Model: "model_4"

Layer (type) Output Shape Param # Connected to

input_5 (InputLayer) [(None, 64, 64, 3)] 0

```
model (Functional) (None, 256, 256, 3) 2040643 input_5[0][0]
```

```
input_6 (InputLayer) [(None, 256, 256, 3) 0
```

```
model_1 (Functional) (None, 16, 16, 1) 5215425 model[0][0]
```

```
model_3 (Functional) (None, 64, 64, 256) 2325568 model[0][0]
```

```
=====
```

Total params: 9,581,636

Trainable params: 2,038,467

Non-trainable params: 7,543,169

None

Our SRGAN setup is ready now. Let's work on the training piece.

Step 8: Define Utility Functions

In this step, we will define utility functions that will help us during the training of the network. First, we will define a function that will generate a training batch of data. Each training batch will have the low-resolution version and high-resolution version of the same images. We will utilize the OpenCV library for creating the low-resolution version (64 x 64 x 3) for our

high-resolution images. Additionally, we will normalize the pixel values of all training images to a range of [-1, 1]. In addition to this, we will also define a utility function for plotting the results of the generator network, along with the low-resolution input image and the actual high-resolution image for comparison.

The following python code defines the utility functions for our setup:

```
from tqdm import tqdm_notebook

new_train = []
# choosing only good quality images for training
for file in tqdm_notebook(train):
    img = cv2.imread(file)
    if (img.shape[0] >= 256):
        new_train.append(file)
len(new_train)

def get_training_samples(batch_size):
    files = np.random.choice(new_train, size=batch_size)
    low_quality_images = []
    high_quality_images = []
    for file in files:
        img = cv2.imread(file)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img2 = cv2.resize(img, (256, 256))
        img3 = cv2.resize(img, (64, 64))
        low_quality_images.append((img3-127.5)/127.5)
        high_quality_images.append((img2-127.5)/127.5)
    low_quality_images = np.array(low_quality_images)
    high_quality_images = np.array(high_quality_images)
    return low_quality_images, high_quality_images

def show_generator_results(generator_network):
```

```

low_quality_images, high_quality_images = get_training_samples(3)
fake_high_quality_images = generator_network.\ 
predict_on_batch(low_quality_images)
print ("Low quality input images")
plt.figure(figsize=(13, 13))
for i in range(3):
    plt.subplot(330 + 1 + i)
    plt.imshow((low_quality_images[i]+1.0)/2.0)
    plt.axis('off')
    plt.show()
print ("Generated high quality images")
plt.figure(figsize=(13, 13))
for i in range(3):
    plt.subplot(330 + 1 + i)
    plt.imshow((fake_high_quality_images[i]+1.0)/2.0)
    plt.axis('off')
    plt.show()
print ("Real high quality images")
plt.figure(figsize=(13, 13))
for i in range(3):
    plt.subplot(330 + 1 + i)
    plt.imshow((high_quality_images[i]+1.0)/2.0)
    plt.axis('off')
    plt.show()

```

We are now all set to start the training.

Step 9: Training SR-GAN

In this step, we will write the training iteration loop for our SRGAN setup. In each training step, we will first update the weights of the generator network using two separate batches of real data and the generated data

respectively. As the discriminator network gives a patch of size 16 x 16 x 1 as output, we will supply a patch of similar size as label for our batches. The batch of real data will have a real patch (a patch filled with ones) while, the generated or fake data will have a fake patch (a patch filled with zeroes). Once the discriminator weights are updated, we will freeze its weights and update the weights of the generator network of our SRGAN setup. The validity output of the SRGAN setup will be compared with a real patch (to make the discriminator believe that it is a real sample), and the VGG features of the generated image will be compared with the VGG features of the real image to calculate the content loss. The gradient will first flow through the frozen discriminator network and then update the generator network weights.

The following python code defines the training setup for our SRGAN model:

```
epochs = 500
batch_size = 1
steps = 2000
for i in range(0, epochs):
    if (i%1 == 0):
        show_generator_results(generator_network)
    for j in range(steps):
        low_quality_images, high_quality_images = \
            get_training_samples(batch_size)
        fake_high_quality_images = generator_network.\ 
            predict_on_batch(low_quality_images)
        fake_patch = np.zeros((batch_size, 16, 16, 1))
        real_patch = np.ones((batch_size, 16, 16, 1))
        # Updating Discriminator weights
        discriminator_network.trainable=True
        loss_d_real = discriminator_network.train_on_batch(\ 
            high_quality_images, real_patch)
        loss_d_fake = discriminator_network.train_on_batch(\
```

```

fake_high_quality_images, fake_patch)
loss_d = np.add(loss_d_real, loss_d_fake)/2.0
# Make the Discriminator believe that these are
#real samples and calculate loss to train the generator
low_quality_images, high_quality_images = \
get_training_samples(batch_size)
discriminator_network.trainable=False
real_vgg_features = custom_vgg.predict_on_batch(high_quality_images)
# Updating Generator weights
loss_g = sr_gan.train_on_batch([low_quality_images,\n
high_quality_images], [real_patch, real_vgg_features])
if j%200 == 0:
    print ("Epoch:%.0f, Step:%.0f, D-Loss:%.3f, D-Acc:%.3f,\n
G-Loss:%.3f"%(i,j,loss_d[0],loss_d[1]*100,loss_g[0]))

```

Following are the training logs:

Output:

```

Epoch:0, Step:0, D-Loss:0.303, D-Acc:48.047, G-Loss:336.211
Epoch:0, Step:200, D-Loss:0.215, D-Acc:78.125, G-Loss:31.675
Epoch:0, Step:400, D-Loss:0.020, D-Acc:99.219, G-Loss:43.861
Epoch:0, Step:600, D-Loss:0.325, D-Acc:53.125, G-Loss:31.401
Epoch:0, Step:800, D-Loss:0.007, D-Acc:99.805, G-Loss:27.879
Epoch:0, Step:1000, D-Loss:0.006, D-Acc:99.609, G-Loss:30.598
Epoch:0, Step:1200, D-Loss:0.003, D-Acc:99.805, G-Loss:34.514
Epoch:0, Step:1400, D-Loss:0.058, D-Acc:94.531, G-Loss:47.920
Epoch:0, Step:1600, D-Loss:0.000, D-Acc:100.000, G-Loss:47.942
Epoch:0, Step:1800, D-Loss:0.000, D-Acc:100.000, G-Loss:20.406
. . . . .
. . . . .
. . . . .
. . . . .

```

As this model is quite big in size, we will only train it for a few steps just to verify the results.

Step 10: Results

Figure 9.4 shows the results of our SRGAN setup. As this was a pretty big model, the training was very slow and we couldn't train it for many epochs. Even though the results look really good.

In the Figure 9.4, the first row represents the low-resolution input images, the second row represents the high-resolution version of these images generated by our SRGAN model, and the final row shows the actual high-resolution version of the input images. We can see that our SRGAN model does a really great job and the output images look very high-quality and realistic. We can also see that the SRGAN output is not changing the details of the actual image, which was possible because of the content loss.

Output:

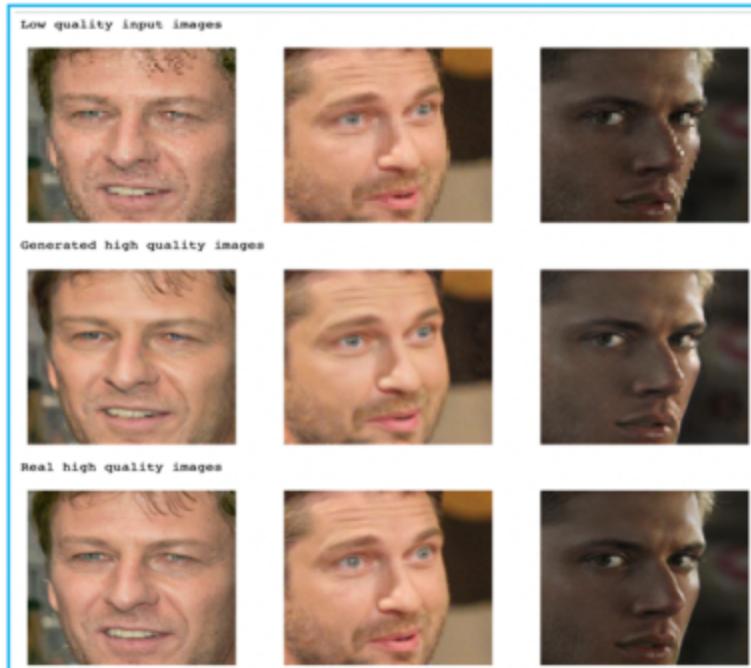


Figure 9.4: SRGAN results for Image Super Resolution. (Top row) low quality input images. (Middle row) high quality image output of SRGAN. (Bottom row) Actual high quality version.

We just saw that SRGAN model can produce high quality images for given low quality images as input. Next, let's learn about Disco GAN, another application area of GANs.

1. Disco GAN

Humans are really great at identifying cross-domain relations. They can observe how things, objects, people and concepts are connected. Finding cross-domain relations is quite natural for humans such as finding matching shoes for a given dress. However, finding such relations using an AI model automatically, is very challenging and often requires a large number of paired examples to understand the relation of two domains.

In 2017, *Taeksoo Kim, et al.* proposed a method that automatically learns to discover relations between two different domains without requiring paired examples. This method, also called *Disco GAN*, is based on GANs and was explained in their research paper titled "*Learning to Discover Cross-Domain Relations with Generative Adversarial Networks*".

Disco GAN is interesting because it doesn't require paired samples to learn relations between two domains. It is an interesting problem to solve because it's not always possible to obtain paired examples for many scenarios. Basically, the Disco GAN takes an image from one domain as an input and generates its corresponding image in another domain.

Let's learn about Disco GAN architecture.

2.1 Disco-GAN architecture design

Disco GAN is made up of two GANs coupled together where each of them aims at learning a mapping function from one domain to its counterpart domain. Thus, the learned mapping is bidirectional.

Suppose, A and B are two different domains and we want to learn cross-domain relations between them in an unsupervised manner, using the Disco GAN framework. The components of the Disco GAN model for this case can be defined as follows:

Generator AB: learns a mapping function from Domain A to Domain B

Discriminator B: validates the images of Domain B (real vs fake)

Generator BA: learns a mapping function from Domain B to Domain A

Discriminator A: validates the images of Domain A (real vs fake)

Each generator is an encoder-decoder kind of model where the encoder part is composed of convolutional layers followed by *LeakyReLU* activations and the decoder part is composed of de-convolutional layers followed by *ReLU* activations. Images from the source domain are fed into the encoder part and the decoder part outputs the corresponding target domain images.

The discriminators are similar to the encoder part of the generator networks, followed by a final *sigmoid* activation layer to output a scalar value between [0, 1] or probability to decide real images from the fake ones coming from the generators.

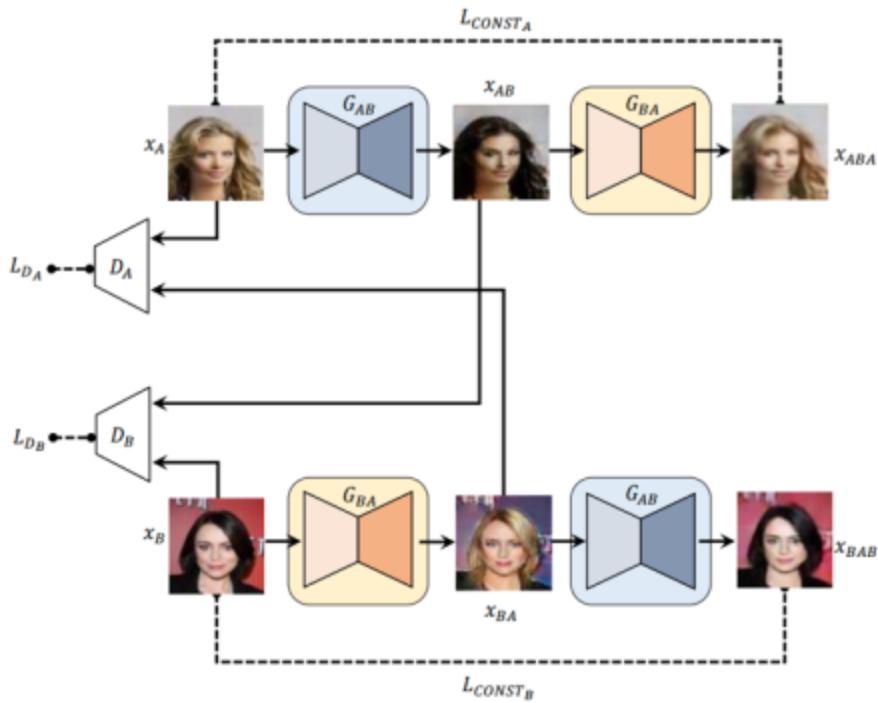


Figure 9.5: Disco-GAN design as per the original paper by Taeksoo Kim, et al.

Figure 9.5 shows the Disco GAN setup as per the paper. Each of the two coupled generator networks, learn the mapping from one domain to another

and both of them are trained simultaneously. Each generator network is trained with two losses: the reconstruction loss and the adversarial loss.

The training of the Disco GAN progresses in the following manner:

- Generator AB translates an image from domain A to domain B
- Generator BA takes this translated image as input and translates it back to domain A
- This doubly translated image should be exactly similar to the original input image. To calculate the distance between these two images, various functions such as cosine similarity, *MSE* or hinge-loss can be used as the “*reconstruction loss*”.
- Secondly, the translated image from Generator AB is passed to the Discriminator B to check how closely it represents the real domain B images and the resulting “*adversarial loss*” is used to improve the Generator AB.
- Generator AB is updated using two losses: reconstruction loss and the adversarial loss.
- Similarly, Generator BA is also trained simultaneously to learn the reverse mapping.

Thus, for each generator network:

$$\textbf{\textit{Generator loss}} = \textbf{\textit{Reconstruction loss}} + \textbf{\textit{Adversarial loss}}$$

Let's now look at some of the impressive results of Disco GAN.

2.2 Results of Disco GAN

To evaluate the capabilities of Disco GAN setup, it was trained and tested using several image-to-image translation tasks that require the use of the automatically discovered cross-domain relations between source and target domains. The original paper shares results on some interesting tasks such as:

- Translation of Gender

- Blonde to Black hair conversion
- Wearing eye-glasses conversion
- Handbags to related shoes translation

Figure 9.6 shows the Disco GAN output for ‘*Handbags to Shoes*’ relation task. We can see that the model is able to successfully creating matching shoes for given handbag images and vice versa.

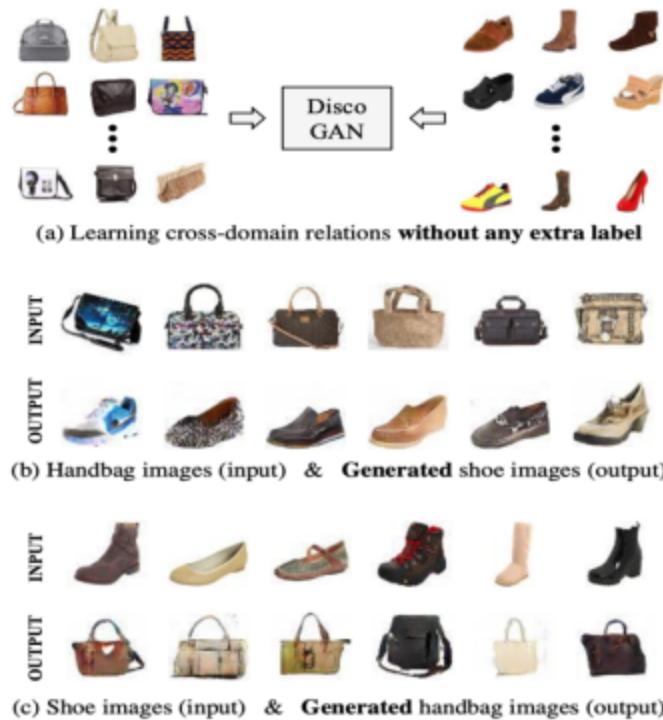


Figure 9.6: Results of cross-domain relation translation from original Disco-GAN paper

We just saw some great results from Disco GAN. Let’s now replicate some results with an experiment.

The code samples shown in this skill can be downloaded from the following Github location:

<https://github.com/kartikgill/The-GAN-Book/tree/main/Skill-09>

Let’s get started.

DISCO GAN FOR MALE TO FEMALE EXPERIMENT

Objective

In this experiment, we will implement and train a Disco GAN model for gender conversion use case and verify its capabilities.

This experiment has the following steps:

- Importing Libraries
- Download and Unzip Data
- Check few Samples
- Define Generator Network
- Define Discriminator Network
- Define Disco-GAN
- Define Utility Functions
- Training Disco-GAN
- Results

Let's get started.

Step 1: Importing Libraries

Open a Jupyter Notebook with python kernel and import useful python packages in a cell. In this experiment, we will use ‘*numpy*’ for data manipulation, ‘*matplotlib*’ for plotting images and graphs, and *TensorFlow2* for implementing the model architecture later.

Checkout the following snippet for importing libraries:

```
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
from tqdm import tqdm_notebook  
%matplotlib inline  
import tensorflow  
print(tensorflow.__version__)
```

Output: 2.4.0

Let's now collect the dataset.

Step 2: Download and Unzip Data

We have already downloaded the face scrub dataset. In this step, we will extract the dataset and divide the images into two lists: actors and actress. The list 'actors' stores about 22k paths of Male face images and the list 'actress' stores about 19k paths of Female face images.

See the following code:

```
!unzip /content/gdrive/MyDrive/\
GAN_datasets/face_scrub.zip -d /
import glob
actors = glob.glob('/actor_faces/*/*.jpeg')
actress = glob.glob('/actress_faces/*/*.jpeg')
len(actors), len(actress)
```

Output: (22809, 19387)

Let's check out few samples from the dataset.

Step 3: Check few Samples

As our final goal is to learn the gender conversion using a Disco GAN. We have divided the dataset into two sets of Males and Females. The following python code shows some samples from both of these sets:

```
print ("Male Actor Faces")
for k in range(2):
    plt.figure(figsize=(13, 13))
    for j in range(9):
        file = np.random.choice(actors)
        img = cv2.imread(file)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        plt.subplot(990 + 1 + j)
        plt.imshow(img)
```

```

plt.axis('off')
#plt.title(trainY[i])
plt.show()
print ("-*80)
print ("Female Actress Faces")
for k in range(2):
    plt.figure(figsize=(13, 13))
    for j in range(9):
        file = np.random.choice(actress)
        img = cv2.imread(file)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        plt.subplot(990 + 1 + j)
        plt.imshow(img)
        plt.axis('off')
    #plt.title(trainY[i])
    plt.show()

```

Output:

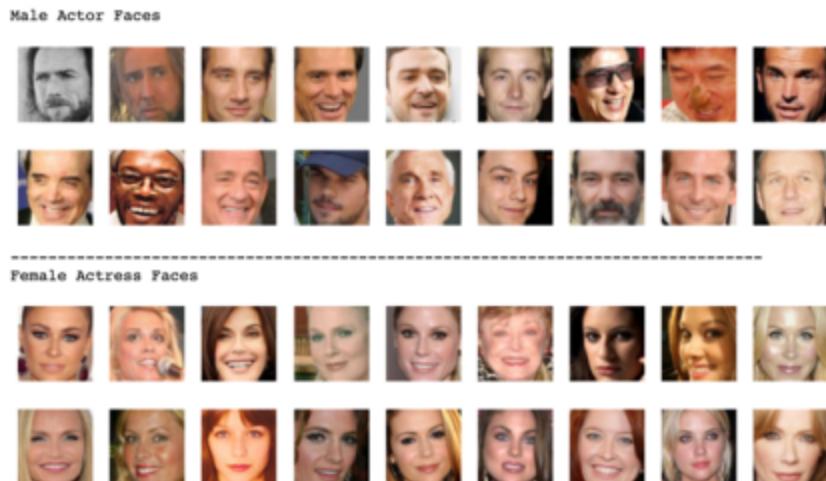


Figure 9.7: Few samples of Male faces (top two rows) and Female faces (bottom two rows).

Figure 9.7, shows some sample images from the dataset. Our dataset is now ready. Let's define the model architecture now.

Step 4: Define Generator Network

In this step, we will define the generator network architecture for our Disco GAN setup. As suggested in the Disco GAN paper, we will design a *UNet* like architecture for our generator network with skip connections. In our setup, the generator network accepts an image of size 128 x 128 x 3 as input (an image from the source domain), and passes it through multiple layers of strided *Conv2D* (for down-sampling), *LeakyReLU* activation with an alpha value of 0.2, and Instance Normalization. Then the resulting features are passed through a bottleneck layer that is a combination of a *Conv2D* layer and *ReLU* activation. The output of the bottleneck layer along with the outputs of previous encoder layers are passed through the decoder network. Each decoder block passes the input features through the layers of *Conv2DTranspose*, *ReLU* activation, Instance Normalization, and finally creates a *UNet* like skip connection using a concatenation layer (see the following python code). The final decoder layer, converts the decoded feature vector into an output image of size 128 x 128 x 3, and uses a '*tanh*' layer to restrict the pixel values of the translated image into a range of [-1, 1].

Using these design guidelines, we will define two exactly similar generator networks; one for each domain. The generator network AB, will translate the images from domain A to domain B. While, the generator network BA, will translate the images from domain B to domain A. The inherent architecture is exactly same.

The following python code defines the generator networks for our Disco GAN setup:

```
import tensorflow_addons as tfa

def encoder_layer(input_layer, filters, bn=True):
    x = tensorflow.keras.layers.Conv2D(filters, \
        kernel_size=(4,4), strides=(2,2),\
        padding='same')(input_layer)
    x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
```

```

if bn:
    x = tfa.layers.InstanceNormalization()(x)
    return x

def decoder_layer(input_layer, skip_input, filters):
    x = tensorflow.keras.layers.Conv2DTranspose(filters,\n
    kernel_size=(4,4), strides=(2,2),\n
    padding='same')(input_layer)
    x = tensorflow.keras.layers.Activation('relu')(x)
    x = tfa.layers.InstanceNormalization()(x)
    x = tensorflow.keras.layers.Concatenate()([x, skip_input])
    return x

source_image_A = tensorflow.keras.layers.Input(shape=(128, 128, 3))
source_image_B = tensorflow.keras.layers.Input(shape=(128, 128, 3))

def make_generator():
    source_image = tensorflow.keras.layers.Input(shape=(128, 128, 3))
    e1 = encoder_layer(source_image, 64, bn=False)
    e2 = encoder_layer(e1, 128)
    e3 = encoder_layer(e2, 256)
    # e4 = encoder_layer(e3, 256)
    e5 = encoder_layer(e3, 512)
    e6 = encoder_layer(e5, 512)
    e7 = encoder_layer(e6, 512)
    bottle_neck = tensorflow.keras.layers.Conv2D(512,\n
    (4,4), strides=(2,2), padding='same')(e7)
    b = tensorflow.keras.layers.Activation('relu')(bottle_neck)
    d1 = decoder_layer(b, e7, 512)
    d2 = decoder_layer(d1, e6, 512)
    d3 = decoder_layer(d2, e5, 512)

```

```

# d4 = decoder_layer(d3, e4, 256)
d5 = decoder_layer(d3, e3, 256)
d6 = decoder_layer(d5, e2, 128)
d7 = decoder_layer(d6, e1, 64)
decoded = tensorflow.keras.layers.Conv2DTranspose(3, \
kernel_size=(4,4), strides=(2,2), padding='same')(d7)
translated_image = tensorflow.keras.layers.Activation('tanh')(decoded)
return source_image, translated_image

source_image, translated_image = make_generator()
generator_network_AB = tensorflow.keras.models.Model(\n
inputs=source_image, outputs=translated_image)
source_image, translated_image = make_generator()
generator_network_BA = tensorflow.keras.models.Model(\n
inputs=source_image, outputs=translated_image)
print(generator_network_AB.summary())

```

Following is the summary of one of the generator networks. It roughly has 42M trainable parameters.

Output:

Model: "model"

Layer (type) Output Shape Param # Connected to

input_6 (InputLayer) [(None, 128, 128, 3) 0]

conv2d_2 (Conv2D) (None, 64, 64, 64) 3136 input_6[0][0]

leaky_re_lu_1 (LeakyReLU) (None, 64, 64, 64) 0 conv2d_2[0][0]

conv2d_3 (Conv2D) (None, 32, 32, 128) 131200 leaky_re_lu_1[0][0]

leaky_re_lu_2 (LeakyReLU) (None, 32, 32, 128) 0 conv2d_3[0][0]

instance_normalization (Instanc (None, 32, 32, 128) 256 leaky_re_lu_2[0][0]

conv2d_4 (Conv2D) (None, 16, 16, 256) 524544 instance_normalization[0][0]

leaky_re_lu_3 (LeakyReLU) (None, 16, 16, 256) 0 conv2d_4[0][0]

instance_normalization_1 (Insta (None, 16, 16, 256) 512 leaky_re_lu_3[0][0]

conv2d_5 (Conv2D) (None, 8, 8, 512) 2097664 instance_normalization_1[0][0]

leaky_re_lu_4 (LeakyReLU) (None, 8, 8, 512) 0 conv2d_5[0][0]

instance_normalization_2 (Insta (None, 8, 8, 512) 1024 leaky_re_lu_4[0][0]

conv2d_6 (Conv2D) (None, 4, 4, 512) 4194816 instance_normalization_2[0][0]

leaky_re_lu_5 (LeakyReLU) (None, 4, 4, 512) 0 conv2d_6[0][0]

instance_normalization_3 (Insta (None, 4, 4, 512) 1024 leaky_re_lu_5[0][0]

conv2d_7 (Conv2D) (None, 2, 2, 512) 4194816 instance_normalization_3[0][0]

leaky_re_lu_6 (LeakyReLU) (None, 2, 2, 512) 0 conv2d_7[0][0]

instance_normalization_4 (Insta (None, 2, 2, 512) 1024 leaky_re_lu_6[0][0]

conv2d_8 (Conv2D) (None, 1, 1, 512) 4194816 instance_normalization_4[0][0]

activation (Activation) (None, 1, 1, 512) 0 conv2d_8[0][0]

conv2d_transpose (Conv2DTranspo (None, 2, 2, 512) 4194816 activation[0][0]

activation_1 (Activation) (None, 2, 2, 512) 0 conv2d_transpose[0][0]

instance_normalization_5 (Insta (None, 2, 2, 512) 1024 activation_1[0][0]

concatenate (Concatenate) (None, 2, 2, 1024) 0 instance_normalization_5[0][0]

instance_normalization_4[0][0]

conv2d_transpose_1 (Conv2DTrans (None, 4, 4, 512) 8389120 concatenate[0][0]

activation_2 (Activation) (None, 4, 4, 512) 0 conv2d_transpose_1[0][0]

instance_normalization_6 (Insta (None, 4, 4, 512) 1024 activation_2[0][0]

concatenate_1 (Concatenate) (None, 4, 4, 1024) 0 instance_normalization_6[0][0]

instance_normalization_3[0][0]

conv2d_transpose_2 (Conv2DTrans (None, 8, 8, 512) 8389120 concatenate_1[0][0]

activation_3 (Activation) (None, 8, 8, 512) 0 conv2d_transpose_2[0][0]

instance_normalization_7 (Insta (None, 8, 8, 512) 1024 activation_3[0][0]

concatenate_2 (Concatenate) (None, 8, 8, 1024) 0 instance_normalization_7[0][0]

instance_normalization_2[0][0]

conv2d_transpose_3 (Conv2DTrans (None, 16, 16, 256) 4194560 concatenate_2[0][0]

activation_4 (Activation) (None, 16, 16, 256) 0 conv2d_transpose_3[0][0]

instance_normalization_8 (Insta (None, 16, 16, 256) 512 activation_4[0][0]

concatenate_3 (Concatenate) (None, 16, 16, 512) 0 instance_normalization_8[0][0]

instance_normalization_1[0][0]

conv2d_transpose_4 (Conv2DTrans (None, 32, 32, 128) 1048704 concatenate_3[0][0]

activation_5 (Activation) (None, 32, 32, 128) 0 conv2d_transpose_4[0][0]

instance_normalization_9 (Insta (None, 32, 32, 128) 256 activation_5[0][0]

concatenate_4 (Concatenate) (None, 32, 32, 256) 0 instance_normalization_9[0][0]

instance_normalization[0][0]

conv2d_transpose_5 (Conv2DTrans (None, 64, 64, 64) 262208 concatenate_4[0][0]

activation_6 (Activation) (None, 64, 64, 64) 0 conv2d_transpose_5[0][0]

instance_normalization_10 (Inst (None, 64, 64, 64) 128 activation_6[0][0]

concatenate_5 (Concatenate) (None, 64, 64, 128) 0 instance_normalization_10[0][0]

leaky_re_lu_1[0][0]

conv2d_transpose_6 (Conv2DTrans (None, 128, 128, 3) 6147 concatenate_5[0][0]

```
activation_7 (Activation) (None, 128, 128, 3) 0 conv2d_transpose_6[0][0]
```

Total params: 41,833,475

Trainable params: 41,833,475

Non-trainable params: 0

None

Let's define the discriminators now.

Step 5: Define Discriminator Network

In this step, we will define the discriminator network for our Disco GAN setup. In our setup, the discriminator network accepts an image of size 128 x 128 x 3 as input (the translated image). This input image is then passed through multiple layers of strided *Conv2D*, *LeakyReLU* activation with an alpha value of 0.2, and an Instance Normalization layer (except for the first layer). The final output of our discriminator network is a patch of size 8 x 8 x 1, as it follows a Patch GAN architecture (as discussed in the last skill).

In our Disco GAN setup, we will require two exactly same discriminators; one for each domain. The discriminator network A, will verify the translated version of domain A, while the discriminator network B, will verify the translated version of domain B. As both the discriminators are Patch GAN based, we can compile them using '*mean squared error (mse)*' loss function.

The following python snippet defines the discriminators for our Disco GAN setup and compiles them:

```
def my_conv_layer(input_layer, filters, bn=True):
    x = tensorflow.keras.layers.Conv2D(filters, \
        kernel_size=(4,4), strides=(2,2),\
        padding='same')(input_layer)
```

```
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
if bn:
    x = tfa.layers.InstanceNormalization()(x)
return x

def make_discriminator():
    target_image_input = tensorflow.keras.layers.Input(shape=(128, 128, 3))
    x = my_conv_layer(target_image_input, 64, bn=False)
    x = my_conv_layer(x, 128)
    x = my_conv_layer(x, 256)
    # x = my_conv_layer(x, 512)
    x = my_conv_layer(x, 512)
    patch_features = tensorflow.keras.layers.Conv2D(1, \
        kernel_size=(4,4), strides=(1,1), padding='same')(x)
    return target_image_input, patch_features
```

```
target_image_input, patch_features = make_discriminator()
discriminator_network_A = tensorflow.keras.models.Model(\n
    inputs=target_image_input, outputs=patch_features)
target_image_input, patch_features = make_discriminator()
discriminator_network_B = tensorflow.keras.models.Model(\n
    inputs=target_image_input, outputs=patch_features)
print (discriminator_network_A.summary())
adam_optimizer = tensorflow.keras.optimizers.Adam(\n
    learning_rate=0.0002, beta_1=0.5)
discriminator_network_A.compile(loss='mse', \n
    optimizer=adam_optimizer, metrics=['accuracy'])
discriminator_network_B.compile(loss='mse', \n
```

```
optimizer=adam_optimizer, metrics=['accuracy'])
```

Following is the summary of one of the discriminators. It roughly has 2.8M trainable parameters.

Output:

Model: "model_2"

Layer (type) Output Shape Param #

input_8 (InputLayer) [(None, 128, 128, 3)] 0

conv2d_16 (Conv2D) (None, 64, 64, 64) 3136

leaky_re_lu_13 (LeakyReLU) (None, 64, 64, 64) 0

conv2d_17 (Conv2D) (None, 32, 32, 128) 131200

leaky_re_lu_14 (LeakyReLU) (None, 32, 32, 128) 0

instance_normalization_22 (I (None, 32, 32, 128) 256

conv2d_18 (Conv2D) (None, 16, 16, 256) 524544

leaky_re_lu_15 (LeakyReLU) (None, 16, 16, 256) 0

instance_normalization_23 (I (None, 16, 16, 256) 512

conv2d_19 (Conv2D) (None, 8, 8, 512) 2097664

leaky_re_lu_16 (LeakyReLU) (None, 8, 8, 512) 0

instance_normalization_24 (I (None, 8, 8, 512) 1024

conv2d_20 (Conv2D) (None, 8, 8, 1) 8193

Total params: 2,766,529

Trainable params: 2,766,529

Non-trainable params: 0

None

Let's define the combined Disco GAN model now.

Step 6: Define Disco-GAN

In this step, we will define the final combined Disco GAN architecture. The Disco GAN model is a combination of four models: two generator networks and two discriminator networks.

The generator network AB, translates images from domain A to domain B, and the discriminator B validates these translated images. Similarly, the generator BA, translates images from domain B to domain A, and the discriminator A validates the translated images. The discriminator networks are Patch GAN based, so they basically validate if the given image is real or fake representation of the given domain.

In the combined model: both discriminator networks are kept frozen as the idea of the combined model is to only update the weights of the generator networks using the frozen discriminators as a gradient supplier. The combined Disco GAN model has 6 outputs: two validity outputs (one for each domain), two translated outputs (one for each domain), and two cycle consistency outputs (one for each domain). Validity outputs come from the discriminator networks, indicating the real vs. fake feedback.

The cycle consistency output is generated using both the generators. The idea is simple: one source image from domain A is translated to domain B using the generator AB, then the translated image is passed into the generator BA and it translates it back to domain A. Now, as per the cycle consistency, this doubly converted image which now represents domain A should be exactly same to the original source image that also belongs to domain A. In the similar manner, the cycle consistency is checked for domain B as well.

For these six outputs, the model needs 6 loss functions. It uses '*mse*' loss for validity outputs and '*mae*' loss for image comparison outputs.

The following python code defines and compiles the final Disco GAN setup:

```
# Domain Transfer
```

```

fake_B = generator_network_AB(source_image_A)
fake_A = generator_network_BA(source_image_B)
# Restoring original Domain
get_back_A = generator_network_BA(fake_B)
get_back_B = generator_network_AB(fake_A)
discriminator_network_A.trainable=False
discriminator_network_B.trainable=False
# Tell Real vs Fake, for a given domain
verify_A = discriminator_network_A(fake_A)
verify_B = discriminator_network_B(fake_B)
disco_gan = tensorflow.keras.models.Model(
    inputs = [source_image_A, source_image_B], \
    outputs = [verify_A, verify_B, fake_B,\ 
    fake_A, get_back_A, get_back_B])
disco_gan.summary()
# Compiling Models
disco_gan.compile(loss=['mse', 'mse', 'mae', \
'mae', 'mae', 'mae'], optimizer=adam_optimizer)

```

Following is the summary of the combined Disco GAN setup:

Output:

Model: "model_4"

Layer (type) Output Shape Param # Connected to

```
=====
=====
```

input_5 (InputLayer) [(None, 128, 128, 3) 0

input_4 (InputLayer) [(None, 128, 128, 3) 0

model_1 (Functional) (None, 128, 128, 3) 41833475 input_5[0][0]

model[0][0]

model (Functional) (None, 128, 128, 3) 41833475 input_4[0][0]

model_1[0][0]

model_2 (Functional) (None, 8, 8, 1) 2766529 model_1[0][0]

model_3 (Functional) (None, 8, 8, 1) 2766529 model[0][0]

=====

Total params: 89,200,008

Trainable params: 83,666,950

Non-trainable params: 5,533,058

Our Disco GAN setup is ready. Let's work on the training piece now.

Step 7: Define Utility Functions

In this step, we will define some utility functions that will help us in creating batches of the data for training. Specifically, we will define the utility functions for: generating the actor to actress results from the first generator, generating the actress to actor results from the second generator, generating a real data batch from actors, generating a real data batch from actresses, and two utility functions for plotting the results of both the generators.

The following python code defines the utility functions:

```
def actors_to_actress(actors, generator_network):
    generated_samples = generator_network.predict_on_batch(actors)
    return generated_samples

def actress_to_actors(actress, generator_network):
    generated_samples = generator_network.predict_on_batch(actress)
    return generated_samples

def get_actor_samples(batch_size):
    random_files = np.random.choice(actors, size=batch_size)
    images = []
    for file in random_files:
        img = cv2.imread(file)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img = cv2.resize(img, (128, 128))
        images.append((img-127.5)/127.5)
    actor_images = np.array(images)
    return actor_images

def get_actress_samples(batch_size):
    random_files = np.random.choice(actress, size=batch_size)
    images = []
    for file in random_files:
        img = cv2.imread(file)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
```

```
img = cv2.resize(img, (128, 128))
images.append((img-127.5)/127.5)
actress_images = np.array(images)
return actress_images

def show_generator_results_actor_to_actress(\n    generator_network_AB, generator_network_BA):\n    images = []\n    for j in range(7):\n        file = np.random.choice(actors)\n        img = cv2.imread(file)\n        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)\n        img = cv2.resize(img, (128, 128))\n        images.append(img)\n    print ('Input Actor Images')\n    plt.figure(figsize=(13, 13))\n    for j, img in enumerate(images):\n        plt.subplot(770 + 1 + j)\n        plt.imshow(img)\n        plt.axis('off')\n        #plt.title(trainY[i])\n    plt.show()\n    print ('Translated (Actor -> Actress) Images')\n    translated = []\n    plt.figure(figsize=(13, 13))\n    for j, img in enumerate(images):\n        img = (img-127.5)/127.5\n        output = actors_to_actress(np.array([img]),\n            generator_network_AB)[0]
```

```
translated.append(output)
output = (output+1.0)/2.0
plt.subplot(770 + 1 + j)
plt.imshow(output)
plt.axis('off')
#plt.title(trainY[i])
plt.show()
print ('Translated reverse ( Fake Actress -> Fake Actor)')
plt.figure(figsize=(13, 13))
for j, img in enumerate(translated):
    output = actress_to_actors(np.array([img]),\
generator_network_BA)[0]
    output = (output+1.0)/2.0
    plt.subplot(770 + 1 + j)
    plt.imshow(output)
    plt.axis('off')
    #plt.title(trainY[i])
    plt.show()

def show_generator_results_actress_to_actor(\n    generator_network_AB, generator_network_BA):\n    images = []
    for j in range(7):\n        file = np.random.choice(actress)\n        img = cv2.imread(file)\n        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)\n        img = cv2.resize(img, (128, 128))\n        images.append(img)
    print ('Input Actress Images')
```

```
plt.figure(figsize=(13, 13))
for j, img in enumerate(images):
    plt.subplot(770 + 1 + j)
    plt.imshow(img)
    plt.axis('off')
    #plt.title(trainY[i])
    plt.show()
print ('Translated (Actress -> Actor) Images')
translated = []
plt.figure(figsize=(13, 13))
for j, img in enumerate(images):
    img = (img-127.5)/127.5
    output = actress_to_actors(np.array([img]),\
generator_network_BA)[0]
    translated.append(output)
    output = (output+1.0)/2.0
    plt.subplot(770 + 1 + j)
    plt.imshow(output)
    plt.axis('off')
    #plt.title(trainY[i])
    plt.show()
print ('Translated reverse (Fake Actor -> Fake Actress)')
plt.figure(figsize=(13, 13))
for j, img in enumerate(translated):
    output = actors_to_actress(np.array([img]),\
generator_network_AB)[0]
    output = (output+1.0)/2.0
    plt.subplot(770 + 1 + j)
```

```
plt.imshow(output)
plt.axis('off')
#plt.title(trainY[i])
plt.show()
```

We are now all set to start the training.

Step 8: Training Disco GAN

In this step, we will write the training iteration loop for our Disco GAN. In each training step, we will first update the weights of the discriminator networks using separate batches of the real translation outputs and the fake translation outputs of each domain. It is advised to train the discriminators on separate batches of real and fake samples. The output of each discriminator network is a patch of size $8 \times 8 \times 1$, and it is compared with a similar patch of similar size. The patch of the real images is filled with ones, while the patch of the fake images is filled with zeroes.

Then, we freeze the weights of the discriminator networks and update the weights of both generator networks in a single pass, by passing the batches of the input images, images from target domain and patches of real images (filled with ones). Here, we pass the patches of real images to make the discriminators believe that they are the real translations of the input images and calculate the loss values.

The following python code defines the training setup for our Disco GAN:

```
epochs = 500
batch_size = 1
steps = 2000
for i in range(0, epochs):
    if i%1 == 0:
        show_generator_results_actor_to_actress(
            generator_network_AB, generator_network_BA)
        print ("-*100")
        show_generator_results_actress_to_actor(
            generator_network_AB, generator_network_BA)
```

```

for j in range(steps):
    # A == Actor
    # B == Actress
    domain_A_images = get_actor_samples(batch_size)
    domain_B_images = get_actress_samples(batch_size)
    fake_patch = np.zeros((batch_size, 8, 8, 1))
    real_patch = np.ones((batch_size, 8, 8, 1))
    fake_B_images = generator_network_AB(domain_A_images)
    fake_A_images = generator_network_BA(domain_B_images)
    # Updating Discriminator A weights
    discriminator_network_A.trainable=True
    discriminator_network_B.trainable=False
    loss_d_real_A = discriminator_network_A.\.
        train_on_batch(domain_A_images, real_patch)
    loss_d_fake_A = discriminator_network_A.\.
        train_on_batch(fake_A_images, fake_patch)
    loss_d_A = np.add(loss_d_real_A, loss_d_fake_A)/2.0
    # Updating Discriminator B weights
    discriminator_network_B.trainable=True
    discriminator_network_A.trainable=False
    loss_d_real_B = discriminator_network_B.\.
        train_on_batch(domain_B_images, real_patch)
    loss_d_fake_B = discriminator_network_B.\.
        train_on_batch(fake_B_images, fake_patch)
    loss_d_B = np.add(loss_d_real_B, loss_d_fake_B)/2.0
    # Make the Discriminator believe that these are
    #real samples and calculate loss to train the generator
    discriminator_network_A.trainable=False

```

```

discriminator_network_B.trainable=False
# Updating Generator weights
loss_g = disco_gan.train_on_batch(
[domain_A_images, domain_B_images],\
[real_patch, real_patch, domain_B_images,\ 
domain_A_images, domain_A_images,\ 
domain_B_images])
if j%200 == 0:
    print ("Epoch:%.0f, Step:%.0f, DA-Loss:%.3f, \
DA-Acc:%.3f, DB-Loss:%.3f,\ 
DB-Acc:%.3f, G-Loss:%.3f"\ \
%(i,j,loss_d_A[0],loss_d_A[1]*100,\ 
loss_d_B[0],loss_d_B[1]*100,loss_g[0]))

```

Following are the training logs of our Disco GAN:

Output:

Epoch:0, Step:0, DA-Loss:1.987, DA-Acc:50.781, DB-Loss:1.785, DB-Acc:51.562, G-Loss:20.377

Epoch:0, Step:200, DA-Loss:0.384, DA-Acc:51.562, DB-Loss:0.481, DB-Acc:60.156, G-Loss:2.674

Epoch:0, Step:400, DA-Loss:0.342, DA-Acc:62.500, DB-Loss:0.339, DB-Acc:50.000, G-Loss:2.298

Epoch:0, Step:600, DA-Loss:0.325, DA-Acc:43.750, DB-Loss:0.296, DB-Acc:53.906, G-Loss:2.159

Epoch:0, Step:800, DA-Loss:0.289, DA-Acc:49.219, DB-Loss:0.378, DB-Acc:42.969, G-Loss:1.638

Epoch:0, Step:1000, DA-Loss:0.352, DA-Acc:43.750, DB-Loss:0.333, DB-Acc:47.656, G-Loss:2.095

Epoch:0, Step:1200, DA-Loss:0.421, DA-Acc:40.625, DB-Loss:0.297, DB-Acc:47.656, G-Loss:1.960

Epoch:0, Step:1400, DA-Loss:0.259, DA-Acc:53.125, DB-Loss:0.272, DB-Acc:59.375, G-Loss:2.479

Epoch:0, Step:1600, DA-Loss:0.282, DA-Acc:50.000, DB-Loss:0.413, DB-Acc:57.031, G-Loss:1.760

Epoch:0, Step:1800, DA-Loss:0.324, DA-Acc:42.969, DB-Loss:0.225, DB-Acc:62.500, G-Loss:1.891

.

.....

.....

.....

Epoch:45, Step:0, DA-Loss:0.286, DA-Acc:40.625, DB-Loss:0.170, DB-Acc:69.531, G-Loss:2.338

Epoch:45, Step:200, DA-Loss:0.223, DA-Acc:63.281, DB-Loss:0.021, DB-Acc:100.000, G-Loss:2.466

Epoch:45, Step:400, DA-Loss:0.207, DA-Acc:67.969, DB-Loss:0.064, DB-Acc:100.000, G-Loss:2.482

Epoch:45, Step:600, DA-Loss:0.195, DA-Acc:77.344, DB-Loss:0.203, DB-Acc:72.656, G-Loss:2.566

Epoch:45, Step:800, DA-Loss:0.134, DA-Acc:88.281, DB-Loss:0.299, DB-Acc:50.000, G-Loss:2.132

It was big model and quite slow to train. But still we had to train it for very long to get meaningful results. Let's check out some of the results.

Step 9: Results

As shown in Figure 9.8, our Disco GAN model is able to perform the gender translation without requiring paired data. If we think about it, it is not really possible to create a paired dataset for this use case, as we rarely find two humans of opposite gender with same facial features. We can see that Disco GAN is successfully able to convert Male faces into Female faces (first two rows in the figure), and vice versa (bottom two rows in the figure). It is interesting to see that the Disco GAN model is also able to introduce long hairs for Females.

Output:

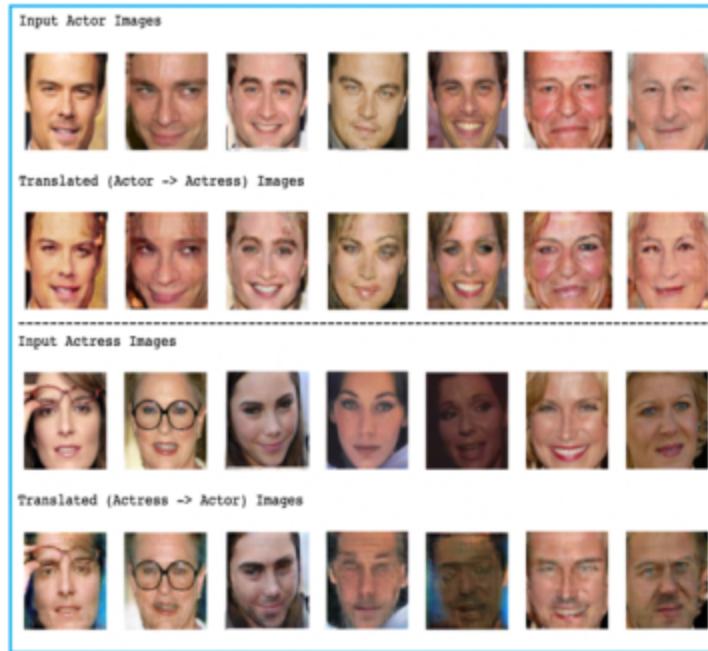


Figure 9.8: Disco GAN results for gender translation. (Top two rows) Results for ‘Male to Female’ translation. (Bottom two rows) Results for ‘Female to Male’ translation.

We just verified some impressive results of learning cross-domain relationships using Disco GAN without paired training data. Let’s now move to the next impressive application of GANs.

1. Cartoon GAN

Many applications such as printed media, storytelling for children and other forms of artwork, require cartoon like images to illustrate real world scenes. As we know that manually recreating real-world scenes in cartoon styles; involves time, patience, hard-work and artistic skills. Therefore, it’s important to develop AI techniques to solve this problem automatically. Cartoon GAN is a GAN based technique that produces high quality results for the task of photo cartoonization.

Cartoon GAN was proposed by *Yang Chen, et al.* in their 2018 paper titled “*CartoonGAN: Generative Adversarial Networks for Photo Cartoonization*”. Cartoon GAN is a GAN based approach that effectively learns the mapping from real-world photos to cartoon images using only unpaired examples.

In a Cartoon GAN setup, the generator network takes real world images as input and generates their cartoon-style versions. The discriminator network is used to validate whether a given image is a real cartoon image or a generated cartoon image. Figure 9.9 shows the cartoon GAN architecture based on the paper.

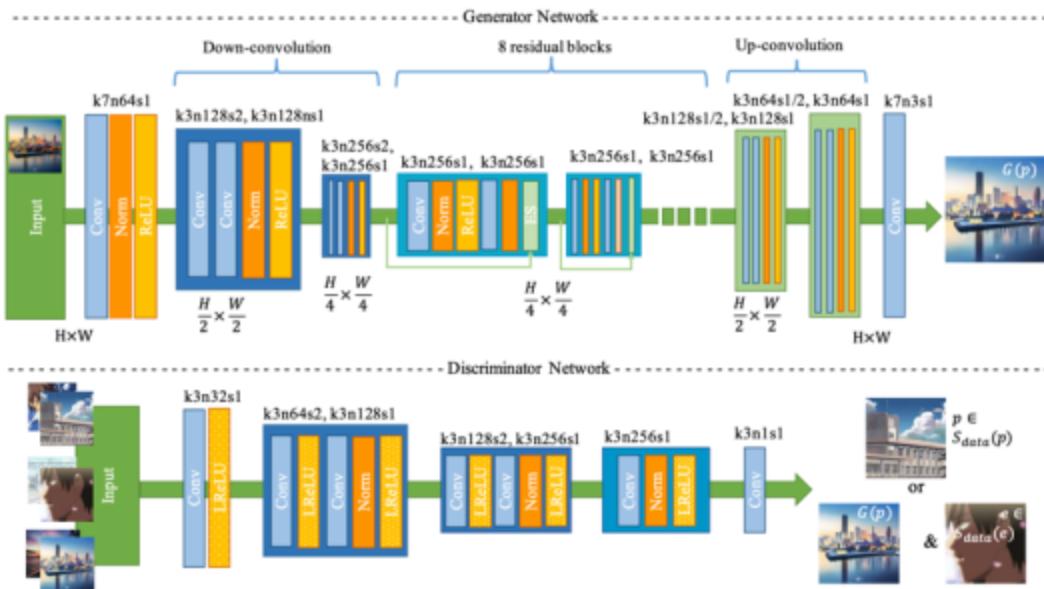


Figure 9.9: Cartoon GAN architecture as per the original research paper
Let’s learn more about the generator and the discriminator network.

3.1 Cartoon GAN Generator

Cartoon GAN generator network starts with one flat convolutional layer followed by two down-convolutional blocks to spatially compress and encode the image. Then, several identical residual blocks are stacked to construct the content and manifold features. Finally, two up-convolutional layers are used to get back original dimensions and the output image (cartoonized version). See the top architecture in Figure 9.9.

Let's look at the discriminator network architecture now.

3.2 Cartoon GAN Discriminator

The discriminator part of Cartoon GAN model, is a simple classification model as its job is just to judge whether the given image is a real cartoon or synthetic (coming from the generator). It is a shallow network and employs two strided convolutional blocks to ‘reduce the resolution and encode’ useful local features for classification. It is a Patch GAN like discriminator and uses *LeakyReLU* (with alpha=0.2) as activation function.

Let's look at the loss function now.

3.3 Cartoon-GAN Loss

Cartoon GAN loss function has two parts: an adversarial loss and a content loss. Adversarial loss helps the generator to get better at generating plausible cartoon images (very similar to the original cartoon examples). The content loss makes sure that the original content of the input image is preserved during the transformation as, we are interested in the exact cartoonization of the real-world images.

The loss function of Cartoon GAN can be written as follows:

$$\text{Cartoon GAN loss} = \text{adversarial loss} + w \times \text{content loss}$$

Here w , is a weight to balance the two losses. Large weights, encourage the model to retain more content information, whereas the lower values encourage more stylization. As per the paper, $w = 10$ provides a good balance for the content and the style.

Let's now look at some of the impressive results from Cartoon GAN paper.

3.4 Results of Cartoon GAN

Figure 9.10 shows some results from the Cartoon GAN paper. We can see that Cartoon GAN is able to create plausible cartoon images of real-world photographs in two different styles.

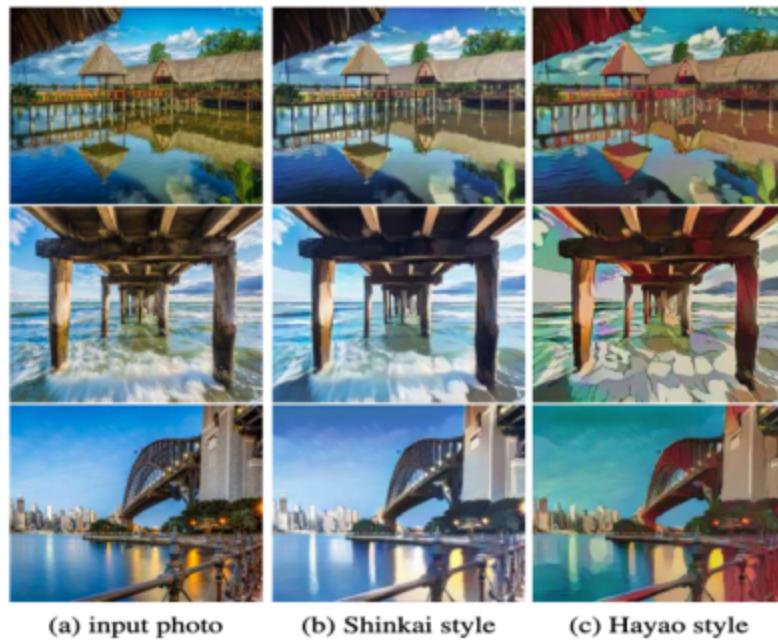


Figure 9.10: Results from original cartoon-GAN paper by Yang Chen, et al.

Let's do some experiments with Cartoon GAN and verify its results.

3.5 Experiments with Cartoon GAN

In this exercise, we will create a slightly modified version of Cartoon GAN (we can call it: custom Cartoon GAN). With our custom cartoon GAN, we will perform the following three fun experiments:

- Selfie2Sketch
- Selfie2Anime
- Selfie2Bitmoji

The code samples shown in this skill can be downloaded from the following Github location:

<https://github.com/kartikgill/The-GAN-Book/tree/main/Skill-09>

Let's get started.

CUSTOMIZED CARTOON GAN EXPERIMENTS

Objective

In this experiment, we will implement and train a customized cartoon GAN architecture and verify its results for three related tasks: *Selfie2Sketch*, *Selfie2Anime* and *Selfie2Bitmoji*.

In this implementation we will focus on *Selfie2Sketch* experiment.

This experiment has the following steps:

- Importing Libraries
- Download and Unzip Data
- Check few Samples
- Define Generator Network
- Define Discriminator Network
- Load pre-trained VGG features
- Define Selfie2Sketch-Cartoon-GAN
- Define Utility Functions
- Training Selfie2Sketch-Cartoon-GAN
- Results

Let's get started.

Step 1: Importing Libraries

Open a Jupyter Notebook with python kernel and import useful python packages in a cell. In this experiment, we will use ‘*numpy*’ for data manipulation, ‘*matplotlib*’ for plotting images and graphs, and *TensorFlow2* for implementing the model architecture later.

Checkout the following snippet for importing libraries:

```
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
from tqdm import tqdm_notebook
```

```
%matplotlib inline  
import tensorflow  
print(tensorflow.__version__)  
Output: 2.4.0
```

Let's get the dataset now.

Step 2: Download and Unzip Data

In this step, we will download and extract the dataset. For this experiment, we need a dataset that has two sets of images: selfies and cartoons. As our setup learns in an unsupervised manner, we don't need paired samples.

The following python code extracts the dataset and stores the paths of images into two training lists: one for faces and one for cartoons. Similarly, two more lists for test set images of both types:

```
!unzip /content/gdrive/MyDrive/  
GAN_datasets/selfie2cartoon.zip -d /  
import glob  
faces = glob.glob('/trainA/*.jpg')  
cartoons = glob.glob("/trainB/*.jpg")  
faces_test = glob.glob('/testA/*.jpg')  
cartoons_test = glob.glob("/testB/*.jpg")  
path = '/Users/k15/Downloads/cartoon/'  
import glob  
faces = glob.glob(path + '../selfie2anime/trainA/*.jpg')  
cartoons = glob.glob(path + "/trainB/*.png")  
faces_test = glob.glob(path + '../selfie2anime/testA/*.jpg')  
cartoons_test = glob.glob(path + "/testB/*.png")  
len(faces), len(cartoons), len(faces_test), len(cartoons_test)
```

Overall, we have 3400 face images and 194 cartoon images in the training set. We have 100 face images and 10 cartoon images in the test set.

Output: (3400, 194, 100, 10)

Let's check out some samples from the dataset.

Step 3: Check few Samples

In this step, we will plot few samples from each set of images. First, we will plot some selfie images and then, we will plot some sketches. Note that we will use an unsharp mask to sharpen the cartoon images, to make them look like sketches.

The following python code shows few samples from the dataset:

```
# Sharpen Images to make them look like sketches
def unsharp_mask(image, kernel_size=(5, 5), \
sigma=1.0, amount=3.0, threshold=0):
    """Return a sharpened version of the image,
    using an unsharp mask."""
    blurred = cv2.GaussianBlur(image, kernel_size, sigma)
    sharpened = float(amount + 1) * image - float(amount) * blurred
    sharpened = np.maximum(sharpened, np.zeros(sharpened.shape))
    sharpened = np.minimum(sharpened, 255 * np.ones(sharpened.shape))
    sharpened = sharpened.round().astype(np.uint8)
    if threshold > 0:
        low_contrast_mask = np.absolute(image - blurred) < threshold
        np.copyto(sharpened, image, where=low_contrast_mask)
    return sharpened

# Display few Samples
print ("Human Faces")
for k in range(2):
    plt.figure(figsize=(13, 13))
    for j in range(6):
        file = np.random.choice(faces)
        img = cv2.imread(file)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
```

```

img = cv2.resize(img, (128,128))
plt.subplot(660 + 1 + j)
plt.imshow(img)
plt.axis('off')
#plt.title(trainY[i])
plt.show()
print ("-"*80)
print ("cartoon Faces")
for k in range(2):
    plt.figure(figsize=(13, 13))
    for j in range(6):
        file = np.random.choice(cartoons)
        img = cv2.imread(file)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img = cv2.resize(img, (128,128))
        img = unsharp_mask(img)
        plt.subplot(660 + 1 + j)
        plt.imshow(img, cmap='gray')
        plt.axis('off')
        #plt.title(trainY[i])
        plt.show()

```

As shown in Figure 9.11, we have selfie images on the top and sketch images in the bottom two rows. Note that we have converted the cartoon images into sketches by sharpening them. These sharp sketches will act as the target domain for our customized cartoon GAN setup.

Output:

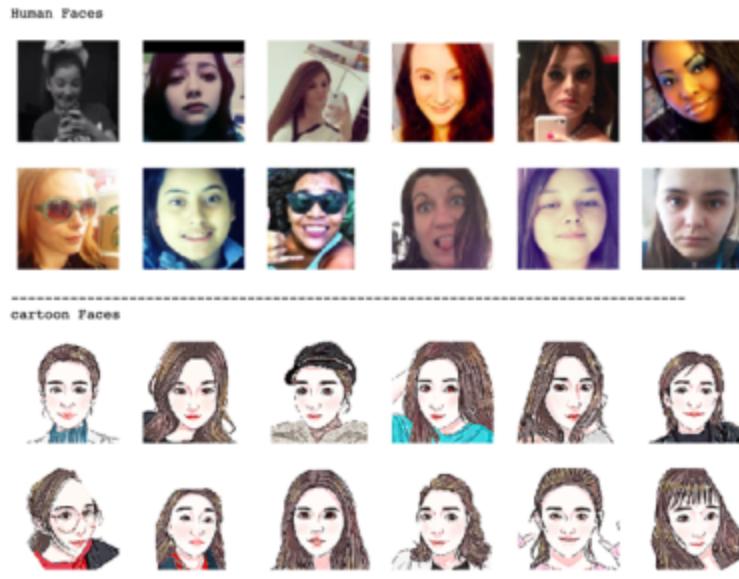


Figure 9.11: Customized Cartoon GAN dataset overview. (Top two rows) selfie images. (Bottom two rows) Sketches of faces.

Our dataset is ready now. Let's define the model architecture.

Step 4: Define Generator Network

In this step, we will define the generator network architecture for our custom Cartoon GAN setup. We will design a *UNet* like architecture for our generator network with skip connections. In our setup, the generator network accepts an image of size $128 \times 128 \times 3$ as input (a selfie image in our case) and a style input of size $16 \times 16 \times 512$ (depicting the target style). The selfie input is passed through multiple layers of strided *Conv2D* (for down-sampling), *LeakyReLU* activation with an alpha value of 0.2, and Instance Normalization. Then the resulting features are passed through a bottleneck layer that is a combination of a *Conv2D* layer and *ReLU* activation. The output of the bottleneck layer along with the outputs of previous encoder layers are passed through the decoder network. Each decoder block passes the input features through the layers of *Conv2DTranspose*, *ReLU* activation, Instance Normalization, and finally creates a *UNet* like skip connection using a concatenation layer (see the following python code).

The style input of size $16 \times 16 \times 512$ is concatenated with the output of 4th decoder block and the combined features are passed through the

remaining decoder blocks.

The final decoder layer, converts the decoded feature vector into an output image of size 128 x 128 x 3 (cartoon version), and uses a ‘*tanh*’ layer to restrict the pixel values of the translated image into a range of [-1, 1].

We will learn about the significance of the target style vector later in this experiment.

The following python code defines the generator network for our custom Cartoon GAN setup:

```
import tensorflow_addons as tfa

def encoder_layer(input_layer, filters, bn=True):
    x = tensorflow.keras.layers.Conv2D(filters, \
        kernel_size=(4,4), strides=(2,2),\
        padding='same')(input_layer)
    x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
    if bn:
        x = tfa.layers.InstanceNormalization()(x)
    return x

def decoder_layer(input_layer, skip_input, filters):
    x = tensorflow.keras.layers.Conv2DTranspose(filters,\n        kernel_size=(4,4), strides=(2,2),\n        padding='same')(input_layer)
    x = tensorflow.keras.layers.Activation('relu')(x)
    x = tfa.layers.InstanceNormalization()(x)
    x = tensorflow.keras.layers.Concatenate()([x, skip_input])
    return x

def make_generator():
    source_image = tensorflow.keras.layers.Input(shape=(128, 128, 3))
    target_style = tensorflow.keras.layers.Input(shape=(16, 16, 512))
    e1 = encoder_layer(source_image, 64, bn=False)
```

```

e2 = encoder_layer(e1, 128)
e3 = encoder_layer(e2, 256)
# e4 = encoder_layer(e3, 256)
e5 = encoder_layer(e3, 512)
e6 = encoder_layer(e5, 512)
e7 = encoder_layer(e6, 512)
bottle_neck = tensorflow.keras.layers.Conv2D(512,\n
(4,4), strides=(2,2), padding='same')(e7)
b = tensorflow.keras.layers.Activation('relu')(bottle_neck)
d1 = decoder_layer(b, e7, 512)
d2 = decoder_layer(d1, e6, 512)
d3 = decoder_layer(d2, e5, 512)
# d4 = decoder_layer(d3, e4, 256)
d5 = decoder_layer(d3, e3, 256)
d5 = tensorflow.keras.layers.Concatenate()([d5, target_style])
d6 = decoder_layer(d5, e2, 128)
d7 = decoder_layer(d6, e1, 64)
decoded = tensorflow.keras.layers.Conv2DTranspose(3,\n
kernel_size=(4,4), strides=(2,2), padding='same')(d7)
translated_image = tensorflow.keras.layers.Activation('tanh')(decoded)
return source_image, target_style, translated_image
source_image, target_style, translated_image = make_generator()
generator_network = tensorflow.keras.models.Model(\n
inputs=[source_image, target_style],\n
outputs=translated_image)
print (generator_network.summary())

```

Following is the summary of the generator network. It roughly has 43M trainable parameters.

Output:

Model: "model"

Layer (type) Output Shape Param # Connected to

input_1 (InputLayer) [(None, 128, 128, 3) 0]

conv2d (Conv2D) (None, 64, 64, 64) 3136 input_1[0][0]

leaky_re_lu (LeakyReLU) (None, 64, 64, 64) 0 conv2d[0][0]

conv2d_1 (Conv2D) (None, 32, 32, 128) 131200 leaky_re_lu[0][0]

leaky_re_lu_1 (LeakyReLU) (None, 32, 32, 128) 0 conv2d_1[0][0]

instance_normalization (Instanc (None, 32, 32, 128) 256 leaky_re_lu_1[0][0]

conv2d_2 (Conv2D) (None, 16, 16, 256) 524544 instance_normalization[0][0]

leaky_re_lu_2 (LeakyReLU) (None, 16, 16, 256) 0 conv2d_2[0][0]

instance_normalization_1 (Insta (None, 16, 16, 256) 512 leaky_re_lu_2[0][0]

conv2d_3 (Conv2D) (None, 8, 8, 512) 2097664 instance_normalization_1[0][0]

leaky_re_lu_3 (LeakyReLU) (None, 8, 8, 512) 0 conv2d_3[0][0]

instance_normalization_2 (Insta (None, 8, 8, 512) 1024 leaky_re_lu_3[0][0]

conv2d_4 (Conv2D) (None, 4, 4, 512) 4194816 instance_normalization_2[0][0]

leaky_re_lu_4 (LeakyReLU) (None, 4, 4, 512) 0 conv2d_4[0][0]

instance_normalization_3 (Insta (None, 4, 4, 512) 1024 leaky_re_lu_4[0][0]

conv2d_5 (Conv2D) (None, 2, 2, 512) 4194816 instance_normalization_3[0][0]

leaky_re_lu_5 (LeakyReLU) (None, 2, 2, 512) 0 conv2d_5[0][0]

instance_normalization_4 (Insta (None, 2, 2, 512) 1024 leaky_re_lu_5[0][0]

conv2d_6 (Conv2D) (None, 1, 1, 512) 4194816 instance_normalization_4[0][0]

activation (Activation) (None, 1, 1, 512) 0 conv2d_6[0][0]

conv2d_transpose (Conv2DTranspo (None, 2, 2, 512) 4194816 activation[0][0]

activation_1 (Activation) (None, 2, 2, 512) 0 conv2d_transpose[0][0]

instance_normalization_5 (Insta (None, 2, 2, 512) 1024 activation_1[0][0]

concatenate (Concatenate) (None, 2, 2, 1024) 0 instance_normalization_5[0][0]

instance_normalization_4[0][0]

conv2d_transpose_1 (Conv2DTrans (None, 4, 4, 512) 8389120 concatenate[0][0]

activation_2 (Activation) (None, 4, 4, 512) 0 conv2d_transpose_1[0][0]

instance_normalization_6 (Insta (None, 4, 4, 512) 1024 activation_2[0][0]

concatenate_1 (Concatenate) (None, 4, 4, 1024) 0 instance_normalization_6[0][0]

instance_normalization_3[0][0]

conv2d_transpose_2 (Conv2DTrans (None, 8, 8, 512) 8389120 concatenate_1[0][0]

activation_3 (Activation) (None, 8, 8, 512) 0 conv2d_transpose_2[0][0]

instance_normalization_7 (Insta (None, 8, 8, 512) 1024 activation_3[0][0]

concatenate_2 (Concatenate) (None, 8, 8, 1024) 0 instance_normalization_7[0][0]

instance_normalization_2[0][0]

conv2d_transpose_3 (Conv2DTrans (None, 16, 16, 256) 4194560 concatenate_2[0][0]

activation_4 (Activation) (None, 16, 16, 256) 0 conv2d_transpose_3[0][0]

instance_normalization_8 (Insta (None, 16, 16, 256) 512 activation_4[0][0]

concatenate_3 (Concatenate) (None, 16, 16, 512) 0 instance_normalization_8[0][0]

instance_normalization_1[0][0]

input_2 (InputLayer) [(None, 16, 16, 512) 0

concatenate_4 (Concatenate) (None, 16, 16, 1024) 0 concatenate_3[0][0]

input_2[0][0]

conv2d_transpose_4 (Conv2DTrans (None, 32, 32, 128) 2097280 concatenate_4[0][0]

activation_5 (Activation) (None, 32, 32, 128) 0 conv2d_transpose_4[0][0]

instance_normalization_9 (Insta (None, 32, 32, 128) 256 activation_5[0][0]

concatenate_5 (Concatenate) (None, 32, 32, 256) 0 instance_normalization_9[0][0]

instance_normalization[0][0]

conv2d_transpose_5 (Conv2DTrans (None, 64, 64, 64) 262208 concatenate_5[0][0]

activation_6 (Activation) (None, 64, 64, 64) 0 conv2d_transpose_5[0][0]

instance_normalization_10 (Inst (None, 64, 64, 64) 128 activation_6[0][0]

concatenate_6 (Concatenate) (None, 64, 64, 128) 0 instance_normalization_10[0][0]

leaky_re_lu[0][0]

conv2d_transpose_6 (Conv2DTrans (None, 128, 128, 3) 6147 concatenate_6[0][0]

activation_7 (Activation) (None, 128, 128, 3) 0 conv2d_transpose_6[0][0]

Total params: 42,882,051

Trainable params: 42,882,051

Non-trainable params: 0

None

Let's define the discriminator network now.

Step 5: Define Discriminator Network

In this step, we will define the discriminator network for our custom Cartoon GAN setup. In our setup, the discriminator network accepts an image of size 128 x 128 x 3 as input (the cartoon version). This input image is then passed through multiple layers of strided *Conv2D*, *LeakyReLU* activation with an alpha value of 0.2, and an Instance Normalization layer (except for the first layer). The final output of our discriminator network is a patch of size 8 x 8 x 1, as it follows a Patch GAN architecture. As the discriminator network is Patch GAN based, we can compile it with '*mean squared error (mse)*' loss function.

The following python snippet defines the discriminator for our custom Cartoon GAN setup and compiles it:

```
def my_conv_layer(input_layer, filters, bn=True):
    x = tensorflow.keras.layers.Conv2D(filters, \
        kernel_size=(4,4), strides=(2,2),\
        padding='same')(input_layer)
    x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
    if bn:
        x = tfa.layers.InstanceNormalization()(x)
    return x

def make_discriminator():
    target_image_input = tensorflow.keras.layers.Input(shape=(128, 128, 3))
    x = my_conv_layer(target_image_input, 64, bn=False)
    x = my_conv_layer(x, 128)
    x = my_conv_layer(x, 256)
    # x = my_conv_layer(x, 512)
    x = my_conv_layer(x, 512)
    patch_features = tensorflow.keras.layers.Conv2D(\n        1, kernel_size=(4,4), strides=(1,1), \
```

```
padding='same')(x)
return target_image_input, patch_features
```

```
target_image_input, patch_features = make_discriminator()
discriminator_network = tensorflow.keras.models.Model(\n
inputs=target_image_input,\n
outputs=patch_features)
print (discriminator_network.summary())
adam_optimizer = tensorflow.keras.optimizers.Adam(\n
learning_rate=0.0002, beta_1=0.5)
discriminator_network.compile(loss='mse', \
optimizer=adam_optimizer, metrics=['accuracy'])
```

Following is the summary of the discriminator network. It roughly has 2.8M trainable parameters.

Output:

Model: "model_1"

Layer (type) Output Shape Param #

input_3 (InputLayer) [(None, 128, 128, 3)] 0

conv2d_7 (Conv2D) (None, 64, 64, 64) 3136

leaky_re_lu_6 (LeakyReLU) (None, 64, 64, 64) 0

conv2d_8 (Conv2D) (None, 32, 32, 128) 131200

leaky_re_lu_7 (LeakyReLU) (None, 32, 32, 128) 0

instance_normalization_11 (I (None, 32, 32, 128) 256

```
conv2d_9 (Conv2D) (None, 16, 16, 256) 524544
```

```
leaky_re_lu_8 (LeakyReLU) (None, 16, 16, 256) 0
```

```
instance_normalization_12 (I (None, 16, 16, 256) 512
```

```
conv2d_10 (Conv2D) (None, 8, 8, 512) 2097664
```

```
leaky_re_lu_9 (LeakyReLU) (None, 8, 8, 512) 0
```

```
instance_normalization_13 (I (None, 8, 8, 512) 1024
```

```
conv2d_11 (Conv2D) (None, 8, 8, 1) 8193
```

```
=====
```

```
Total params: 2,766,529
```

```
Trainable params: 2,766,529
```

```
Non-trainable params: 0
```

```
None
```

Next, we will load the pre-trained VGG weights.

Step 6: Load pre-trained VGG features

In this step, we will load the pre-trained weights of a VGG19 network. We will choose the input and output layers carefully, such that it accepts an image of size 128 x 128 x 3 as input and converts it into a feature vector of size 16 x 16 x 512. This output feature vector will act as a style vector in our customized Cartoon GAN setup.

The following python code loads the VGG19 weights as discussed:

```
image_input = tensorflow.keras.layers.Input(shape=(128, 128, 3))  
pre_trained_vgg = tensorflow.keras.applications.\  
    vgg19.VGG19(weights='imagenet', \  
    input_shape=(128, 128, 3), include_top=False)  
pre_trained_vgg_model = tensorflow.keras.models.\
```

```
Model(inputs=pre_trained_vgg.input,\  
outputs=pre_trained_vgg.\  
get_layer('block4_conv4').output)  
pre_trained_image_feautures = pre_trained_vgg_model(image_input)  
custom_vgg = tensorflow.keras.models.Model(\  
inputs=image_input, \  
outputs=pre_trained_image_feautures)  
print (custom_vgg.summary())
```

Following is the summary of the loaded VGG19 architecture:

Output:

Model: "model_3"

Layer (type) Output Shape Param #

input_4 (InputLayer) [(None, 128, 128, 3)] 0

model_2 (Functional) (None, 16, 16, 512) 10585152

Total params: 10,585,152

Trainable params: 10,585,152

Non-trainable params: 0

None

Let's now define the final Cartoon GAN model.

Step 7: Define Selfie2Sketch-Cartoon-GAN

In this step, we will define the final combined architecture for our customized Cartoon GAN setup. Our final setup accepts two inputs: input source image (selfie), and target style vector (from VGG). The generator network uses these two inputs and generates a cartoon version of the source image (or sketch in our case). This cartoon version is then passed into the discriminator network to get the validity output (real vs. fake).

The final cartoon GAN setup has three outputs: validity output (from the discriminator), fake cartoon output (the generator output, to be compared with sharpened features of selfies and retain face features), another fake cartoon output (the generator output, to be compared with features of random cartoon images and retain cartoon style, or sketch style in our case).

The validity is a Patch GAN like output so we can use '*mean squared loss*' function for this one. For the other two outputs, we will write a custom content loss function that applies '*mean absolute error*' loss function over the style features of outputs and real images (style features come from the pre-trained *VGG19* model).

The following python code defines the customized Cartoon GAN setup as discussed and compiles it with aforementioned loss functions with loss weights of 1, 1, 0.1 respectively for three outputs:

```
# Customized-Cartoon-GAN-for-Selfie2Sketch
source_image = tensorflow.keras.layers.Input(shape=(128, 128, 3))
target_features = tensorflow.keras.layers.Input(shape=(16, 16, 512))
# Domain Transfer
custom_vgg.trainable=False
fake_cartoon = generator_network([source_image,\n        target_features])
discriminator_network.trainable=False
# Tell Real vs Fake
real_vs_fake = discriminator_network(fake_cartoon)
face2cartoon_gan = tensorflow.keras.models.Model(\n        inputs =[source_image, target_features],\n        outputs = [real_vs_fake,\n            fake_cartoon, fake_cartoon])
face2cartoon_gan.summary()
# Custom Content Loss (vgg features Loss)
def custom_content_loss(y_true, y_pred):
```

```
custom_vgg.trainable=False
y_true_features = custom_vgg(y_true)
y_pred_features = custom_vgg(y_pred)
content_loss = tensorflow.keras.losses.\
mean_absolute_error(y_true_features,\n
y_pred_features)
return content_loss
# Compiling Models
face2cartoon_gan.compile(loss=['mse', \
custom_content_loss, custom_content_loss],\
optimizer=adam_optimizer, loss_weights=[1, 1, 0.1])
```

Following is the summary of our combined Custom Cartoon GAN setup:

Output:

Model: "model_4"

Layer (type) Output Shape Param # Connected to

input_6 (InputLayer) [(None, 128, 128, 3) 0

input_7 (InputLayer) [(None, 16, 16, 512) 0

model (Functional) (None, 128, 128, 3) 42882051 input_6[0][0]

input_7[0][0]

```
model_1 (Functional) (None, 8, 8, 1) 2766529 model[0][0]
```

```
=====
```

```
=====
```

Total params: 45,648,580

Trainable params: 42,882,051

Non-trainable params: 2,766,529

Our Custom Cartoon GAN is all set now. Let's work on the training piece.

Step 8: Define Utility Functions

In this step, we will define some utility functions that will help us in creating data batches for training. Specifically, we will define utility functions for: converting faces to cartoons using generator network, a function to apply smoothing on images, a function for sharpening images, a function to get training samples from both domains, and finally a function to display the generator network results.

The following python code defines the utility functions for our setup:

```
def faces_to_cartoons(faces, styles, generator_network):
    styles = custom_vgg(styles)
    generated_samples = generator_network.\n        predict_on_batch([faces, styles])
    return generated_samples

def do_smoothing(img):
    # taken from https://github.com/penny4860/Keras-
    CartoonGan/blob/master/cartoon/utils.py
```

```

kernel_size = 5
kernel = np.ones((kernel_size, kernel_size), np.uint8)
gauss = cv2.getGaussianKernel(kernel_size, 0)
gauss = gauss * gauss.transpose(1, 0)
rgb_img = img
gray_img = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
pad_img = np.pad(rgb_img, ((2,2), (2,2), (0,0)), mode='reflect')
edges = cv2.Canny(gray_img, 100, 200)
dilation = cv2.dilate(edges, kernel)
gauss_img = np.copy(rgb_img)
idx = np.where(dilation != 0)
for i in range(np.sum(dilation != 0)):
    gauss_img[idx[0][i], idx[1][i], 0] = np.sum(np.multiply(pad_img[idx[0][i]:idx[0][i] + kernel_size, idx[1][i]:idx[1][i] + kernel_size, 0], gauss))
    gauss_img[idx[0][i], idx[1][i], 1] = np.sum(np.multiply(pad_img[idx[0][i]:idx[0][i] + kernel_size, idx[1][i]:idx[1][i] + kernel_size, 1], gauss))
    gauss_img[idx[0][i], idx[1][i], 2] = np.sum(np.multiply(pad_img[idx[0][i]:idx[0][i] + kernel_size, idx[1][i]:idx[1][i] + kernel_size, 2], gauss))
return gauss_img
def unsharp_mask(image, kernel_size=(5, 5), \
sigma=1.0, amount=3.0, threshold=0):
    """Return a sharpened version of the
    image, using an unsharp mask."""
blurred = cv2.GaussianBlur(image, kernel_size, sigma)
sharpened = float(amount + 1) * image - float(amount) * blurred
sharpened = np.maximum(sharpened, np.zeros(sharpened.shape))
sharpened = np.minimum(sharpened, 255 * np.ones(sharpened.shape))
sharpened = sharpened.round().astype(np.uint8)
if threshold > 0:

```

```
low_contrast_mask = np.absolute(image - blurred) < threshold
np.copyto(sharpened, image, where=low_contrast_mask)
return sharpened

def get_training_samples(batch_size):
    random_files = np.random.choice(faces, size=batch_size)
    images = []
    sharps = []
    for file in random_files:
        img = cv2.imread(file)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img = cv2.resize(img, (128, 128))
        img2 = unsharp_mask(img)
        images.append((img-127.5)/127.5)
        sharps.append((img2-127.5)/127.5)
    face_images = np.array(images)
    face_sharps = np.array(sharps)
    random_files = np.random.choice(cartoons, size=batch_size)
    images = []
    smooth_images = []
    for file in random_files:
        img = cv2.imread(file)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img = cv2.resize(img, (128, 128))
        smooth = do_smoothing(img)
        img = unsharp_mask(img)
        images.append((img-127.5)/127.5)
        smooth_images.append((smooth-127.5)/127.5)
    cartoon_images = np.array(images)
```

```
cartoon_smooth_images = np.array(smooth_images)
return face_images, face_sharps, \
cartoon_images, cartoon_smooth_images

def show_generator_results(generator_network):
images = []
styles = []
for j in range(5):
file = np.random.choice(faces_test)
img = cv2.imread(file)
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
img = cv2.resize(img, (128, 128))
images.append(img)
file = np.random.choice(cartoons_test)
img = cv2.imread(file)
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
img = cv2.resize(img, (128, 128))
img = unsharp_mask(img)
styles.append(img)
print ('Human Face Images')
plt.figure(figsize=(13, 13))
for j, img in enumerate(images):
plt.subplot(550 + 1 + j)
plt.imshow(img)
plt.axis('off')
#plt.title(trainY[i])
plt.show()
print ('Customized cartoon Version')
plt.figure(figsize=(13, 13))
```

```

for j, img in enumerate(images):
    img = (img-127.5)/127.5
    style = (styles[j]-127.5)/127.5
    output = faces_to_cartoons(np.array([img]),\
        np.array([style]), generator_network)[0]
    output = (output+1.0)/2.0
    plt.subplot(550 + 1 + j)
    #output = cv2.cvtColor(output, cv2.COLOR_RGB2GRAY)
    plt.imshow(output, cmap='gray')
    plt.axis('off')
    #plt.title(trainY[i])
    plt.show()

```

We are all set to start the training now.

Step 9: Training Selfie2Sketch-Cartoon-GAN

In this step, we will define the training iteration loop for our Custom Cartoon GAN setup. In each training step, we will first update the weights of the discriminator network. In our setup, we will update the discriminator weights with following three batches: a batch of real cartoons (or sketches), a batch of smoothed version of cartoons (or sketches), and a batch of fake cartoons (sketches from the generator output). Here, we will pass a real patch (filled with ones) of size $8 \times 8 \times 1$ as label for the real cartoon images. A fake patch (filled with zeroes) as label for fake cartoon images. And a fake patch as label for smoothed cartoon images as we don't want the generator to learn to output smooth cartoon images here, as the cartoon images are usually sharp. Secondly, we are training for sketches and they are even sharper.

Then, we will update the weights of the generator network by passing faces and a style vector (from VGG) as input and compare its outputs with a

real patch (to make the discriminator network believe that the output is real), features of sharp face images and features of cartoon images.

The following python code defines the training setup for our Custom Cartoon GAN setup:

```
epochs = 500
batch_size = 1
steps = 3400
for i in range(0, epochs):
    for j in range(steps):
        if j%100 == 0:
            show_generator_results(generator_network)
            generator_network.save("/Users/k15/Downloads/\
                cartoonDataset/weights/sharp_mod_"\
                + str(i) + "_" + str(j))
        human_faces, sharp_faces, cartoon_faces, \
        smooth_cartoon_faces = \
        get_training_samples(batch_size)
        fake_patch = np.zeros((batch_size, 8, 8, 1))
        real_patch = np.ones((batch_size, 8, 8, 1))
        custom_vgg.trainable=False
        styles = custom_vgg(cartoon_faces)
        fake_cartoon_faces = generator_network([human_faces, styles])
        # Updating Discriminator weights
        discriminator_network.trainable=True
        loss_d_real = discriminator_network.\
        train_on_batch(cartoon_faces, real_patch)
        loss_d_smooth = discriminator_network.\
        train_on_batch(smooth_cartoon_faces, fake_patch)
        loss_d_fake = discriminator_network.\
```

```

train_on_batch(fake_cartoon_faces, fake_patch)
loss_d = np.add(loss_d_real, np.add(loss_d_smooth, loss_d_fake))/3.0
# Make the Discriminator believe that these are
#real samples and calculate loss to train the generator
discriminator_network.trainable=False
# Updating Generator weights
loss_g = face2cartoon_gan.train_on_batch([human_faces,\ 
styles],[real_patch, sharp_faces, cartoon_faces])
if j%100 == 0:
    print ("Epoch:%.0f, Step:%.0f, D-Loss:%.3f, D-Acc:%.3f,\ 
G-Loss:%.3f"%(i,j,loss_d[0],loss_d[1]*100,loss_g[0]))

```

Following are the training logs:

Output:

```

Epoch:0, Step:0, D-Loss:4.169, D-Acc:51.042, G-Loss:6.552
Epoch:0, Step:100, D-Loss:0.326, D-Acc:63.542, G-Loss:2.704
Epoch:0, Step:200, D-Loss:0.272, D-Acc:64.062, G-Loss:2.579
Epoch:0, Step:300, D-Loss:0.154, D-Acc:81.250, G-Loss:2.485
Epoch:0, Step:400, D-Loss:0.106, D-Acc:88.021, G-Loss:2.899
Epoch:0, Step:500, D-Loss:0.182, D-Acc:72.396, G-Loss:2.655
. . .
. . .
. . .
. . .

```

```

Epoch:1, Step:3000, D-Loss:0.011, D-Acc:100.000, G-Loss:2.088
Epoch:1, Step:3100, D-Loss:0.011, D-Acc:100.000, G-Loss:1.971

```

As this model is pretty heavy in size, the training process is also slow. So, we only train it for a small number of steps. Let's now check out the results.

Step 10: Results

Figure 9.12 shows the results of our Custom Cartoon GAN setup for the task of selfie to sketch conversion. We can see that even after only a small number of steps of training our Custom Cartoon GAN model is able to generate plausible sketch images for selfies. The output sketches are sharp as expected and they preserve the features of actual faces.

Output:

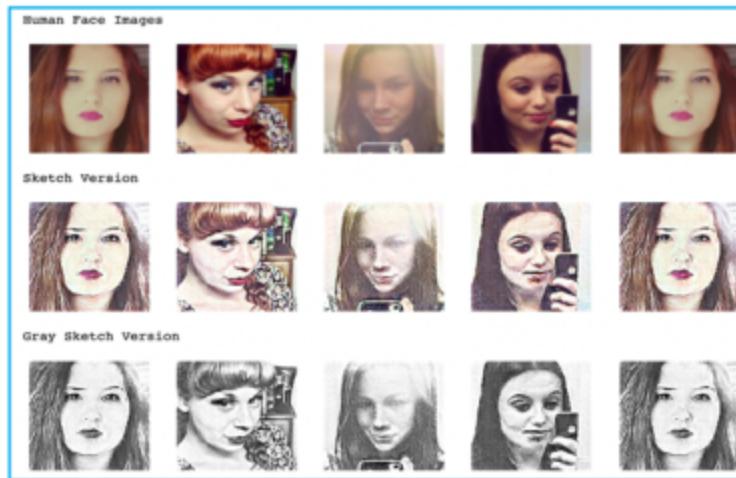


Figure 9.12: Custom Cartoon GAN results for Selfie2Sketch conversion
Let's check out some more results from our setup.

→ More results from Custom Cartoon GAN

We use our Custom Cartoon GAN setup for two more fun experiments and to verify its capabilities even further. In one of these experiments, we try to convert selfie images into their anime versions using our Custom Cartoon GAN. Figure 9.13 shows the results of *Selfie2Anime* experiment. We can see that our setup is able to convert selfie images into plausible anime versions. The results, however, are not perfect as we could not train the model to its optimality due to its big size.

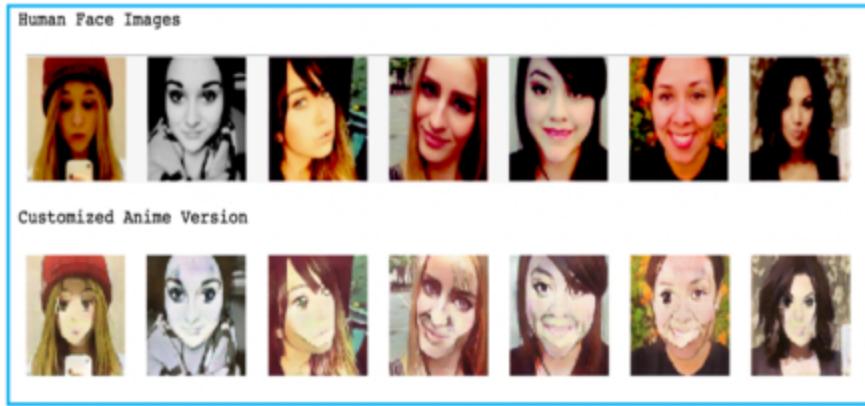


Figure 9.13: Custom Cartoon GAN results for Selfie2Anime Conversion

In our third experiment, we use our Custom Cartoon GAN setup to convert selfies into *Bitmoji* styles. Figure 9.14 shows the *Selfie2Bitmoji* results. We can see that our setup is able to learn to convert selfie images into their *Bitmoji* versions without losing important facial features.

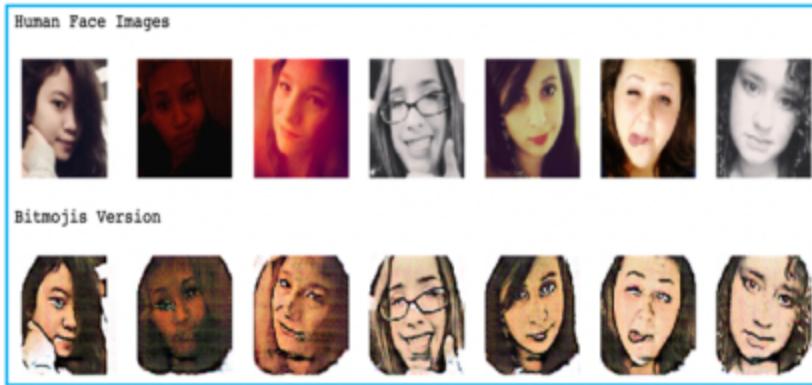


Figure 9.14: Custom Cartoon GAN results for Selfie2Bitmoji Conversion

We just saw some amazing results from a customized Cartoon GAN setup. We performed three fun experiments: *Selfie2Sketch*, *Selfie2Anime* and *Selfie2Bitmoji*. The Jupyter Notebooks for each of these experiments can be found in the Github repository of this book.

Let's now jump into one more application area of GANs for Image inpainting.

1. Image Inpainting with Context Encoders

Image Inpainting is the task of re-constructing (or filling) the missing parts in an image. It is an important computer vision problem to solve due to its multiple computer graphics applications such as object removal, image restoration, image manipulation, rendering and so on.

Context-Encoder was introduced by *Deepak Pathak, et al.* in 2016 in their research paper titled “*Context Encoders: Feature Learning by Inpainting*”. A context-encoder based model leverages Convolutional Neural Networks to predict missing parts of a scene from its surroundings.

Let’s learn more about the architecture of a Context Encoder.

4.1 Context Encoder Architecture

The overall architecture of a Context Encoder model is a simple encoder-decoder pipeline (very similar to autoencoders). Encoder part takes an input image with missing regions and produces a latent feature representation of that image. The decoder part takes this latent representation as input and produces the missing content as output.

The encoder part uses multiple convolutional layers followed by pooling layers to compute feature representations. These feature representations are then passed through a channel-wise fully connected layer before passing to the decoder model. This makes the feature maps location invariant.

The decoder part of the model takes encoder features as input and passes them through multiple up-convolutional layers with *ReLU* activation function, to generate the output image. The output image represents the missing pixels from the input image.

Let’s learn about the loss function of a Context Encoder.

4.2 Context Encoder Loss Function

As there are infinite possible ways of filling the missing regions of an image, this behavior needs to be modeled somehow. A Context Encoder makes use of a de-coupled joint loss function to handle the continuity within the context as well as for the mode selection from multiple possible modes.

The joint loss function has two parts: a reconstruction loss and an adversarial loss.

The reconstruction loss (L2) helps the model in capturing overall structure of the missing region and coherence with regards to its context. While, the adversarial loss pushes the model to generate more realistic outputs, just like in GANs, and also helps the model in picking relevant mode from multiple possible modes to fill the missing regions.

The joint loss function can be written as:

$$\text{Context Encoder joint Loss} = \text{reconstruction loss (L2)} + \text{adversarial loss}$$

Let's now look at some of the impressive results of Context Encoder for image inpainting.

1. Image Inpainting Results of Context Encoder

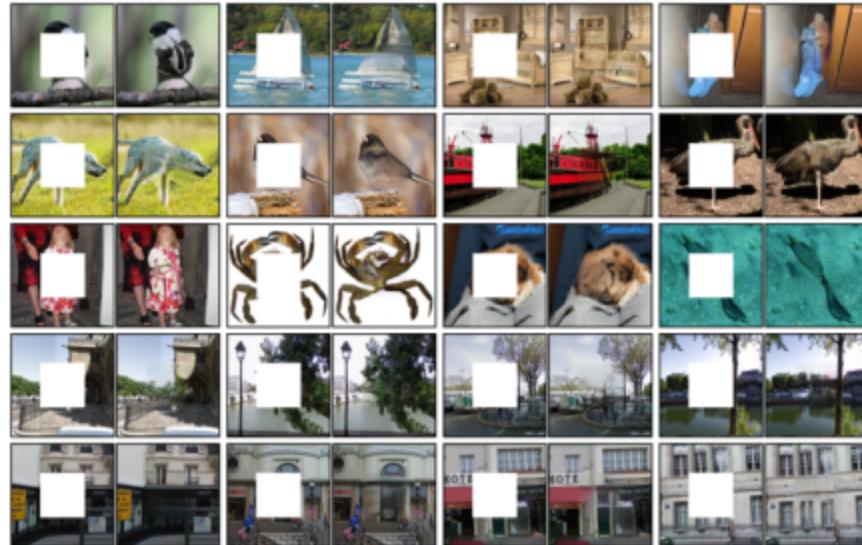


Figure 9.15: Image Inpainting results from the Context-Encoders paper by Deepak Pathak, et al.

Context Encoders achieve plausible results for the task of automatic image inpainting. Figure 9.15 shows some image inpainting results from the original paper. We can see that the model is able to fill the missing parts with creative and realistic content.

Let's now implement a Context Encoder and validate its results.

The code samples shown in this skill can be downloaded from the following Github location:

<https://github.com/kartikgill/The-GAN-Book/tree/main/Skill-09>

Let's get started.

CONTEXT ENCODER EXPERIMENT

Objective

In this experiment, we will implement and train a Context Encoder GAN for the task of image inpainting and verify its results.

This experiment has the following steps:

- Importing Libraries
- Download and Unzip Data
- Check few Samples
- Define Generator Network
- Define Discriminator Network
- Define Context Encoder
- Define Utility Functions
- Training Context Encoder
- Results

Let's get started.

Step 1: Importing Libraries

Open a Jupyter Notebook with python kernel and import useful python packages in a cell. In this experiment, we will use ‘*numpy*’ for data manipulation, ‘*matplotlib*’ for plotting images and graphs, and *TensorFlow2* for implementing the model architecture later.

Checkout the following snippet for importing libraries:

```
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
from tqdm import tqdm_notebook  
%matplotlib inline  
import tensorflow  
print(tensorflow.__version__)
```

Output: 2.4.1

Let's get the dataset now.

Step 2: Download and Unzip Data

In this step, we will extract the anime faces dataset and store the image paths in a list. This dataset has about 63k anime faces. See the following code:

```
!unzip /content/gdrive/MyDrive/\
GAN_datasets/anime.zip -d /
!ls /
import glob
input_images = glob.glob('/images/*.jpg')
print (len(input_images))
Output: 63565
```

Let's check out few data samples.

Step 3: Check few Samples

In this step, we will plot some samples from our dataset. As we are using the anime faces dataset for this experiment, we will plot a few anime face images. See the following python code:

```
print ("Images")
for k in range(2):
    plt.figure(figsize=(13, 13))
    for j in range(6):
        file = np.random.choice(input_images)
        img = cv2.imread(file)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        plt.subplot(660 + 1 + j)
        plt.imshow(img)
        plt.axis('off')
        #plt.title(trainY[i])
    plt.show()
```

Figure 9.16 shows some samples from the anime faces dataset.

Output:

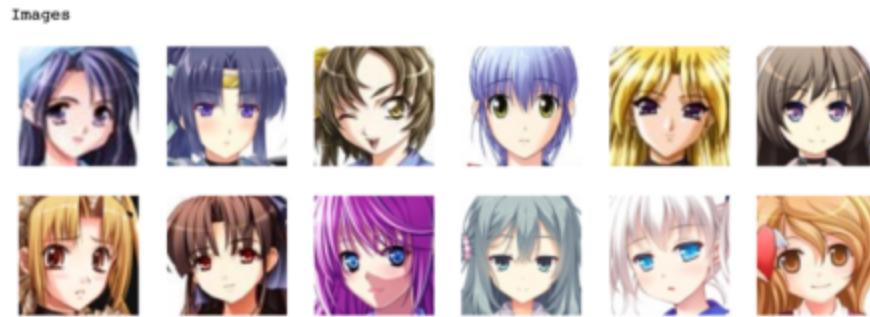


Figure 9.16: A few samples from the anime face dataset

Let's now define the model architecture.

Step 4: Define Generator Network

In this step, we will define the generator network architecture for our Context Encoder. In our setup, the generator network accepts a masked image of size 64 x 64 x 3 as input (an image with some missing part) and generates an image to fill that missing part. In our setup, the missing part has dimensions 16 x 16 x 3. The overall architecture of the generator network is an encoder-decoder kind of architecture as explained earlier in this skill.

The following python code defines the generator network:

```
def make_generator():
    masked_image = tensorflow.keras.layers.Input(\n        shape=(64, 64, 3))
    enc = masked_image
    #Encoder part
    enc = tensorflow.keras.layers.Conv2D(\n        32, kernel_size=3, strides=2,\n        padding='same')(enc)
    enc = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(enc)
    enc = tensorflow.keras.layers.BatchNormalization(\n
```

```
momentum=0.8)(enc)
enc = tensorflow.keras.layers.Conv2D(\n
64, kernel_size=3,\n
strides=2, padding='same')(enc)
enc = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(enc)
enc = tensorflow.keras.layers.BatchNormalization(\n
momentum=0.8)(enc)
enc = tensorflow.keras.layers.Conv2D(\n
128, kernel_size=3, strides=2,\n
padding='same')(enc)
enc = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(enc)
enc = tensorflow.keras.layers.BatchNormalization(\n
momentum=0.8)(enc)
enc = tensorflow.keras.layers.Conv2D(\n
512, kernel_size=1, strides=2,\n
padding='same')(enc)
enc = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(enc)
enc = tensorflow.keras.layers.Dropout(0.5)(enc)
#Decoder part
dec = enc
dec = tensorflow.keras.layers.Conv2DTranspose(\n
128, kernel_size=3, strides=2,\n
padding='same')(dec)
dec = tensorflow.keras.layers.Activation('relu')(dec)
dec = tensorflow.keras.layers.BatchNormalization(\n
momentum=0.8)(dec)
dec = tensorflow.keras.layers.Conv2DTranspose(\n
64, kernel_size=3, strides=2,\n
```

```

padding='same')(dec)
dec = tensorflow.keras.layers.Activation('relu')(dec)
dec = tensorflow.keras.layers.BatchNormalization(\n
momentum=0.8)(dec)
dec = tensorflow.keras.layers.Conv2D(\n
3, kernel_size=3,\n
padding='same')(dec)
missing_image = tensorflow.keras.layers.\n
Activation('tanh')(dec)
return masked_image, missing_image
masked_image, missing_image = make_generator()
generator_network = tensorflow.keras.models.Model(\n
inputs=masked_image, outputs=missing_image)
generator_network.summary()

```

Following is the summary of the generator network. It roughly has 825k trainable parameters.

Output:

Model: "model"

Layer (type)	Output Shape	Param #
<hr/>		
input_1 (InputLayer)	[(None, 64, 64, 3)]	0
<hr/>		
conv2d (Conv2D)	(None, 32, 32, 32)	896
<hr/>		
leaky_re_lu (LeakyReLU)	(None, 32, 32, 32)	0
<hr/>		
batch_normalization (BatchNo)	(None, 32, 32, 32)	128
<hr/>		
conv2d_1 (Conv2D)	(None, 16, 16, 64)	18496
<hr/>		
leaky_re_lu_1 (LeakyReLU)	(None, 16, 16, 64)	0

batch_normalization_1 (Batch (None, 16, 16, 64) 256

conv2d_2 (Conv2D) (None, 8, 8, 128) 73856

leaky_re_lu_2 (LeakyReLU) (None, 8, 8, 128) 0

batch_normalization_2 (Batch (None, 8, 8, 128) 512

conv2d_3 (Conv2D) (None, 4, 4, 512) 66048

leaky_re_lu_3 (LeakyReLU) (None, 4, 4, 512) 0

dropout (Dropout) (None, 4, 4, 512) 0

conv2d_transpose (Conv2DTran (None, 8, 8, 128) 589952

activation (Activation) (None, 8, 8, 128) 0

batch_normalization_3 (Batch (None, 8, 8, 128) 512

conv2d_transpose_1 (Conv2DTr (None, 16, 16, 64) 73792

activation_1 (Activation) (None, 16, 16, 64) 0

batch_normalization_4 (Batch (None, 16, 16, 64) 256

conv2d_4 (Conv2D) (None, 16, 16, 3) 1731

activation_2 (Activation) (None, 16, 16, 3) 0

=====

Total params: 826,435

Trainable params: 825,603

Non-trainable params: 832

Let's now define the discriminator network.

Step 5: Define Discriminator Network

In this step, we will define the discriminator network for our Context Encoder GAN setup. The discriminator network is quite simple. In our setup, the discriminator network accepts an image of size 16 x 16 x 3 (the missing part of the masked image), as input and passes it through multiple layers of strided *Conv2D*, *LeakyReLU* activation with an alpha value of 0.2 and Batch Normalization with a momentum of 0.8. The final layer has a ‘*sigmoid*’ activation function to generate a probability value indicating the validity of the input image (real or fake). This discriminator design is very similar to the DCGAN’s discriminator network. We can compile the discriminator network with ‘*binary_crossentropy*’ loss function.

The following python code defines the discriminator network and compiles it:

```
def make_discriminator():
    input_image = tensorflow.keras.layers.Input(shape=(16, 16, 3))
    x = input_image
    x = tensorflow.keras.layers.Conv2D(\n        64, kernel_size=3, strides=2,\n        padding='same')(x)
    x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
    x = tensorflow.keras.layers.BatchNormalization(\n        momentum=0.8)(x)
    x = tensorflow.keras.layers.Conv2D(\n        128, kernel_size=3, strides=2,\n        padding='same')(x)
    x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
    x = tensorflow.keras.layers.BatchNormalization(\n        momentum=0.8)(x)
    x = tensorflow.keras.layers.Conv2D(\n        256, kernel_size=3,\n        padding='same')(x)
    x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
    x = tensorflow.keras.layers.BatchNormalization(\n        momentum=0.8)(x)
    x = tensorflow.keras.layers.Conv2D(\n        1, kernel_size=3,\n        padding='same')(x)
    x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
    return tensorflow.keras.Model(inputs=input_image, outputs=x)
```

```

padding='same')(x)
x = tensorflow.keras.layers.LeakyReLU(alpha=0.2)(x)
x = tensorflow.keras.layers.BatchNormalization(\n
momentum=0.8)(x)
x = tensorflow.keras.layers.Flatten()(x)
real_vs_fake = tensorflow.keras.layers.Dense(\n
1, activation='sigmoid')(x)
return input_image, real_vs_fake
input_image, real_vs_fake = make_discriminator()
discriminator_network = tensorflow.keras.models.\n
Model(inputs=input_image, outputs=real_vs_fake)
discriminator_network.summary()
adam_optimizer = tensorflow.keras.optimizers.Adam(\n
learning_rate=0.0002, beta_1=0.5)
discriminator_network.compile(loss='binary_crossentropy',\n
optimizer=adam_optimizer, metrics=['accuracy'])

```

Following is the summary of the discriminator network. It roughly has 375k trainable parameters:

Output:

Model: "model_1"

Layer (type)	Output Shape	Param #
--------------	--------------	---------

input_2 (InputLayer)	[(None, 16, 16, 3)]	0
----------------------	---------------------	---

conv2d_5 (Conv2D)	(None, 8, 8, 64)	1792
-------------------	------------------	------

leaky_re_lu_4 (LeakyReLU)	(None, 8, 8, 64)	0
---------------------------	------------------	---

batch_normalization_5 (Batch	(None, 8, 8, 64)	256
------------------------------	------------------	-----

```
conv2d_6 (Conv2D) (None, 4, 4, 128) 73856
```

```
leaky_re_lu_5 (LeakyReLU) (None, 4, 4, 128) 0
```

```
batch_normalization_6 (Batch (None, 4, 4, 128) 512
```

```
conv2d_7 (Conv2D) (None, 4, 4, 256) 295168
```

```
leaky_re_lu_6 (LeakyReLU) (None, 4, 4, 256) 0
```

```
batch_normalization_7 (Batch (None, 4, 4, 256) 1024
```

```
flatten (Flatten) (None, 4096) 0
```

```
dense (Dense) (None, 1) 4097
```

```
=====
```

```
Total params: 376,705
```

```
Trainable params: 375,809
```

```
Non-trainable params: 896
```

Let's define the combined model now.

Step 6: Define Context Encoder

In this step, we will define the combined Context Encoder setup. The combined model accepts a masked image as input and passes it through the generator network. The generator network generates the missing part as output which is further passed to the discriminator network for validation (real or fake). The combined setup has two outputs: the generator output for missing part (for content loss), the validity output from the discriminator network (for adversarial loss).

The discriminator weights are kept frozen for this combined Context Encoder setup. As the idea of this combined model is to update the generator weights only. The gradient flows through the frozen discriminator network first and then updates the weights of the generator network.

The following python code defines the Context Encoder GAN setup:

```
discriminator_network.trainable=False
g_output = generator_network(masked_image)
d_output = discriminator_network(g_output)
context_encoder = tensorflow.keras.models.Model(
    inputs = masked_image, \
    outputs = [g_output, d_output])
context_encoder.summary()
```

Following is the summary of the combined model:

Output:

Model: "model_2"

Layer (type) Output Shape Param #

input_1 (InputLayer) [(None, 64, 64, 3)] 0

model (Functional) (None, 16, 16, 3) 826435

model_1 (Functional) (None, 1) 376705

Total params: 1,203,140

Trainable params: 825,603

Non-trainable params: 377,537

Our Context Encoder is ready. Let's setup the training piece.

Step 7: Define Utility Functions

In this step, we will define some utility functions that will help us in creating data for the training. Specifically, we will define utility functions for: generating masked input images, getting training batches of masked images and their missing parts, and a function for plotting the generator results.

The following python code defines the utility functions for our setup:

```
def get_masked_images(images, mask_size=16):
```

```

y1 = np.random.randint(0, images[0].shape[0] - mask_size, len(images))
y2 = y1 + mask_size
x1 = np.random.randint(0, images[0].shape[0] - mask_size, len(images))
x2 = x1 + mask_size
masked_imgs = np.empty_like(images)
missing_parts = np.empty((len(images), mask_size, mask_size, 3))
for i, img in enumerate(images):
    masked_img = img.copy()
    _y1, _y2, _x1, _x2 = y1[i], y2[i], x1[i], x2[i]
    missing_parts[i] = masked_img[_y1:_y2, _x1:_x2, :].copy()
    masked_img[_y1:_y2, _x1:_x2, :] = 0
    masked_imgs[i] = masked_img
return masked_imgs, missing_parts, (y1, y2, x1, x2)

def get_training_samples(batch_size):
    random_files = np.random.choice(input_images, size=batch_size)
    images = []
    for file in random_files:
        img = cv2.imread(file)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img = cv2.resize(img, (64, 64))
        images.append((img-127.5)/127.5)
    images = np.array(images)
    masked_imgs, missing_parts, _ = get_masked_images(images)
    return masked_imgs, missing_parts

def show_generator_results(generator_network):
    images = []
    for j in range(5):
        file = np.random.choice(input_images)

```

```
img = cv2.imread(file)
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
img = cv2.resize(img, (64, 64))
images.append((img-127.5)/127.5)
masked_imgs, missing_parts, (y1s, y2s, x1s, x2s) \
= get_masked_images(images)
predicted_missing_parts = generator_network.\
predict_on_batch(masked_imgs)
print ('Masked Images')
plt.figure(figsize=(13, 13))
for j, img in enumerate(masked_imgs):
    plt.subplot(550 + 1 + j)
    plt.imshow((img+1.0)/2.0)
    plt.axis('off')
plt.show()
print ('Completed by Context Encoder')
plt.figure(figsize=(13, 13))
for j, img in enumerate(images):
    y1, y2, x1, x2 = y1s[j], y2s[j], x1s[j], x2s[j]
    filled_image = images[j].copy()
    filled_image[y1:y2,x1:x2,:] = predicted_missing_parts[j]
    plt.subplot(550 + 1 + j)
    plt.imshow((filled_image+1.0)/2.0)
    plt.axis('off')
    #plt.title(trainY[i])
    plt.show()
print ('Original Images')
plt.figure(figsize=(13, 13))
```

```
for j, img in enumerate(images):  
    plt.subplot(550 + 1 + j)  
    plt.imshow((img+1.0)/2.0)  
    plt.axis('off')  
    plt.show()
```

We are now all set to start the training.

Step 8: Training Context Encoder

In this step, we will write the training iteration loop for our Context Encoder GAN setup. In each iteration, we will first update the weights of the discriminator network using two separate batches of: real missing parts with label 1 and, the generated missing parts with label 0. Then, we freeze the weights of the discriminator network and update the weights of the generator network using a batch of masked image inputs and an inverted validity label of 1 as output (to make the discriminator believe that these are real samples and calculate the loss for the generator). The real missing part is also passed as one of the outputs as it helps in calculating the content loss for the missing part.

The following python code defines the training setup for our Context Encoder GAN:

```
epochs = 500  
batch_size = 64  
steps = 3400  
for i in range(0, epochs):  
    for j in range(steps):  
        if j%200 == 0:  
            show_generator_results(generator_network)  
            masked_imgs, missing_parts = get_training_samples(batch_size)  
            fake = np.zeros((batch_size, 1))  
            real = np.ones((batch_size, 1))  
            fake_missing_parts = generator_network(masked_imgs)
```

```

# Updating Discriminator weights
discriminator_network.trainable=True
loss_d_real = discriminator_network.\ 
train_on_batch(missing_parts, real)
loss_d_fake = discriminator_network.\ 
train_on_batch(fake_missing_parts, fake)
loss_d = np.add(loss_d_real, loss_d_fake)/2.0
# Make the Discriminator believe that these are
#real samples and calculate loss to train the generator
discriminator_network.trainable=False

# Updating Generator weights
loss_g = context_encoder.train_on_batch(\ 
masked_imgs,[missing_parts, real])
if j%100 == 0:
    print ("Epoch:%.0f, Step:%.0f, D-Loss:%.3f, D-Acc:%.3f,\ 
G-Loss:%.3f"%(i,j,loss_d[0],loss_d[1]*100,loss_g[0]))

```

Let's now check out the results.

Step 9: Results

Figure 9.17 shows the output results of our Context Encoder GAN for the task of image inpainting. Top row shows the masked input images. Middle row represents the completed version of masked images using the Context Encoder. The bottom row shows the original unmasked images. We can see that our model does a very good job at filling the missing parts of the masked images. It is interesting to see that the model fills the missing areas of hairs with the same colors.

Output:

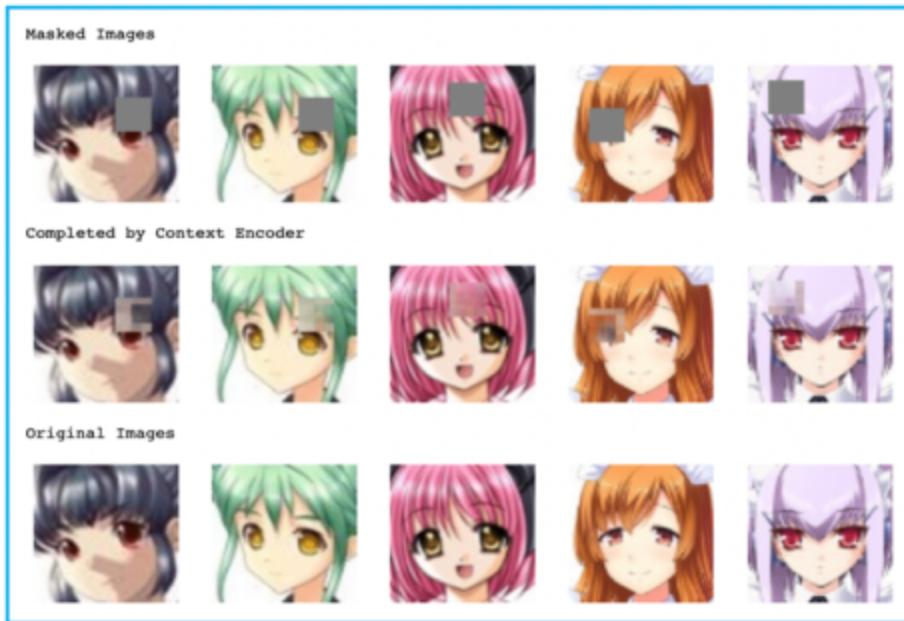


Figure 9.17: Context Encoder results for image inpainting of anime faces

We have just implemented, trained and verified the results a Context Encoder GAN for the task of image inpainting.

1. Conclusion

In this skill, we have learned about four new variants of GANs and their application areas. Specifically, we have learned: Super resolution GAN or SRGAN, Disco GAN, Cartoon GAN and Context Encoder. We understood the architecture and application areas of each of these variants in details. We have also implemented, trained and verified results of each of these variants for our custom datasets.

End of Skill-09

SKILL 9: PART II

Advanced Scaling of GANs

Generative Adversarial Networks, or GANs for short, are the most effective generative models for image synthesis. However, they are usually restricted to generating only low-resolution images (e.g., 100 x 100, 64 x 64) and training them still remains challenging as they require specific settings or hyperparameters for achieving good results. Due to these challenges, the scaling-up of GANs for generating high-resolution images is not straight forward and it requires some advanced techniques and guidelines.

Researchers have done many studies regarding the scaling of GANs. In this skill, we will learn about three popular techniques of scaling-up GANs and generating high-resolution images in high quality.

Following are the three popular scaling techniques for GANs that we will cover in this skill:

- BigGAN for Generating High-Resolution Images
- Progressive Growing of Generative Adversarial Network
- StyleGAN for Generating Photorealistic Human Faces

Let's get started.

1. BigGAN for Generating High-Resolution Images

BigGAN is a type of Generative Adversarial Network that was designed for scaling the capabilities of GAN generators for generating high-resolution and high-fidelity images.

BigGAN was introduced by *Andrew Brock, et al.* in 2019 in their paper titled “*Large Scale GAN Training for High Fidelity Natural Image Synthesis*”. BigGAN is a class-conditioned GAN, means it also takes class information as input along with the latent space to generate class-conditioned images. This paper proposed a number of incremental changes and

innovations that help in boosting the overall performance of GANs for generating high-resolution and high-fidelity images.

As the name suggests, BigGAN is focused on scaling up the GANs and it introduces the following four main changes to the architecture of traditional GANs for scaling purposes:

- Self-Attention based Base Model
- Increased Model Capacity
- Bigger Batch Size
- Some other important architectural changes

Let's get into the details of each of these.

1.1 Base Model

BigGAN uses Self-Attention GAN, or SAGAN for short, as the base model for their experimentations and study. SAGAN was proposed by *Han Zhang, et. al.* in 2018 in their paper titled '*Self-Attention Generative Adversarial Networks*'. SAGAN basically adds an attention module on top of the deep convolutional GAN or DCGAN and uses hinge loss for training.

Using SAGAN architecture as the base model, BigGAN applies some structure and training modifications that help in scaling-up the framework for generating high-resolution and high-quality outputs.

1.2 Model Capacity

The BigGAN experiments show that increasing the capacity of networks, improves the overall quality of the results (with respect to the complexity of the dataset). First, the authors increase the capacity of both models by increasing the number of channels in each layer (or increasing the width of the network). This change results in a very large number of training parameters but also improves the quality of results significantly.

Secondly, the authors try to increase the depth of the model (increasing number of layers) but this change doesn't give much improvements initially. To make the deeper models more effective, the authors applied some changes in the residual structure of blocks, in a BigGAN-deep model. We will learn about these changes soon.

1.3 Batch Size

The experiments and study from the BigGAN paper show that GANs benefit greatly from the bigger batches of data during training. This makes sense because both models are able to look at more variety (or modes) of data at each pass for updating the parameters.

Due to the larger batch sizes, training becomes faster and the results become significantly better. One side effect of the drastic scaling of batch size is that sometimes, the training becomes unstable and goes for a complete collapse. The authors also discuss the causes and the ramifications of this issue in details, within their paper.

1.4 Some other important architectural changes

Although, increasing the batch size and model capacities drastically, were the two main changes proposed by the BigGAN paper, they also presented a few more structural changes to improve the BigGAN model further.

Following is a list of some other structural changes described in the original BigGAN paper:

- **Class-Conditioned Information:** The BigGAN generator model uses class information via class-conditioned batch normalization. In order to reduce the number of weights, BigGAN utilizes a shared embedding for all class labels instead of specific label-embeddings as input.
- **Spectral Normalization:** This technique normalizes the spectral norm of the weight matrix of the generator network in BigGAN setup. Spectral Normalization technique for GANs training was first applied by *Takeru Miyato, et. al.* in their 2018 paper titled '*Spectral Normalization for Generative Adversarial Networks*'. This paper also shares the efficient way to implement it for mini-batch trainings.
- **Discriminator Updates:** BigGAN updates the discriminator network twice for each update to the generator network, during training process.
- **Weight Initialization:** Network weights are initialized using orthogonal weight initialization technique. In this method, the initial weights of the

networks are set to a random orthogonal matrix. *TensorFlow* library also supports orthogonal weight initialization for model trainings.

- **Skip-z connections:** BigGAN utilizes skip-connections to connect the latent input to not just the first layer, but also to some other intermediate layers, such that the latent space could also influence the features at different resolutions.
- **Moving Average of Generator Weights:** Before evaluation the generator network, its parameter weights are replaced with a moving average from the prior training iterations.
- **Regularization:** BigGAN uses an orthogonal regularization technique to encourage the network weights to maintain the orthogonal nature. Note that these weights were also initialized using the orthogonal weight initialization technique.
- **Truncation Trick:** This trick uses different distributions of latent space during training and inference. A gaussian latent space is used for training and a truncated gaussian distributed latent space is used at the inference time. This hack is known as the truncation trick and it leads to the improved quality of the output images but at the cost of sample variety.

We now have a good idea of the common tricks used by BigGAN for scaling up the framework for higher resolution and higher quality of image generation. Let's look at some of the impressive results from the original BigGAN paper.

1.5 Results of BigGAN

Figure 9(II).1 shows the BigGAN results at 512 x 512 resolution as shown in the paper. These results look very realistic, high quality and also have a good resolution. We can see that the BigGAN results are significant improvements over the regular DCGANs.



Figure 9(II).1: BigGAN results taken from the original paper by Andrew Brock et. al.

Now that we have a good idea on how BigGAN scales up the base GAN models to produce high quality and high-resolution outputs. Let's look into one more technique for scaling up the GANs for image synthesis: PGGAN.

1. Progressive Growing of Generative Adversarial Network

Progressive Growing of Generative Adversarial Network, also known as PGGAN, introduces a new way of scaling GANs for high-resolution image synthesis.

PGGAN was introduced in 2017 by *Tero Karras et. al.* in their paper titled '*Progressive Growing of GANs for Improved Quality, Stability and Variation*'. PGGAN was able to produce very realistic looking face images. Let's learn more about the methodology.

PGGAN introduces an incremental training approach; where it starts by generating low-resolution images and progressively increases the resolution of the output images by adding layers to the networks during the training

process. This methodology lets the networks, first learn the output distribution and progressively it starts improving the quality of the output images by learning more fine-grained details about them, instead of having to learn all the complex details in a single go.

To achieve incremental training process, PGGAN keeps the things simple by making the generator and the discriminator networks mirror images of each other. During the training process, new layers are added to both the networks, keep all the existing layers trainable. The progressive growth of networks is illustrated in Figure 9(II).2, as per the PGGAN paper.

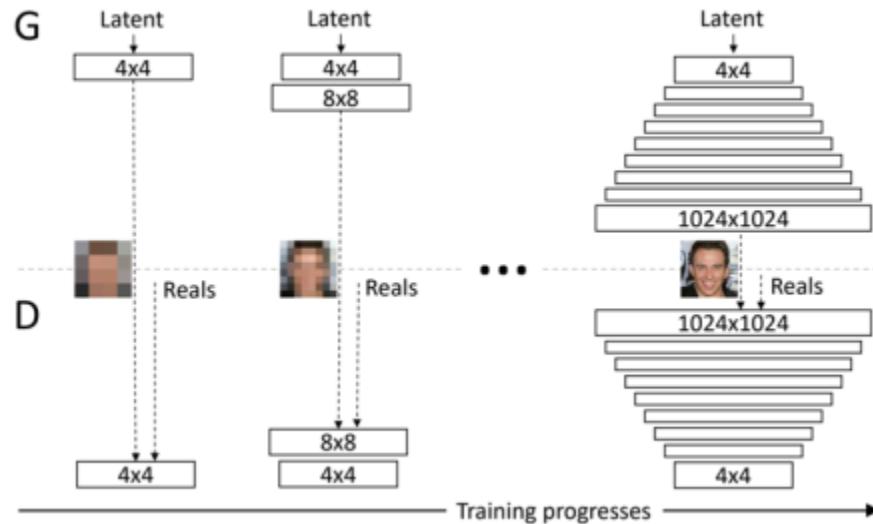


Figure 9(II).2: Progressive Training approach of PGGAN, as per the original paper

The PGGAN paper highlights several benefits of the progressive training approach. One key benefit of this method is that: we start by generating very low-resolution images to make the life of PGGAN simples, and we keep adding layers one at time, asking simpler questions to the network. This way the PGGAN model progressively learns to generate high-resolution images, one step at a time, instead of generating high-resolution images directly from the latent vectors in a single go.

Now that we understand the key concepts behind the PGGANs training methodology, let's look at some of the impressive results generated by them.

2.1 Results of PGGAN

PGGAN was applied on celebrity faces dataset (CelebA-HQ dataset) to check its ability of generating high resolution human faces. Figure 9(II).3, shows the



Figure 9(II).3: High Quality face generation output of PGGAN, taken from the original paper

PGGAN results for generating face images. These face images are 1024 x 1024 in size and look massively realistic. These results prove the benefit and potential of progressive growth of the generator and the discriminator networks during the training process. *CelebA* dataset, with high quality celebrity face images, was also contributed by the authors of PGGAN Paper.

Now we know two scaling mechanisms for making GANs output high-resolution and more realistic images: the BigGAN and the PGGAN. *StyleGAN* is another approach that also takes the scaling of GANs to the next level. Let's now learn about the key concepts behind *StyleGAN*.

1. StyleGAN for Generating Photorealistic Human Faces

Style-based Generative Adversarial Network, or StyleGAN for short, is a variant of GAN framework that produces high quality results for human face generation tasks. It also provides the control over the styles of the generated images using something called style vectors.

StyleGAN was introduced by *Tero Karras et. al.* in their 2019 paper titled '*A Style-Based Generator Architecture for Generative Adversarial Networks*'. In this paper, the authors have introduced some novel ideas for designing the generator network such that, it doesn't only produce highly realistic images but also provides some control over the styles of the generated images. Some of these innovations were inspired from style-transfer literature and hence the name: StyleGAN.

Most of the studies, prior to StyleGAN, for improving GANs had focused on creating a better discriminator network, which makes sense, because the discriminator evaluates the quality of the outputs generated by the generator network, against the real images. The intuition was that a good discriminator would provide good signals to the generator for improving the image generation quality. This totally makes sense; however, in the process, the generator network was mostly neglected and considered as a black-box.

The limited understanding of the generator network also resulted in the lack of control over the generated images. StyleGAN paper shows that GAN results can be controlled to some extent by adding controls to the generator network. Let's understand how StyleGAN controls the styles of the generated images.

StyleGAN uses the Progressive Growing of GAN, or PGGAN, as the base network for further refinement. As discussed in the last section, PGGAN is a framework for scaling GANs by incrementally adding layers to both networks during training. In addition to the progressive training similar to PGGAN, the StyleGAN makes significant changes to the architecture of the generator network.

Unlike the traditional GANs, the StyleGAN generator does not take a random latent space as input in the first layer. Whereas, it works with a constant latent vector (which is also learned during the initial training) as input to the first layer. Now comes the question that where does it get the randomness from? So, basically StyleGAN introduces two new ways of

passing randomness to the generator network: a mapping network and noise layers.

3.1 Mapping Network

StyleGAN generator uses a standalone mapping network (a fully connected non-linear network) which outputs a style vector. This style vector is then integrated into the StyleGAN generator network at multiple places through a new layer called Adaptive Instance Normalization, or *AdaIN*. The use of this style vector gives control over the styles of the generated images.

3.2 Noise Layers

Some explicit noise inputs are also added at multiple places in the StyleGAN generator network to introduce the stochastic variations. These noise inputs are single-channel un-correlated gaussian noise images. This noise is broadcasted to all the features at all the input layers of the synthesis network.

This block level incorporation of the style vectors and noise, allows each block of the synthesis network to localize both the interpretation of style and the stochastic details, to a good extent.

Figure 9(II).4, shows the StyleGAN generator architecture as compared with the traditional GAN generator, taken from the original paper.

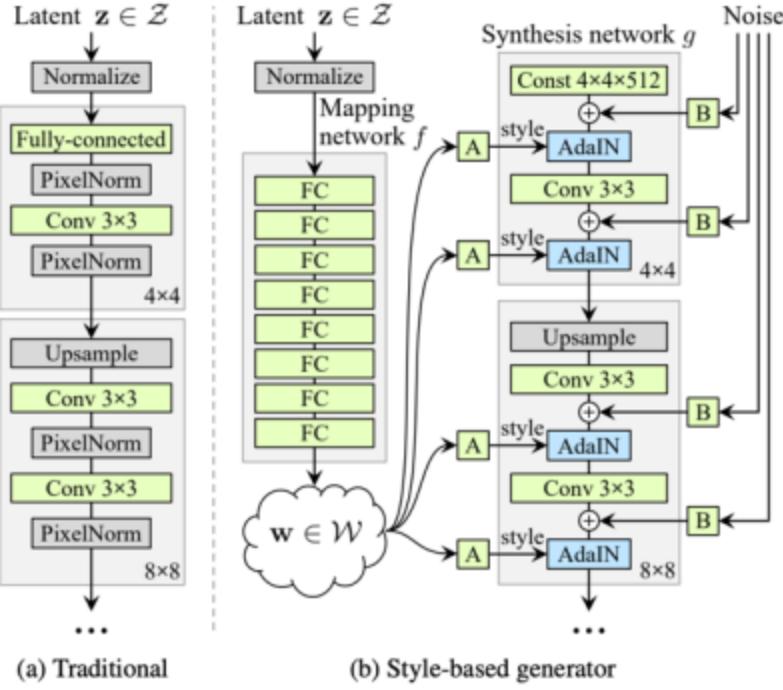


Figure 9(II).4: StyleGAN generator architecture as compared to the traditional GAN generator, taken from the StyleGAN paper by Karras et. al.

We now have some basic idea about the architectural changes that were proposed in the StyleGAN paper; especially for the generator network. Now let's go through all the changes and recommendations provided in the paper for generating high quality photorealistic human face images.

Following is a list of the major architectural changes suggested by the authors of StyleGAN paper:

- **Base Model:** StyleGAN uses the PGGAN as a base model and then incorporates the new changes into the generator network.
- **Up Sampling Layers:** Unlike the traditional GANs that utilize nearest neighbor or transpose convolutional layers for up sampling the images, the StyleGAN generator uses bilinear up sampling layers.
- **Mapping Network:** As discussed earlier, the StyleGAN generator introduces randomness into the synthesis network by passing the output

of a mapping network into each block of synthesis network, using a new layer called Adaptive Instance Normalization or *AdaIN*.

- **Latent Point Input:** Unlike the traditional GANs, the StyleGAN synthesis network uses a learned constant vector as the first input layer instead of a random latent vector.
- **Noise Images:** As discussed earlier, the StyleGAN generator uses single-channel images of noise at each block of the synthesis network, to introduce stochastic variations within all the features.
- **Mixing Regularization:** Mixing Regularization involves using two style vectors from the mapping network at the same time. To incorporate this: a split point in the synthesis network is chosen, and then all the blocks prior to it, use the first style vector and the remaining blocks use the second style vector for generating images. This type of mixing is applied over a given percentage of images during training. This encourages the blocks of the synthetic network to localize the style to a specific part of the model and the corresponding level of detail in the generated images.

We now have a good idea about the architectural changes proposed by the authors of StyleGAN paper, specifically, for the generator network. Now let's look at some of the impressive results of the StyleGAN framework from the original paper.

3.3 Results of StyleGAN

StyleGAN is capable of generating high-quality photorealistic human face images as shown in the Figure 9(II).5. These faces look highly realistic, and some other details such as glasses, hats and so on, are also looking amazing.



Figure 9(II).5: Photo-realistic human faces generated through StyleGAN, as shown in the original StyleGAN paper by Karras et. al.

Figure 9(II).5, shows the results of StyleGAN for different resolutions, starting from 4 x 4 till 32 x 32.

3.4 Style Control

The StyleGAN authors performed multiple experiments by providing modified style-vectors or noise vectors as input to the synthesis network, and noted down their effects. First experiment was done using the style vectors: in this experiment, the authors found out that passing different style vectors at different points of synthesis network, gives control over the styles of the generated images. In a StyleGAN setup, lower resolution blocks (4 x 4 and 8 x 8) in the synthesis network control high level of details in the styles such as pose, hairstyles and so on. Middle resolution blocks (16 x 16 and 32 x 32) control the medium level of details such as expressions on the face and hair styles. Finally, the blocks with higher resolutions (from 64 x 64 up to 1024 x 1024), control very low level of details in the generated face images such as skin color.

Let's look at Figure 9(II).6 to understand this better. In this figure, we have two sets of images generated using StyleGAN: Source A and Source B. First column represents the source A images and first row represents the source B images.



Figure 9(II).6: Style controlled results from StyleGAN, Intermediate images adopt high level details from left column and low level details from top rows.

Image is taken from StyleGAN paper

All the intermediate images are generated using the style vectors from both sources together. Here, the style vectors from source A are used at higher resolution blocks in synthesis network (to control low level of details) and the style vectors from source B are passed into the lower resolution blocks of synthesis network (to get high level of details). This encourages the model to adopt high-level of details from source B images and low-level small details from source A images. Thus, the high-level details such as pose, glasses, gender and so on, are preserved from top to bottom, while the low-level details such as skin color and facial expressions are preserved from left to right in the intermediate images.

3.5 Control over Details

Another level of control can be achieved by varying the noise vectors at different levels of the synthesis network. This results in a control over the details in the generated images from the synthesis network of StyleGAN. Coarse noise (noise inserted at the coarse layers of the synthesis network, or the low-resolution layers) causes appearance of larger background and large-scale curling of the hairs, while the fine noise (noise inserted at the finer layers or the high-resolution layers) brings out the finer curls of hair, skin pores and finer details in the backgrounds. Figure 9(II).7 shows some effects of varying noise vectors at different locations of the StyleGAN synthesis network.

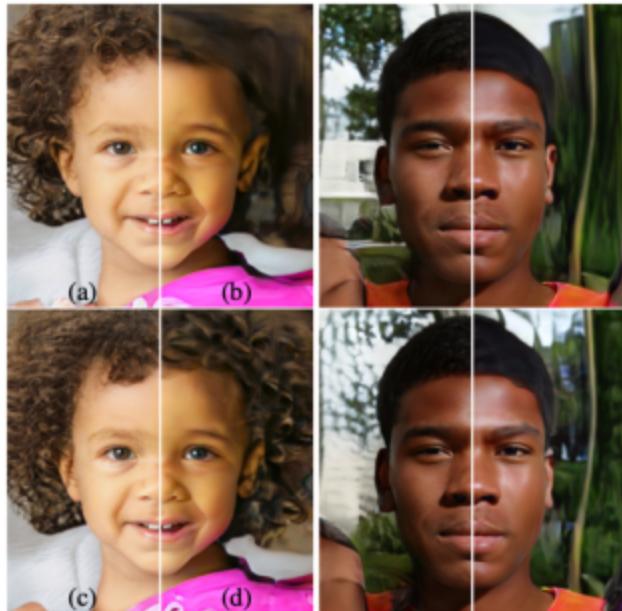


Figure 9(II).7: Effect of the noise inputs at different layers of StyleGAN generator. (a) Applied to all layers (b) No noise (c) Only in fine layers (64x64 till 1024x1024). (d) Only in coarse layers (4x4 till 32x32).

We now have a good understanding about the innovations proposed by *Karras et. al.* in their StyleGAN paper. StyleGAN is capable of generating highly realistic human faces and also provides control over the style of generated content.

1. Conclusion

In this skill, we picked up the topic of scaling Generative Adversarial Networks for generating high resolution and more realistic images. We discussed three main approaches of scaling up GANs for generating high quality images: BigGAN, PGGAN and StyleGAN. The BigGAN shows that increasing the model capacity and batch size during the training of GANs can help in generating high quality images. The PGGAN proposed a new incremental training mechanism for scaling up GANs. It generates the higher resolutions of images in a step-wise manner and achieves great success in generating photorealistic images and high-quality synthetic human faces. Finally, the StyleGAN incorporated some big changes to the generator network of PGGAN model and improved the results significantly. We saw that the StyleGAN model doesn't only generate highly photorealistic human faces but also provides some control over the styles and details of the generated images.

After reading this skill, we should now be aware about the best-practices and guidelines of scaling up GANs for generating high-quality images. With this background, we should feel confident about developing our own large-scale GANs to solve challenging real-world problems.

End of Skill-09(II)

SKILL 10

How to Evaluate GANs?

Generative Adversarial Networks are an effective way of developing deep generative models. Traditional deep learning models are trained using an objective function, also known as a loss function, until the convergence. It is often easy to assess the quality of the traditional models by looking at the loss values during training. However, the things get a bit tricky in case of GANs. A GAN generator model is trained using a second deep learning model (or the discriminator model) that learns to differentiate real data from the generated fake data. Both the generator and the discriminator models are trained simultaneously and an equilibrium is maintained. Due to the nature of their training approach, there is no objective function or empirical way to assess the quality of outputs. Moreover, there is also no obvious way to know if training is progressing in the right direction or not.

This issue led to the research and development of multiple qualitative and quantitative techniques to evaluate or compare the outputs of GAN based models. In this skill, we will learn about some popular techniques to evaluate GANs.

This skill covers the following main topics:

- Challenges of Evaluating GANs
- Manual Evaluation of GANs
- Qualitative Evaluation of GANs
- Quantitative Evaluation of GANs
- Conclusion

Let's get started.

1. Challenges of Evaluating GANs

Generative Adversarial Networks, or GANs, are a type of deep learning approach for training generative models. GAN based generative models have shown impressive results in a wide range of problem domains. As we know that the GAN generator is trained using a second deep learning model, the objective functions of both models usually measure how well they are doing relative to the opponent. In other words, we measure how well the generator model is fooling the discriminator model. However, this measure doesn't guarantee the quality of the generated data. For example, our generator network might find a tricky way to fool the discriminator while generating only the garbage data.

Due to these challenges, the researchers have devised various new techniques to evaluate the quality and diversity of the data generated from a GAN based generator model. We will learn about these techniques shortly.

1. Manual Evaluation of GANs

One way to evaluate Generative Adversarial Networks is to manually assess the quality of the generated samples. In this technique, first a trained generator model is asked to generate a batch of samples, and then the quality and diversity of these samples is evaluated manually by humans, with respect to the desired or target domain. This manual way of visually inspecting the quality of generator model is a quite common way of evaluating GANs.

The generator model is trained iteratively as per the feedbacks from a discriminative model, and due to the nature of training, there is no objective function to directly evaluate the quality of the generator model. So, it is hard to say whether the training is progressing in the right direction or not. Due to this challenge, it is very common to save a batch of the generated results after every few training iterations along with the generator weights. This lets the researchers, manually inspect the quality of the generated samples for various iterations and choose the generator weights that provide the most promising results based on the manual inspection. Thus, manually inspecting the quality of GANs is easy and practical but it also has a few limitations.

Here are some of the common limitations of manually evaluating Generative Adversarial Networks:

- This method is not very scalable, as there is always a limit on how many samples someone can manually review.
- Manually reviewing the results introduces human bias and thus, a model that looks promising to one person might not look good to another.
- To visually measure the quality of the outputs, the reviewer needs to be an expert on the target domain. Then only he/she can decide the quality of results.

With some limitations and its simplicity, the manual evaluation of GANs is still heavily adopted in many practical scenarios. Let's now look at some of the qualitative evaluation methods for GANs.

1. Qualitative Evaluation of GANs

As the name suggests, qualitative measures do not use any numerical method to evaluate the generator results. A number of qualitative measures have been proposed for GANs. These methods typically involve human perception or evaluation via comparison.

A research paper titled "*Pros and Cons of GAN Evaluation Measures - 2018*" by *Ali Borji* presents multiple qualitative and quantitative measures for evaluating the generative models.

Some common qualitative techniques of evaluating GANs are listed below. These techniques are explained in details by *Ali Borji* in his work "*Pros and Cons of GAN Evaluation Measures – 2018 and 2021*".

- Nearest Neighbors.

- Rapid Scene Categorization.
- Rating and Preference Judgment.
- Mode Drop and Collapse.
- Network Internals.
- Human Eye Perceptual Evaluation (HYPE).
- Neuro-score.
- Seeing what a GAN cannot generate.
- Measuring GAN steerability.
- GAN Dissection.
- A Universal Fake vs. Real detector.

As described earlier, all these techniques require human intervention. Nearest Neighbors method compares the generated images with their nearest neighbor from the training dataset and can detect overfitting.

In Rapid Scene Categorization method, the participants are asked to classify few real vs. fake examples in a short span of time.

In Preference Judgement method, participants are asked to rank models based on the generated results. Similarly, other methods require human intervention in different ways to assess the quality of generated content. To read more about all these methods please refer to the aforementioned research paper. Let's now look at the quantitative evaluation methods.

1. Quantitative Evaluation of GANs

Quantitative Evaluation methods generate a numerical score to assess the quality of generated content from a GAN generator. A number of quantitative evaluation techniques have been invented and applied by the researchers.

Following a list of common quantitative evaluation techniques for GANs. These techniques are explained in details by *Ali Borji* in his work “*Pros and Cons of GAN Evaluation Measures – 2018 and 2021*”.

- Average Log-Likelihood.
- Coverage Metric.
- Inception Score (IS).
- Modified Inception Score (m-IS).
- Mode Score (MS).
- AM Score.
- Fréchet Inception Distance (FID).
- Maximum Mean Discrepancy (MMD).
- The Wasserstein Critic.
- Birthday Paradox Test.
- Classifier Two Sample Test (C2ST).
- Classification Performance.
- Boundary Distortion.
- Number of Statistically-Different Bins (NDB).

- Image Retrieval Performance.
- Generative Adversarial Metric (GAM).
- Tournament Win Rate and Skill Rating.
- Normalized Relative Discriminative Score (NRDS).
- Adversarial Accuracy and Divergence.
- Geometry Score.
- Reconstruction Error.
- Image Quality Measures.
- Low-level Image Statistics.
- Precision, Recall and F₁ score.
- Spatial FID (sFID).
- Class-aware FID (CAFD) and Conditional FID.
- Fast FID.
- Memorization-informed FID (MiFID).
- Perceptual Path Length.
- Perplexity.

It's always a good idea to combine a few of these techniques to evaluate the generator model. Though all these methods have their own pros and cons, “*Inception Score (IS)*” and “*Fréchet Inception Distance (FID)*” are two widely used metrics for evaluating GAN generator performances.

To learn more about all these methods, please refer to the aforementioned paper. In this skill, we will discuss about two methods: *Inception Score (IS)* and *Fréchet Inception Distance (FID)*.

4.1 Inception Score (IS)

Inception Score was proposed by *Tim Salimans, et al.* in their 2016 paper titled “*Improved Techniques for Training GANs*”. It measures the performance of GANs based on the following two criteria:

- Quality of generated Images
- Diversity of generated Images

Inception Score uses a pre-trained deep learning image classification model for evaluation. The idea here is: if the generated images are realistic, image classification model should be able to recognize or predict objects from them with a very high confidence. It’s called Inception score because the classification model that was used for evaluation initially, was a popular Inception V3 model.

In this method, a large number of generated images (from the GAN generator) are scored using the pre-trained image classification model. These predictions are then combined to generate the inception score. This score aims to capture both the quality and the diversity of the generated images.

In this method, the quality of the generated images is measured based on the prediction confidences of the classifier. To understand the diversity, we look at the distribution of the output classes, a uniform distribution of output classes represents high diversity of the generated images.

Inception Score is quite effective but there are a few limitations as well. Following are some of those limitations:

- There is always a limit on the number of classes your image classifier is trained on. So, it will only be able to classify the known objects from the generated images. Inception v3 is trained with 1000 classes but we may not need them all, and they may not be all we need.

- It is also possible that the GAN generator is generating objects from multiple different classes but there is not much variety within a single class. In this case as well, the Inception Score will mark the model to be significantly diverse which is incorrect.
- Image classifier might require the generated images to be within specific dimensions (for example a square image of dim. – 300 x 300).

4.2 Fréchet Inception Distance (FID)

Fréchet Inception Distance or *FID* for short, also uses a pre-trained image classification model (such as Inception v3) for evaluation. This method calculates a similarity score between two groups of images (the group of real images vs. the group of the generated images). Metric FID calculates the distance between the feature vectors coming from two groups of images (real and generated).

These feature vectors are calculated from a pre-trained computer vision model such as Inception v3. The score determines whether two groups are statistically similar or not. A score of 0.0 represents two identical groups of images.

Lower scores indicate that the two groups are statistically similar and have shown to correlate well with the quality of images. If the FID score is low between the group of real images vs. the group of GAN generated images, it means that the generated images are very realistic and high quality.

FID is often considered superior to the Inception Score as it also takes the real images distribution also into account for evaluating the generator model.

1. Conclusion

In this skill, we discussed multiple evaluation techniques for Generative Adversarial Networks; based on the generated content. Following are few important points to remember from our study:

- There is no proper objective function used during the training of the generator network of a GAN and thus, the model must be evaluated based on the quality of the generated content.
- During training, manual inspection of the generated content, could be a great starting point but this is not very scalable and has some limitations.
- We understood how IS and FID scores are calculated and how they can be used to evaluate GAN generators.
- Though the best evaluation method depends upon the end goal of the project, it's probably a good idea to combine a few qualitative and a few quantitative techniques to understand the overall quality of the GAN generators in a robust way.

In the next skill, we will learn about the adversarial examples.

End of Skill-10

SKILL 11

Adversarial Examples

Adversarial examples are inputs to the machine learning (ML) models that are intentionally designed to fool the model. These examples can be created by performing intentional feature perturbation on the inputs and, as a result they can make the ML models do false predictions. Adversarial examples are often indistinguishable to human eye. It is really important to study such inputs because they make ML models vulnerable to attacks. These attacks may cause serious security violations in some scenarios. Thus, it is important to study them and make ML models robust to such adversarial attacks.

This skill covers the following topics:

- What is Adversarial Machine Learning?
- What are common types of Adversarial Attacks?
- How to defend against Adversarial Attacks?
- Why it is hard to defend against Adversarial Attacks?
- Conclusion

Let's get started.

1. What is Adversarial Machine Learning?

Adversarial Machine Learning is the study of adversarial examples and defense mechanisms against them. Adversarial examples are intentionally

generated in order to trick ML models by using deceptive inputs. Today, the ML models are extensively used in many security or safety departments and the adversarial attacks can be seriously dangerous in such scenarios. Thus, it is important to study such attacks and prepare defense mechanisms against them.

Based on the intentions of the attacker, the adversarial attacks can be classified into two categories: *the targeted attacks* and *the untargeted attacks*.

In an untargeted attack, the only intention of the attacker is to make the model do a wrong prediction. On the other hand, a targeted attack has a target class and the intention of the attacker is not just to make the model do a mistake but also to make the model generate the wrong prediction that belongs to the desired output class.

Based on the attacker's knowledge, the adversarial attacks can be classified into the following two categories: white-box attack and black-box attack.

A white-box attack is a scenario where the attacker has full access to the ML model. The attacker is aware of the model's architecture and its parameters.

In a black-box attack, the attacker only can see the output of the ML model and doesn't have any other knowledge about it.

ML has become an essential part of almost every organization and their key business processes. Many critical tools used for security and safety purposes are developed using ML algorithms today. Thus, the need to protect these ML systems is also growing fast. In the next sections, we will learn about some common methods of generating adversarial examples and defense mechanisms against them.

1. What are common methods of Adversarial Attacks?

As discussed earlier, the adversarial examples are intentionally prepared to make the ML model do a wrong prediction. These examples appear normal to

humans but they can cause misclassifications for the target ML models. Many different methods of generating the adversarial examples have been studied.

Following are some common methods of generating adversarial examples:

- Fast Gradient Sign Method (*FGSM*)
- Limited-Memory BFGS (*L-BFGS*)
- Jacobian-based Saliency Map Attack (*JSMA*)
- Deep Fool Attack
- Carlini & Wagner Attack
- Generative Adversarial Networks (*GANs*)
- Zeroth-Order Optimization Attack (*ZOO*)

Let's learn about these methods.

2.1 Fast Gradient Sign Method (*FGSM*)

Fast Gradient Sign Method, of FGSM, works by using the gradients of the neural network. Because FGSM has access to the model, it comes under the white-box attacking mechanisms. In this method, the input image pixels are perturbed using the information of the gradient of the loss function, such that the model loss is increased. This newly generated input image is called the adversarial image.

The image panda shown in Figure 11.1, is a very popular adversarial example generated using FGSM method. This image was taken from the research paper titled “*Explaining and Harnessing Adversarial Examples-2015*” by Ian J. Goodfellow and fellow researchers.

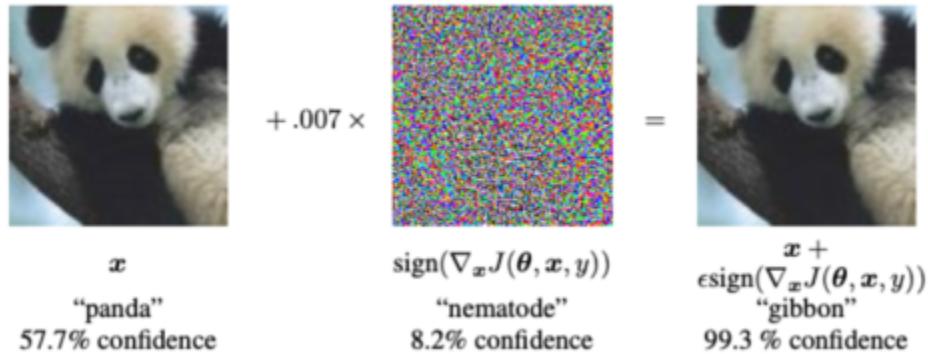


Figure 11.1: Generating an adversarial example using FGSM taken from the 2015 paper of Ian Goodfellow et.al.

Let's learn about L-BFGS now.

2.2 Limited-Memory BFGS (L-BFGS)

Limited memory BFGS, or L-BFGS, is an optimization algorithm to minimize the number of perturbations added to an image. It is a non-linear gradient-based method. It is quite effective in generating adversarial examples. But it is very computationally expensive.

Next, let's learn about JSMA.

2.3 Jacobian-based Saliency Map Attack (JSMA)

JSMA method uses feature selection method to minimize the number of feature perturbations needed to cause the model make a classification mistake. This method is also computationally intensive.

Let's now learn about deep fool attack.

2.4 Deep Fool Attack

Deep Fool attack technique works by minimizing the Euclidean distance between the perturbed inputs and the original inputs. In this method, the perturbations are added iteratively based on the decision boundaries. This method of generating adversarial examples is also computationally expensive.

Let's move to the next method.

2.5 Carlini & Wagner Attack

This method is based on the L-BFGS algorithm and is more efficient at generating the adversarial examples. This method was able to defeat certain defense mechanisms. Again, this attack is also very computationally expensive.

Let's learn about GAN based attack now.

2.6 Generative Adversarial Networks (GANs)

Generative Adversarial Networks have been used for generating adversarial examples. A GAN can be used to generate desired number of examples that are different from the training dataset. Training a GAN is also computationally intensive and highly unstable.

Let's now learn about ZOO attack.

2.7 Zeroth-Order Optimization Attack (ZOO)

ZOO attack is a black-box adversarial attack that works without even the knowledge of the ML algorithm. This method collects gradient information of the ML model, indirectly by querying the model with modified individual features. One common dis-advantage of ZOO technique is that it requires large number of queries to the underlying ML model.

Now that we know common adversarial attack methodologies. Let's learn about some defense mechanism against them.

1. How to defend against Adversarial Attacks?

Traditional optimization techniques such as weight decay, dropouts and so on, generally don't provide defense against the adversarial attacks. The paper "*Adversarial Attacks and Defences: A Survey-2018*" by *Anirban chakraborty* and team describes some common adversarial attacks and defense mechanisms.

Following are some of the common defense mechanisms against the adversarial attacks, as per their paper:

- Adversarial Training

- Defensive Distillation
- Gradient Hiding
- Feature Squeezing
- Blocking the Transferability
- Defense GAN
- Mag Net

Let's discuss these mechanisms.

3.1 Adversarial Training

The adversarial training is a simple brute-force solution to defend the ML models against the adversarial attacks. In this method, a large number of adversarial examples are generated and then the ML model is explicitly trained on them, to make sure that it is not fooled by them.

In this way, the ML models become robust to such known types of the adversarial examples. However, the attacker might still break this defense using a new attacking mechanism that the model has not been trained against. Let's move to the second technique now.

3.2 Defensive Distillation

This method is based on the concept of knowledge distillation of the Neural Networks where, a small ML model is trained to imitate a very large ML model, in order to obtain the computational savings. This small ML model, trained using the probabilistic outputs of the larger ML model, is robust to the adversarial attacks as its loss surface is smoothed in the directions that an adversary will try to exploit. This mechanism makes it hard for the attacker to tweak the adversarial examples for getting a wrong prediction from the ML model.

Let's now learn about Gradient Hiding method.

3.3 Gradient Hiding

As discussed earlier, some of the attacking mechanisms such as FGSM, rely on the gradient information from the ML model. Gradient Hiding is one way to make those attacks ineffective by using non-differentiable models such as Decision Trees, KNN, Random Forests and so on.

Let's now learn about the next mechanism.

3.4 Feature Squeezing

Feature squeezing is a method of model hardening. In this method, the ML models are made less complex by reducing or squeezing the input features. Simpler ML models are more robust to the small feature perturbations and noise. One dis-advantage of this method is that these models are often less accurate because of their simpler nature.

Let's now learn about Blocking and Transferability method.

3.5 Blocking and Transferability

The main reason behind the defeat of most of the well-known defense mechanisms; is due to the strong transferability property of the neural networks. For example, the adversarial examples generated on one classifier are expected to cause another classifier to perform the same mistake. The transferability property holds true even if the classifiers have different architectures or have been trained on disjoint datasets.

Hence, the key for protecting against a black-box attack is to block the transferability of the adversarial examples. Let's now learn about the next mechanism.

3.6 Defense-GAN

The Defense GAN mechanism leverages the power of Generative Adversarial Networks to reduce the efficiency of the adversarial perturbations. The central idea of this method is to *project* the input images onto the range of the generator by minimizing the reconstruction error, prior to feeding the image to the classifier. Due to this extra step, the legitimate samples will be closer to the range of the generator than the adversarial samples, resulting in substantial reduction of the potential adversarial perturbations.

Let's now learn about MagNet.

3.7 MagNet

MagNet is an adversarial defense framework that uses the classifier as a black box, to read the outputs of the classifier's last layer without reading the data on any internal layer or modifying the classifier. MagNet uses detectors to distinguish the normal samples from the adversarial examples. The detector measures the distance between the given test example and the manifold and, rejects the sample if the distance exceeds a threshold. It also uses a reformer to reform the adversarial example to a similar legitimate example using the autoencoders.

Although MagNet is quite successful in thwarting a range of the black-box attacks, its performance degrades significantly in the case of the white-box attacks where, the attackers are supposed to be aware of the parameters of the MagNet. So, the authors came up with an idea of using varieties of autoencoders and randomly picking one at a time to make it difficult for the adversary to predict which autoencoder was actually used for the defense.

We now have learned about quite a few defense mechanisms against the adversarial attacks. Although these mechanisms are really good at identifying the adversarial attacks, still, they can be broken if the attacker has significantly large number of computational resources. So, an absolute defense against the adversarial attacks, remains a challenge. Let's now learn about the common challenges of defending against the adversarial attacks.

1. Why it is hard to defend against Adversarial Attacks?

The adversarial examples are hard to defend against because: it is difficult to build a model of the process of creating an adversarial example. The adversarial examples are the solutions to an optimization problem (or results of an optimization algorithm) that is non-linear and non-convex for many neural networks.

ML models are required to provide the proper outputs for every possible input. A considerable modification of the model to incorporate the robustness

against the adversarial examples, may also change the elementary objective of the model. And, it may result in a bad performing model.

The defense strategies discussed earlier, work well against only a certain type of attacks but fail for new kind of attacks because they are not adaptive. If an attacker knows the defense strategies used in a system, then it's a major vulnerability. Moreover, the implementation of such defense strategies may cause performance overhead as well as the degradation in model accuracy for the actual or intended inputs.

Thus, it is important to design powerful defense mechanism which are adaptive, to protect ML systems for the adversarial attacks. It is also a growing area of research and let's see what the future holds.

1. Conclusion

In this skill, we learned about the adversarial attacks, common types of adversarial attacks, common defense mechanisms again them and the importance of designing the appropriate defense mechanisms against them to protect the ML based intelligent systems. Specifically, we learned:

- What is adversarial Machine Learning? And, what are the adversarial examples? Why they are dangerous to the ML systems and how they are created?
- What are some of the known types of adversarial attacks and why it is hard to defend against them?
- How can we make our ML systems robust against the various adversarial attacking mechanisms?

In the next skill, we will learn about some impressive applications of the Generative Adversarial Networks.

End of Skill-11

SKILL 12

Impressive Applications of GANs

Generative Adversarial Network, or GAN for short, is a framework for developing generative models using deep learning techniques. The generative models try to learn the underlying distribution of training samples during the training process, and once trained, they can be utilized for generating the desired number of new samples that follow approximately the same distribution as the training dataset.

A GAN is trained using two different neural networks known as: the generator and the discriminator. The generator network learns to generate new plausible samples and the discriminator network learns to differentiate between the real samples and the samples coming from the generator network. These two networks are trained in such a way that the generator network tries to fool the discriminator by generating realistic samples, while the discriminator network tries to get better at distinguishing the fake samples from the real ones.

Once the training is complete, the generator network can be utilized for generating infinite number of plausible samples and the discriminator network is usually discarded. GANs have a large number of applications.

Following is a list of common application areas of GANs:

- Generating Examples for Image Datasets
- Generating Human Faces and Cartoon Characters
- Generating Realistic Photographs
- Image-to-Image Translation
- Text-to-Image Translation
- Photo Editing and Face Aging apps
- Image Super Resolution
- Image Inpainting
- 3D Object Generation
- Video Prediction

Let's learn more about these applications.

1. Generating examples for Image Datasets

Generative Adversarial Networks, or GANs, are capable of generating new plausible samples as described in the original GAN paper by *Ian Goodfellow, et al.* titled '*Generative Adversarial Networks*' in 2014. In this paper, GANs were utilized for generating new plausible examples for the MNIST handwritten digit dataset, the CIFAR-10 small object photograph dataset, and the Toronto Face Database. (See Figure 12.1).

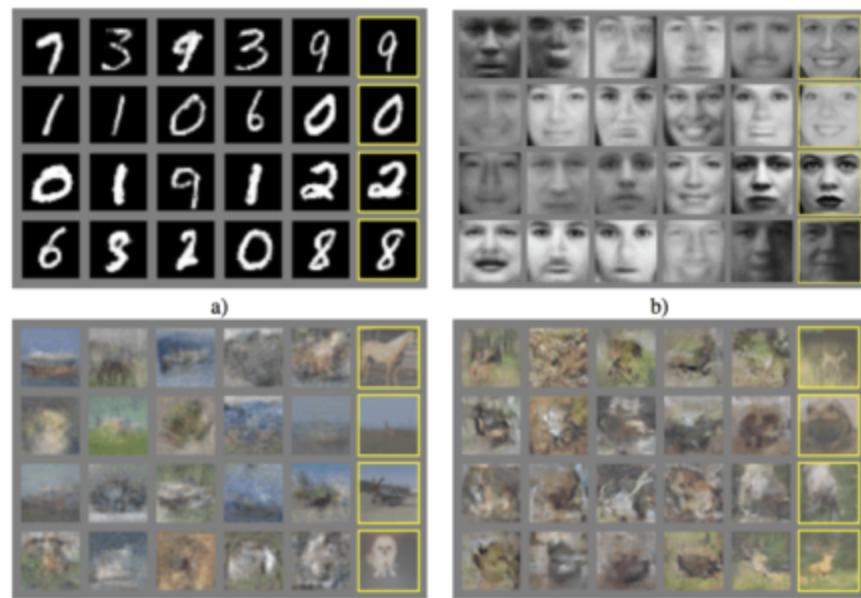


Figure 12.1: Examples of GANs used to Generate New Plausible Examples for Image Datasets. Taken from Generative Adversarial Nets, 2014.

Another important research paper by Alec Radford, et el. titled "*Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*" in 2015 introduced the DCGAN framework for training stable GANs at scale. They demonstrated the potential of DCGAN based generative models (which are GAN based), by generating new examples of bedroom images. (See Figure 12.2).

In this way, GANs can be utilized for generating new plausible images samples related to the given target domain. These generated samples then can be used at multiple places, such as, training better ML models with more data, creating new artwork or designs and so on.

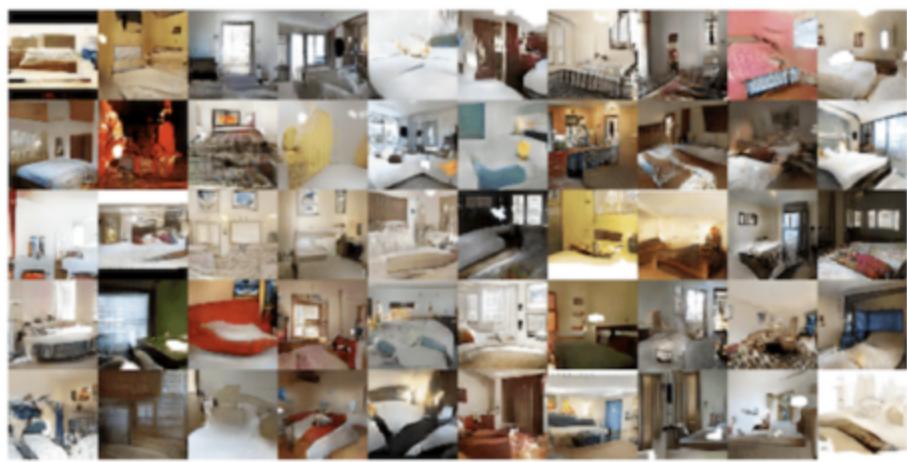


Figure 12.2: Example of GAN-Generated Photographs of Bedrooms. Taken from Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, 2015.

Let's now look at the next application.

1. Generating Human Faces and Cartoon Characters

Generative Adversarial Networks, or GANs, have been successfully trained to generate human face images. These newly generated faces don't actually exist in the real world but look quite real.

Tero Karras, et al. in their 2017 paper titled “*Progressive Growing of GANs for Improved Quality, Stability, and Variation*” demonstrate the generation of plausible realistic photographs of human faces. The results were so realistic that they received a lot of media attention. (See Figure 12.3).



Figure 18: Examples of realistic faces generated through PGGAN model as per the original paper by ‘Tero Karras, et el.’

Similarly, GANs have been utilized for generating Cartoon characters. Yanghua Jin, et el. in their 2017 paper titled “*Towards the Automatic Anime Characters Creation with Generative Adversarial Networks*” demonstrate the capability of GANs for generating the faces of anime characters. Figure 12.4 is taken from the same paper.

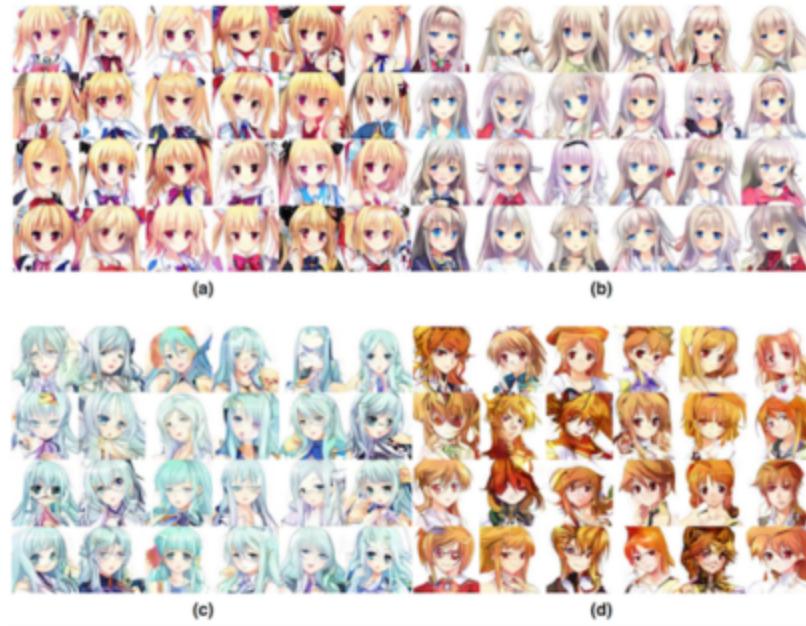


Figure 12.4: Examples of newly generated anime faces using GANs.

Let's now move to the next application of GANs.

1. Generating Realistic Photographs

The improved training methods and network architectures have led the development of GANs that are capable of generating very realistic photographs. The synthetic images generated using *BigGAN* are practically indistinguishable from the real photographs. Figure 12.5 shows some samples generated using the *BigGAN* model.



Figure 12.5: Examples of synthetically generated photographs using Big-GAN, this image is taken from original 2018 Big-GAN paper by Andrew Brock, et al.

Let's jump to the next application.

1. Image-to-Image Translation

Just like Image Generation, Generative Adversarial Networks have also been successfully applied for the task of Image-to-Image translation. *Phillip Isola, et al.* in their 2016 paper titled “*Image-to-Image Translation with Conditional Adversarial Networks*” introduce *Pix2Pix* GAN architecture and demonstrate its image-to-image translation capability for many tasks such as:

- Translation of Satellite Images to Google Maps (see Figure 12.6).
- Translation of daylight photos to night photos (see Figure 12.7).
- Translation of Black-n-White to color photos.
- Translation of sketches to color photos.

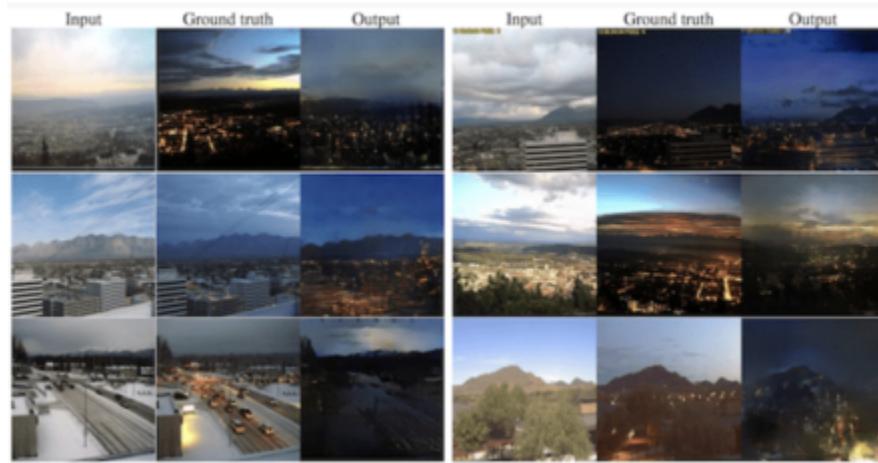


Figure 12.6: Examples of translating daylight photos into night photos using pix2pix-GAN. Taken from Original Pix2Pix GAN paper by Phillip Isola, et el.



Figure 12.7: Examples of translation of satellite images to Google Maps images using Pix2Pix-GAN taken from original paper by Phillip Isola, et el.

Pix2Pix GAN has shown promising results for the task of image-to-image translation. One limitation of *Pix2Pix* framework is that it requires paired examples for training. Thus, it requires a new training dataset creation for each new domain. As it is impossible to create the paired training dataset for every single domain, this limits the capabilities of *Pix2Pix* framework.

Jun-Yan Zhu, et el. in their 2017 paper titled “*Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*” introduce the Cycle GAN architecture. Cycle GAN paper demonstrates impressive unpaired image-to-image translation results for multiple tasks such as (see Figure 12.8):

- Translation of real photographs into paintings.
- Translation of Horses to Zebras and vice-versa.
- Translation of Summer photos to Winter photos.
- Translation of Satellite photos to Google Map photos.

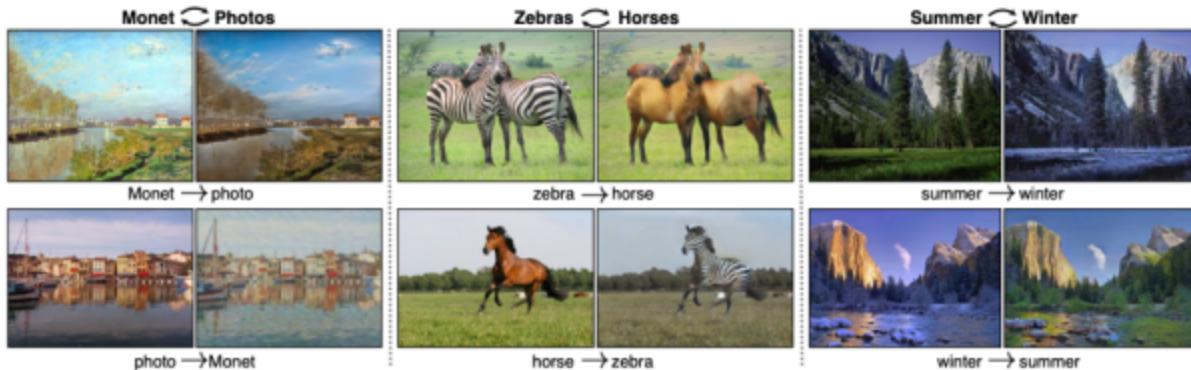


Figure 19: Examples of Unpaired Image-to-Image translation results from original Cycle-GAN paper by Jun-Yan Zhu, et al.

Cycle GAN overcomes the limitation of *Pix2Pix* GAN framework as it works with unpaired training dataset. Let’s now jump to the next application area of GANs.

1. Photo Editing and Face Aging apps

Many popular photo editing apps today, utilize GANs in some of their creative features. Many popular image editing features such as changes in hair colors, styles, face aging, gender changes and so on, inherently utilize GAN based models.

Grigory Antipov, et al. in their 2017 paper titled “*Face Aging with Conditional Generative Adversarial Networks*” use GANs to generate photographs of faces with different apparent ages, from younger to older. (See Figure 12.9).

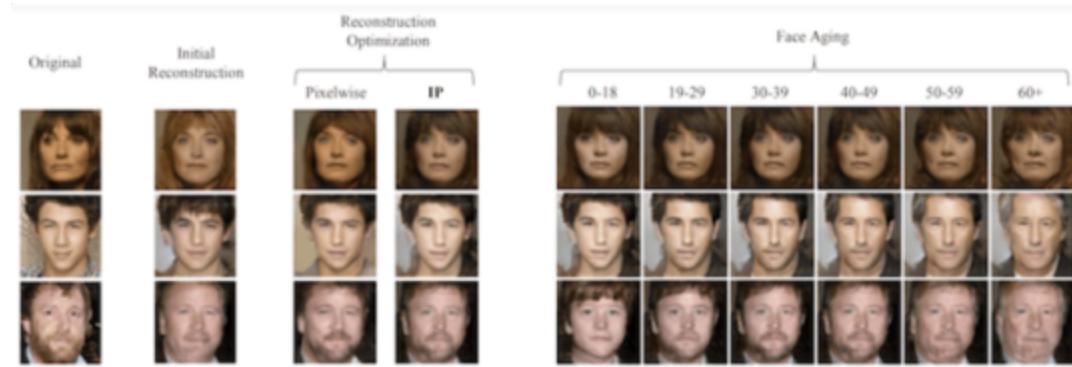


Figure 12.9: Examples of face-aging. Image is taken from Grigory Antipov, et al.

Let's move to the next impressive application of GANs.

1. Image Super Resolution

Super Resolution (or SR) is a class of techniques that enhances the resolution of the imaging system.

In 2016, *Christian Ledig, et al.* proposed SRGAN in his work titled “*Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network*” and demonstrated its capability of generating high-resolution images using low-resolution images as input. Figure 12.10 shows the SRGAN results from the original paper.



Figure 12.10: Example of SRGAN output from original paper.

Next, we will learn about Image Inpainting.

1. Image Inpainting

Image inpainting is the task of filling missing regions in an image. It is an important problem to solve, as it has many applications such as unwanted object removal from a photograph, image restoration, manipulation, image-based rendering and so on.

Creating dataset for image inpainting is quite easy, as we can intentionally cut out some parts of an image and train a neural network to reconstruct the original one. GANs have achieved commendable performance for the task of image inpainting.

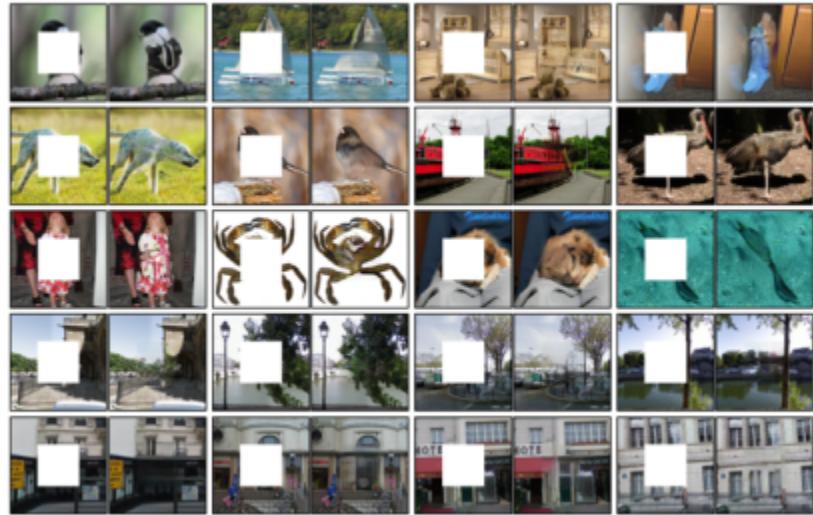


Figure 12.11: Examples of image inpainting from original Context Encoder paper by Deepak Pathak, et al.

Deepak Pathak, et al. in their 2016 paper titled “*Context Encoders: Feature Learning by Inpainting*” describe the use of GANs for photograph inpainting with an architecture called *Context Encoders*. Figure 12.11 shows some results from the original paper. Let’s now learn about 3D object generation using GANs.

1. 3D Object Generation

Another exciting application area for GANs is: 3D object generation.

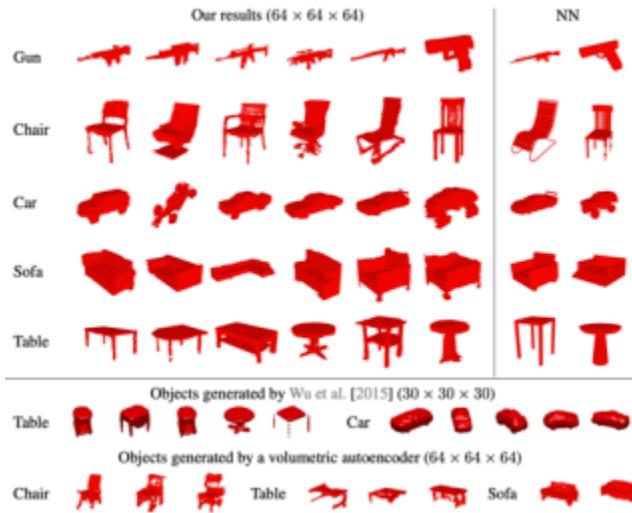


Figure 12.12: Some 3D objects generated using GAN, taken from the paper 'Learning A Probabilistic Latent Space of Object Shapes via 3D Generative Adversarial Modeling' by Jiajun Wu, et al.

Jiajun Wu, et al. in their 2016 paper titled “*Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling*” demonstrate the capability of a GAN based model for generating new three-dimensional objects such as chairs, cars, sofas, tables, and so on. Figure 12.12 shows some results from the original paper.

Let’s now look at the task of Video Prediction.

1. Video Prediction

Video Prediction is the process of predicting which video frames will come next, given a series of past frames. Convolutional LSTM based neural networks have been studied for the application of next-frame prediction in videos.

Carl Vondrick, et al. in their 2016 paper titled “*Generating Videos with Scene Dynamics*” demonstrate the capability of GANs for video prediction up

to a second of video frames with success. Figure 12.13 shows some results from the original paper.

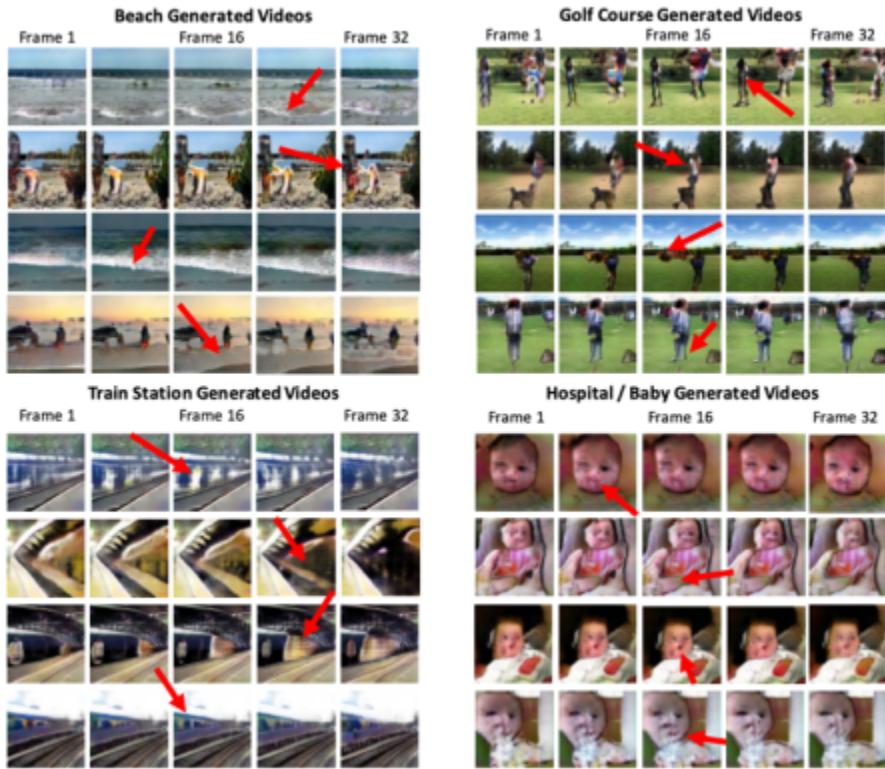


Figure 12.13: Examples of video frame prediction, taken from the paper 'Generating Videos with Scene Dynamics' by Carl Vondrick, et al.

Let's now jump to the next application area of GANs.

1. Text to Image Translation

Text-to-Image translation is the task of generating realistic images of scenes as described in the textual descriptions. Textual descriptions are provided as input, and the generative model is expected to generate a realistic image that matches the input text description.

Han Zhang, et al. in their 2016 paper titled “*StackGAN: Text to photo-realistic Image Synthesis with stacked Generative Adversarial Networks*”,

demonstrate the use of GANs for solving text-to-image translation task. Figure 12.14 shows some results from the original Stack GAN paper.

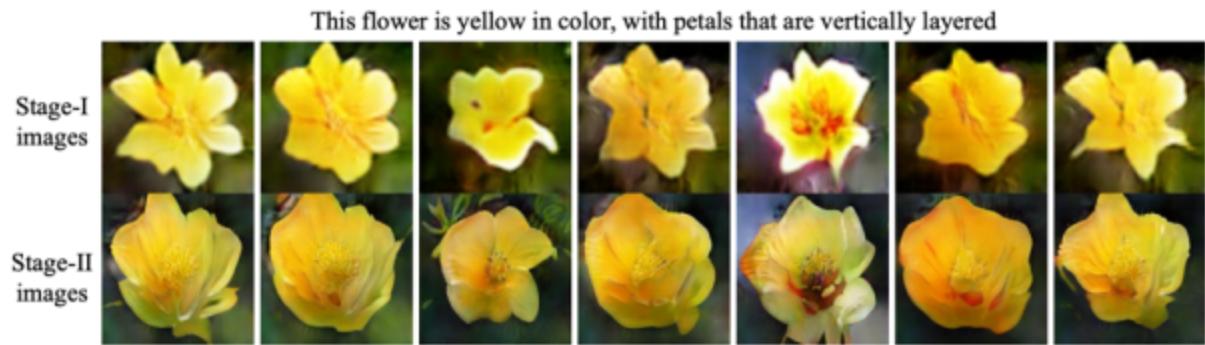


Figure 12.14: Some text-to-image translation results from the original Stack GAN paper. Input text description is present on the top of the Figure.

As we can see in Figure 12.14, Stack GAN is able to generate plausible images from textual description.

We just looked at around 10 impressive applications of GANs. However, GANs are not limited to these 10 applications. They have huge potential to solve lot many problems.

1. Conclusion

In this skill, we discussed about 10 impressive applications of Generative Adversarial Networks or GANs. We saw that GANs are able to effectively solve very complex and important image related problems. The possible application areas for GANs are unlimited, only limited by the imagination of the ML practitioner. After reading this book, the readers should be able to replicate all these applications and also, should be able to apply GANs for solving new impressive problems related.

I hope this skill helps the readers in developing an intuition for the types of problems where GANs can be helpful. In the next skill, we will list down some popular research papers related to GANs.

End of Skill-12

SKILL 13

Top Research Papers

Generative Adversarial Networks, or GANs, are one of the most innovative ideas proposed in last few years. GANs are a type of deep generative models and have been successfully applied for the tasks of image, text and sound generation. GANs have also been applied for the tasks such as image-to-image translation, 3D object generation, video prediction, image inpainting, image super resolution and so on, as seen in the last skill as well.

With increasing popularity and application areas of GANs, it has become important for the AI practitioners to be on top of popular GAN research papers. This skill provides a list of popular GAN research papers with appropriate links to the source of the information. If someone wants to become an expert on GANs then these research papers are a must read, for others looking for any specific application areas, can directly jump to the relevant paper. Still, I would recommend everyone to read at least first five papers from the list given in this skill.

This skill covers the following two topics:

- Top 20 Research Papers on GANs
- Conclusion

Let's get started.

1. Top 20 Research Papers on GANs

In this section, we will list down top 20 research papers related to GANs. These research papers represent some important details, extensions and application areas of GANs. Let's get started.

1.1 Paper #1

Title: “*Generative Adversarial Nets.*”

Authors: *Ian Goodfellow, et al.*

Year: 2014

Link: <https://arxiv.org/pdf/1406.2661.pdf>

Overview: This paper introduces GANs, as a new framework for Generative Learning. Ian Goodfellow introduced GANs in 2014, and since then, it has gained lots of popularity. As this is the foundational paper and has explained the framework beautifully, it is recommended to read this research paper as a first thing.

1.2 Paper #2

Title: “Unsupervised representation learning with deep convolutional generative adversarial networks”

Authors: *Alec Radford, et al.*

Year: 2015

Link: <https://arxiv.org/pdf/1511.06434.pdf>

Overview: This paper extends the idea of GANs, using deep convolutional networks and studies the capability of this framework for unsupervised representation learning. DCGAN was introduced in 2015, and it showed a way to develop CNN based GANs for plausible image generation. This paper shares some important guidelines for designing stable GAN architectures using CNN layers.

1.3 Paper #3

Title: “*Conditional Generative Adversarial Nets*”

Authors: *Mirza, Mehdi, and Simon Osindero.*

Year: 2014

Link: <https://arxiv.org/pdf/1411.1784.pdf>

Overview: In this paper, the authors have tried condition-based content generation by modifying the concept of the traditional GAN a little bit. Conditional GANs, provide us some control over the generated content. This control is quite useful for many image generation tasks. So, it is important to learn about the conditional GANs.

1.4 Paper #4

Title: “*Semi-Supervised Learning with Generative Adversarial Networks*”

Authors: *Augustus Odena*

Year: 2016

Link: <https://arxiv.org/pdf/1606.01583.pdf>

Overview: This study proves the capability of GAN based models for semi-supervised learning. SSGAN or SGAN was proposed in 2016. SGAN reveals the capability of GANs for learning unsupervised representations that can help in boosting the performance of supervised ML tasks such as classification when the labelled training data is limited.

1.5 Paper #5

Title: “*Infogan: Interpretable representation learning by information maximizing generative adversarial nets*”

Authors: *Xi Chen, et al.*

Year: 2016

Link: <https://arxiv.org/pdf/1606.03657.pdf>

Overview: This study explores the potential of learning interpretable representations by making small changes to the basic concept of GANs. Info GANs uses a small trick to make the GANs learn interpretable representations without making the overall framework complex.

1.6 Paper #6

Title: “*Conditional Image Synthesis with Auxiliary Classifier GANs*”

Authors: *Augustus Odena, et al.*

Year: 2017

Link: <https://arxiv.org/pdf/1610.09585.pdf>

Overview: This paper shows a new way of generating conditioned content using GANs. ACGANs were introduced in 2017, and they are able to generate class conditioned results. This method, however, requires class labelled training data for leaning class distributions. ACGANs were able to achieve great class conditioned results for many multi-class image generation tasks.

1.7 Paper #7

Title: “*Wasserstein generative adversarial networks*”

Authors: *Martin Arjovsky, et al.*

Year: 2017

Link: <https://arxiv.org/pdf/1701.07875.pdf>

Overview: WGAN was introduced in 2017. WGAN paper shows a new way of designing objective function for training GANs which makes the training process stable. This paper introduces *Wasserstein* loss for trainings GANs.

1.8 Paper #8

Title: “*Improved Training of Wasserstein GANs*”

Authors: *Ishaan Gulrajani, et al.*

Year: 2017

Link: <https://arxiv.org/pdf/1704.00028.pdf>

Overview: This paper introduces WGAN-GP which is an extension and improvement over the WGAN framework. WGAN-GP uses a gradient penalty mechanism to improve the training and results of WGAN framework.

1.9 Paper #9

Title: “*Least squares generative adversarial networks*”

Authors: *Xudong Mao, et al.*

Year: 2017

Link: <https://arxiv.org/pdf/1611.04076.pdf>

Overview: LSGAN was introduced in 2017. This paper applies a new loss function: least squares error, for training GANs. LSGAN was more stable and provided high quality results.

1.10 Paper #10

Title: “*Image-to-Image Translation with Conditional Adversarial Networks*”

Authors: *Phillip Isola, et al.*

Year: 2017

Link: <https://arxiv.org/pdf/1611.07004.pdf>

Overview: Pix2Pix GAN was introduced in 2017 for the task of image-to-image translation. This approach requires a paired dataset for converting the images from a source domain to the given target domain. The results achieved by this framework were remarkable.

1.11 Paper #11

Title: “Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Nets”

Authors: *Jun-Yan Zhu, et al.*

Year: 2017

Link: <https://arxiv.org/pdf/1703.10593.pdf>

Overview: Cycle GAN was introduced in 2017 for the task of image-to-image translation. Unlike the Pix2Pix GAN, Cycle GAN does not require a paired dataset for learning the image-to-image translation tasks. The results achieved by Cycle GAN were quite impressive even without a paired dataset.

1.12 Paper #12

Title: “Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Networks”

Authors: *Christian Ledig, et al.*

Year: 2017

Link: <https://arxiv.org/pdf/1609.04802.pdf>

Overview: SRGAN was introduced in 2017. This paper shows the capability of GANs for converting low-resolution images into high-resolution images. SRGAN achieves impressive results for upto 4 x resolution upgrade.

1.13 Paper #13

Title: “Learning to Discover Cross-Domain Relations with Generative Adversarial Networks”

Authors: *Taeksoo kim, et al.*

Year: 2017

Link: <https://arxiv.org/pdf/1703.05192.pdf>

Overview: Disco GAN was introduced in 2017. It is capable of learning cross domain relationships without requiring any paired dataset. This framework is quite handy for style transfer related use cases.

1.14 Paper #14

Title: “*CartoonGAN: Generative Adversarial Networks for Photo Cartoonization*”

Authors: Yang Chen, et al.

Year: 2018

Link:

https://openaccess.thecvf.com/content_cvpr_2018/papers/Chen_CartoonGAN_Generative_Adversarial_CVPR_2018_paper.pdf

Overview: Cartoon GAN was introduced in 2018. This framework is capable of converting realistic scenic images into their *cartoonized* versions without needing any labelled dataset.

1.15 Paper #15

Title: “*Context Encoders: Feature Learning by Inpainting*”

Authors: Deepak Pathak, et al.

Year: 2016

Link: <https://arxiv.org/pdf/1604.07379.pdf>

Overview: Context Encoders were introduced in 2016. They are capable of solving the image inpainting problem very efficiently. Image inpainting has a large number of applications in image editing.

1.16 Paper #16

Title: “*Large Scale GAN Training for High Fidelity Natural Image Synthesis*”

Authors: Andrew Brock, et al.

Year: 2018

Link: <https://arxiv.org/pdf/1809.11096.pdf>

Overview: BigGAN was introduced in 2018. This paper provides some guidelines for scaling GANs for generating high-fidelity natural images. BigGAN is capable of generating realistic looking high resolution images.

1.17 Paper #17

Title: “*Progressive Growing of GANs for Improved Quality, Stability, and Variation*”

Authors: *Tero karras, et al.*

Year: 2017

Link: <https://arxiv.org/pdf/1710.10196.pdf>

Overview: PGGAN was introduced in 2017. PGGAN presents a new way of scaling GANs for generating high-resolution images. This framework was successful in generating very high-quality and realistic images.

1.18 Paper #18

Title: “*A Style-Based Generator Architecture for Generative Adversarial Networks*”

Authors: *Tero Karras, et al.*

Year: 2019

Link: <https://arxiv.org/pdf/1812.04948.pdf>

Overview: StyleGAN was introduced in 2019. This framework introduces many changes and guidelines for designing the generator part of GANs, in such way that it results in more control over the generated styles of the images and also, the resulting images are very realistic and high-quality.

1.19 Paper #19

Title: “*Pros and Cons of GAN Evaluation Measures*”

Authors: *Ali Borji*

Year: 2018

Link: <https://arxiv.org/pdf/1802.03446.pdf>

Overview: This paper discusses about the common evaluation techniques of GANs with their common pros and cons.

1.20 Paper #20

Title: “*Improved Techniques for Training GANs*”

Authors: *Tim Salimans, et al.*

Year: 2016

Link: <https://arxiv.org/pdf/1606.03498.pdf>

Overview: This paper shows presents guidelines for training stable GANs.

1. Conclusion

In this skill, we have gone through a list of 20 popular research papers. If we are able to read these research papers, our foundation about GANs will be very strong. This book has covered important details from most of these research papers already. But still, if you feel the need to learn any of these concepts in more details, kindly read the relevant research paper thoroughly.

This book ends here. *Congratulations!* for reaching this ending page. I hope it was helpful for all the readers in learning GANs and achieving their goals. I would love to hear your feedback about the content covered in this book. Kindly leave a review.

Once again thank you for reading this book!

It makes all my efforts for writing this book count ☺

Happy Coding!!

End of Skill-13

Bibliography

- Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... & Bengio, Y. (2014). Generative adversarial nets. *Advances in neural information processing systems*, 27.
- Radford, A., Metz, L., & Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*.
- Mirza, M., & Osindero, S. (2014). Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*.
- Odena, A. (2016). Semi-supervised learning with generative adversarial networks. *arXiv preprint arXiv:1606.01583*.
- Chen, X., Duan, Y., Houthooft, R., Schulman, J., Sutskever, I., & Abbeel, P. (2016). Infogan: Interpretable representation learning by information maximizing generative adversarial nets. *Advances in neural information processing systems*, 29.
- Odena, A., Olah, C., & Shlens, J. (2017, July). Conditional image synthesis with auxiliary classifier gans. In *International conference on machine learning* (pp. 2642-2651). PMLR.
- Arjovsky, M., Chintala, S., & Bottou, L. (2017, July). Wasserstein generative adversarial networks. In *International conference on machine learning* (pp. 214-223). PMLR.
- Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., & Courville, A. C. (2017). Improved training of wasserstein gans. *Advances in neural information processing systems*, 30.

- Mao, X., Li, Q., Xie, H., Lau, R. Y., Wang, Z., & Paul Smolley, S. (2017). Least squares generative adversarial networks. In Proceedings of the IEEE international conference on computer vision (pp. 2794-2802).
- Isola, P., Zhu, J. Y., Zhou, T., & Efros, A. A. (2017). Image-to-image translation with conditional adversarial networks. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 1125-1134).
- Zhu, J. Y., Park, T., Isola, P., & Efros, A. A. (2017). Unpaired image-to-image translation using cycle-consistent adversarial networks. In Proceedings of the IEEE international conference on computer vision (pp. 2223-2232).
- Ledig, C., Theis, L., Huszár, F., Caballero, J., Cunningham, A., Acosta, A., ... & Shi, W. (2017). Photo-realistic single image super-resolution using a generative adversarial network. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 4681-4690).
- Kim, T., Cha, M., Kim, H., Lee, J. K., & Kim, J. (2017, July). Learning to discover cross-domain relations with generative adversarial networks. In International conference on machine learning (pp. 1857-1865). PMLR.
- Chen, Y., Lai, Y. K., & Liu, Y. J. (2018). Cartoongan: Generative adversarial networks for photo cartoonization. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 9465-9474).
- Pathak, D., Krahenbuhl, P., Donahue, J., Darrell, T., & Efros, A. A. (2016). Context encoders: Feature learning by inpainting. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 2536-2544).
- Brock, A., Donahue, J., & Simonyan, K. (2018). Large scale GAN training for high fidelity natural image synthesis. arXiv preprint arXiv:1809.11096.

- Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2017). Progressive growing of gans for improved quality, stability, and variation. arXiv preprint arXiv:1710.10196.
- Karras, T., Laine, S., & Aila, T. (2019). A style-based generator architecture for generative adversarial networks. In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition (pp. 4401-4410).
- Chaudhary, K. (2020). Understanding audio data, fourier transform, fft and spectrogram features for a speech recognition system. Towards Data Science-Medium.
- Kartik, C. Understanding Audio Data, Fourier Transform, FFT and Spectrogram Features for a Speech Recognition System.
- Borji, A. (2019). Pros and cons of gan evaluation measures. Computer Vision and Image Understanding, 179, 41-65.
- Borji, A. (2022). Pros and cons of GAN evaluation measures: New developments. Computer Vision and Image Understanding, 215, 103329.
- Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., & Chen, X. (2016). Improved techniques for training gans. Advances in neural information processing systems, 29.
- Chaudhary, Kartik. "How does a Generative Learning Model Work?". Web blog post. *Drops of AI*. Web. 16 January 2024.
- Chaudhary, Kartik. "Image Synthesis using Pixel CNN based Autoregressive Generative Model." Web blog post. *Drops of AI*. Web. 17 February 2024.
- Chaudhary, Kartik. "What are Autoregressive Generative Models." Web blog post. *Drops of AI*. Web. 2 February 2024.

- Chaudhary, Kartik. "Building blocks of Deep Generative Models." Web blog post. *Drops of AI*. Web. 23 January 2024.
- Chaudhary, Kartik. "Generative Learning and its Differences from the Discriminative Learning." Web blog post. *Drops of AI*. Web. 9 January 2024.
- Chaudhary, Kartik. "Variational AutoEncoders and Image Generation with Keras." Web blog post. *Drops of AI*. Web. 10 November 2020.