VISVESVARAYA TECHNOLOGICAL UNIVERSITY
JNANA SANGAMA, BELAGAVI 590018

Project Entitled

IMAGE REGENERATION WITH GENERATIVE MODELS

Submitted in partial fulfillment of the requirements for the award of degree of
BACHELOR OF ENGINEERING
In
COMPUTER SCIENCE AND ENGINEERING
For the Academic year 2017-2018

Submitted by

ABHIJITH C.	**1MV14CS004**
RAGHAVA G. DHANYA	**1MV14CS077**
SHASHANK S.	**1MV14CS131**

Project carried out at
Sir M. Visvesvaraya Institute of Technology
Bengaluru-562157

Under the Guidance of
MRS. SUSHILA SHIDNAL
Assistant Professor, Department of CSE
Sir M Visvesvaraya Institute of Technology, Bengaluru.

Department Of Computer Science & Engineering
Sir M. Visvesvaraya Institute Of Technology
Hunasamaranahalli, Bengaluru 562157

VISVESVARAYA TECHNOLOGICAL UNIVERSITY
JNANA SANGAMA, BELAGAVI 590018

Project Entitled

IMAGE REGENERATION WITH GENERATIVE MODELS

Submitted in partial fulfillment of the requirements for the award of degree of
BACHELOR OF ENGINEERING
In
COMPUTER SCIENCE AND ENGINEERING
For the Academic year 2017-2018

Submitted by

ABHIJITH C.	**1MV14CS004**
RAGHAVA G. DHANYA	**1MV14CS077**
SHASHANK S.	**1MV14CS131**

Project carried out at
Sir M. Visvesvaraya Institute of Technology
Bengaluru-562157

Under the Guidance of
MRS. SUSHILA SHIDNAL
Assistant Professor, Department of CSE
Sir M Visvesvaraya Institute of Technology, Bengaluru.

Department Of Computer Science & Engineering
Sir M. Visvesvaraya Institute Of Technology
Hunasamaranahalli, Bengaluru 562157

SIR M VISVESVARAYA INSTITUTE OF TECHNOLOGY

BENGALURU 562157

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

This is to certify that the project work entitled "Image Regeneration With Generative Models" has been carried out by Abhijith C. (1MV14CS004), Raghava G. Dhanya (1MV14CS077) and Shashank S. (1MV14CS131), bona-fide students of Sir M Visvesvaraya Institute of Technology, in partial fulfillment for the award of the Degree of Bachelor of Engineering in Computer Science and Engineering of the Visvesvaraya Technological University, Belagavi during the year 2017-2018. It is certified that all corrections and suggestions indicated during Internal Assessment have been incorporated into the report deposited in the department library. The project report has been approved as it satisfies the academic requirements with respect to the project work prescribed for the course of Bachelor of Engineering.

MRS. SUSHILA SHIDNAL	**PROF. DILIP K SEN**	**DR. V. R. MANJUNATH**
Asst. Prof & Internal Guide	**Head of Department**	**Principal**
Dept. of CSE, Sir MVIT	**Dept. of CSE, Sir MVIT**	**Sir MVIT**

Name of the examiners **Signature with date**

1)

2)

DECLARATION

We hereby declare that the entire project work embodied in this dissertation has been carried out by us and no part has been submitted for any degree or diploma of any institution previously.

Place: Bengaluru
Date:

Signature of Students

Abhijith C.
1MV14CS004

Raghava G. Dhanya
1MV14CS077

Shashank S.
1MV14CS131

ACKNOWLEDGMENTS

This project would not have been possible without the contributions of many people.

We express our sincere thanks to the administration of **Sir M. Visvesvaraya Institute of Technology**, Bengaluru, which has given us the opportunity and the resources to carry out our projects in their facilities.

We would also like to convey our sincere thanks to **Dr. V. R. Manjunath**, principal, Sir MVIT, for his kind support.

We would like to thank **Prof. Dilip K. Sen**, Head of the department, CSE, Sir MVIT, for providing an intellectual environment where we could devote a tremendous amount of time to work on this project.

We wish to express our heartfelt gratitude to our guide, **Smt. Sushila Shidnal**, for her invaluable guidance, encouragement, support and advice.

We thank **Google** for **Google colaboratory** where we ran most of our Deep Learning model training.

Our immense gratitude for the members of the faculty of **Department of Computer Science and Engineering**, Sir MVIT, for their support and co-operation. Lastly we wish to thank all our friends for their help and suggestions, without which this project would not have been possible.

- ABHIJITH C.	**1MV14CS004**
- RAGHAVA G. DHANYA	**1MV14CS077**
- SHASHANK S.	**1MV14CS131**

ABSTRACT

Current advances in Generative Adversarial Networks allow us to obtain near realistic images but it is still quite distinguishable from actual photographic images. The technology is also not very amiable to changes in the orientation of images in Convolutional Neural Networks(CNN). Additionally, the amount of data required to train the network must be exhaustible, for example, in case different perspectives of a face are required the various perspectives must be explicitly present in the training data to achieve the result. Thus the network requires humongous amounts of data.

In this project we propose a novel approach to accomplish the same results using CapsNet. CapsNet employs a dynamic routing algorithm which replaces the scalar-output feature detectors of the CNN with vector-output capsules. A capsule is essentially a group of neurons describing a specific part of object or image. Active capsules at one level make predictions, via transformation matrices, for the instantiation parameters of higher-level capsules. In essence, the CapsNet is the reverse of the common Computer Graphics pipeline where we convert objects to their renders. The CapsNet works from the pixel level and works up towards the object.

We propose that the amount of data required to train a comparable model is very small while it gives comparable, if not better, results.

CONTENTS

LIST OF FIGURES

Chapter 1
Introduction

CHAPTER 1

INTRODUCTION

"What I cannot create, I do not understand."

Richard Feynman

One of the main aspirations of Artificial Intelligence is to develop algorithms and techniques that enrich computers with ability to understand our world. Generative models are one of the most promising approaches towards achieving this goal.

1.1 Generative Models

A generative model is a mathematical or statistical model to generate all values of a phenomena. To train such a model, we first collect a large amount of data in some domain (e.g., think millions of images, sentences, or sounds, etc.) and then train a model to generate data like it.

A generative algorithm models how data was generated to classify a data instance. It poses the question: according to my generation hypotheses, which category is most likely to generate this data instance? A discriminative algorithm does not care about how the data was generated, it just classifies a given data instance; that is, given the features of a data instance, they predict a label or category to which that data belong. Discriminative models learn the boundary between classes while Generative models model the distribution of individual classes; that is, a generative model learns the joint probability distribution $p(x, y)$ while a discriminative model learns the conditional probability distribution $p(y|x)$ "probability of y given x".

The trick is that the neural networks that we use as generating models have a significantly smaller number of parameters than the amount of data on which we train them, so the models are forced to effectively discover and internalize the essence of the data to generate it.

There are multiple approaches to build a generative models

1.1.1 Generative adversarial networks

Generative adversarial networks (GANs) are a class of generative algorithms used in unsupervised machine learning, implemented by a system of two neural networks competing in a zero-sum game framework. They were presented by Ian Goodfellow *et al*. [13]. This technique can generate photographs that seem at least superficially authentic to human observers, having many realistic features (though in tests people can tell real from generated in some cases).

1.1.2 Variational Autoencoders

An autoencoder network is actually a pair of two connected networks, an encoder and a decoder. An encoder network receives an input and converts it into a smaller, denser representation that the decoder network can use to convert it back to the original input. Variational Autoencoders (VAEs) have one fundamentally unique property that separates them from vanilla autoencoders, and it is this property that makes them so useful for generative modeling: their latent spaces are, by design, continuous, allowing easy random sampling and interpolation. Variational Autoencoders (VAEs) allow us to formalize generative modeling problem in the framework of probabilistic graphical models where we are maximizing a lower bound on the log likelihood of the data

1.1.3 Autoregressive models

Autoregressive models such as PixelRNN, on the other hand train a network that models the conditional distribution of every individual pixel given previous pixels (to the left and to the top). These models efficiently generate independent,exact samples via ancestral sampling. This is similar to plugging the pixels of the image into a char-rnn, but the RNNs runs both horizontally and vertically over the image instead of just a 1D sequence of characters.

1.2 Generative Adversarial Networks

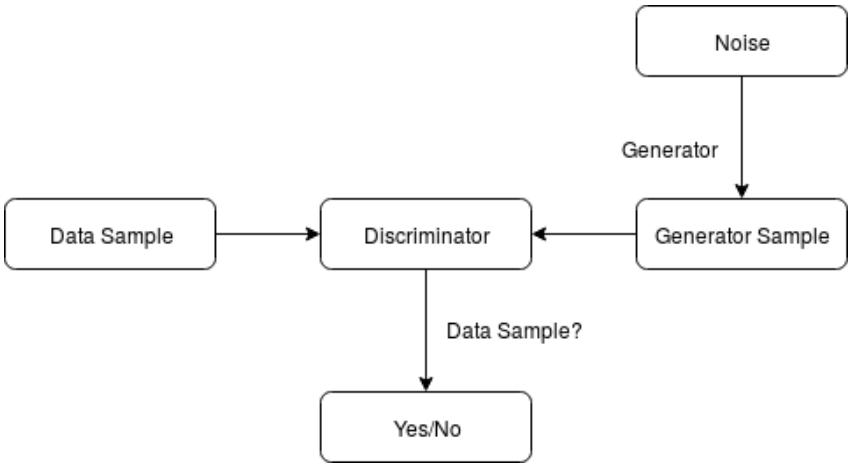


Figure 1.1: Vanilla Generative Adversarial Network

Generative Adversarial Networks, which we already discussed above, pose the training process as a game between two distinct networks: A neural network, called the generator, generates new instances of data, while the other, the discriminator, evaluates their authenticity; discriminator network tries to classify samples as either coming from the true distribution $p(x)$ or the model distribution $\hat{p}(x)$. Every time the discriminator notices a difference between the two distributions the generator adjusts its parameters slightly to make it go away, until at the end (in theory) the generator exactly reproduces the true data distribution and the discriminator is guessing at random, unable to find a difference.

The generator takes noise as input and attempts to produce an image that belongs to the real distribution; that is, it tries to fool the discriminator to accept it as real image. Discriminator takes a generated image or a real image as input and attempts to correctly classify the image as real or fake (generated).

To learn the distribution of the generator p_g over data $\boldsymbol{x}$, we define a prior on input noise variables $p_z(\boldsymbol{z})$, then represent a mapping to data space as $G(\boldsymbol{z};\theta_g)$, where G is a differentiable function represented by a neural network with parameters θ_g. We define a second neural network $D(\boldsymbol{x};\theta_d)$ that outputs a single scalar. $D(\boldsymbol{x})$ represents the probability that $\boldsymbol{x}$ came from the data rather than p_g. We train D to maximize the probability of assigning the correct label to the training examples and samples of G. We simultaneously train G to minimize $\log(1 - D(G(\boldsymbol{z})))$.

This can be represented minimax game

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_z(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))] \tag{1.1}$$

1.3 Convolutional Neural Networks

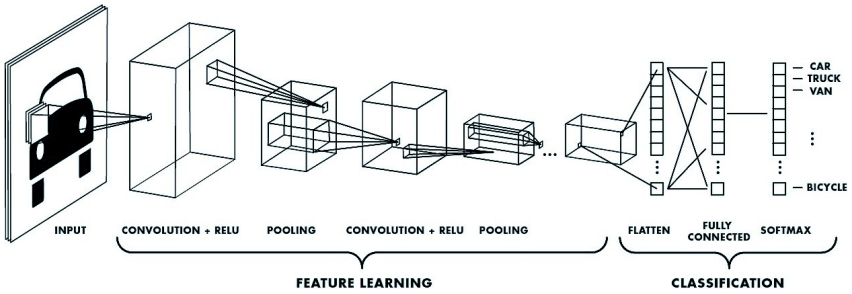


Figure 1.2: Convolutional Neural Network

Before we can jump to understanding Capsule Networks, we need to know about Convolutional Neural Networks(CNNs). CNNs are very similar to ordinary neural networks, they consist of neurons that have learn-able weights and biases. Each neuron receives inputs, performs a scalar product and possibly follows it with a nonlinearity. The entire network expresses a single differentiable score function: raw image pixels at one end to class scores at the other end. And they still have a loss function on the last layer.

The major difference is that CNN explicitly assumes that the inputs are images, which allows us to encode certain properties in the architecture. These then make the forward functions more efficient to implement and significantly reduces the amount of parameters in the network.

Ordinary neural networks don't scale well to full images, for example, A colour image of dimensions of 150x150 (which is considered as low resolution by most people) has a shape (150,150,3), a fully connected neuron on first layer which receives this image would require 67500 weights. Unlike an ordinary neural network, the layers of a CNN have neurons arranged in 3 dimensions: width, height, depth as shown in figure 1.2. The neurons in a layer will only be connected to a small region of the layer

before it, instead of all of the neurons in a fully-connected manner. CNN will reduce the full image into a single vector of class scores, arranged along the depth dimension.

CNNs use a "Pooling" layer to reduce the spatial size of the input for each convolutional layer. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, generally using the MAX operation, hence pooling layer is sometimes referred to as Max Pooling layer.

1.4 Capsule Networks

"The pooling operation used in convolutional neural networks is a big mistake and the fact that it works so well is a disaster." says Geoffrey Hinton, one of the founders of deep learning (Also known as Godfather of Deep Learning) and an inventor of numerous models and algorithms that are widely used today. CNNs perform exceptionally great when they are classifying images which are very close to the data set. If the images have rotation, tilt or any other different orientation then CNNs have poor performance. This problem is usually partially solved by adding different variations of the same image during training. But CNNs still require large amount of data to perform reasonably well. We use pooling after each layer to make it compute in reasonable time frames. But in essence, it also loses out the positional data.

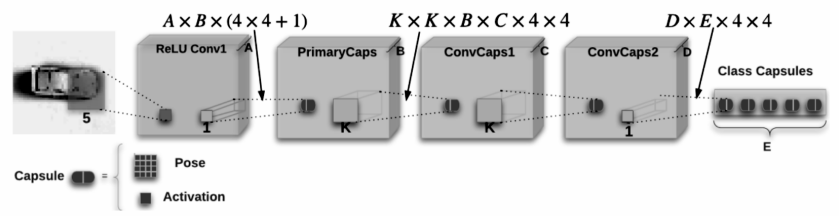


Figure 1.3: Capsule Networks

What we need is not invariance but equivariance. Invariance makes a CNN tolerant to small changes in the viewpoint. Equivariance makes a CNN understand the rotation or proportion change and adapt itself accordingly so that the spatial positioning inside an image is not lost. This leads us to Capsule Networks.

Capsule is a nested set of neural layers as shown in figure 1.3. Capsules are like cortical columns in human brains. Deep neural nets learn by back-propagation of errors over the entire network. In contrast real brains supposedly wire neurons by Hebbian principles: "units that fire together, wire together". Capsules mimic Hebbian learning in the way that: "A lower-level capsule prefers to send its output to higher level capsules whose activity vectors have a big scalar product with the prediction coming from the lower-level capsule". Capsules, combination of capsules encodes objects parts AND their relative positions, so an object instance can be accurately derived from the presence of the parts at the right locations, and not just their presence. Capsules produce equivariant features. Capsules predict the activity of higher-layer capsules to route information to right higher-layer capsules, this is called "Dynamic routing".

1.5 Semantic Inpainting

To demonstrate the application of our modified GAN, we will be using Semantic Inpainting. Inpainting is the process of reconstructing lost or deteriorated parts of

images and videos. In the museum world, in the case of a valuable painting, this task would be carried out by a skilled art conservator or art restorer. In the digital world, inpainting (also known as image interpolation or video interpolation) refers to the application of sophisticated algorithms to replace lost or corrupted parts of the image data (mainly small regions or to remove small defects). Here we will be using GAN to implement semantic inpainting.

1.6 Scope of work

Generative Adversarial Networks are one of the hottest topics in Deep Learning right now. The applications of GANs are far ranging and immense. Creating Infographics from text, creating animations for rapid development of marketing content, generating website designs are to name a few. Our focus in this project is to implement a way to complete images of faces by generating the missing pieces using a GAN.

This particular implementation of the technology would be immensely useful in a variety of circumstances. A few straightforward applications include face sketching of suspects in a crime using eye witness accounts, super resolution of CCTV camera footage to enhance faces, filling in of old degraded color photos, etc.

1.7 Motivation

The existing latest state-of-the-art GAN architectures use Convolution Neural Networks in their Generators and Discriminators. The CNNs are said to have the drawbacks as mentioned before, where they cannot understand orientation and spatial relationships unless they are extensively trained with all possible images. This major drawback is handled by Capsule Networks.

Using the CapsNet architecture into the Generator/Discriminator could improve these Adversarial Networks quite drastically. This mating of the revolutionary Generative Adversarial Networks along with the ground-breaking Capsule Networks, resulting in “Capsule Net GANs” is the overarching objective.

CHAPTER 2
LITERATURE SURVEY

CHAPTER 2

LITERATURE SURVEY

"Adversarial training is the coolest thing since sliced bread"

Yann LeCun,
Director of AI Research at Facebook and Professor at NYU

GANs were first introduced by Ian Goodfellow *et al*. [13] in Neural Information Processing Systems 2014. The paper proposes a completely new framework for estimating generative models via an adversarial process. In this process two models are simultaneously trained. According to [13] the network has a generative model G that captures the data distribution, and a discriminative model D that estimates the probability that a sample came from the training data rather than G. This original work by Ian Goodfellow uses fully connected neural networks in the generator and the discriminator.

2.1 DCGAN

Since GANs were introduced, there has been tremendous advancements in Deep Learning. A convolutional neural network (CNN, or ConvNet) [15] is a class of deep, feed-forward artificial neural networks that has successfully been applied to analyzing visual imagery. The convolution layer parameters consist of a set of learn-able filters, also called as kernels, which have a small receptive field, but they extend through the full depth of the input volume. As a result, the network learns filters that activate when it detects some specific type of feature at some spatial position in the input.

A breakthrough development that occurred in Adversarial Networks was the introduction of "Deep Convolutional Generative Adversarial Networks" by Alec Radford *et al* [12]. DCGAN uses CNNs as generator and discriminator as shown in 8.2. He applied a list of empirically validated tricks as the substitution of pooling and fully connected layers with convolutional layers.

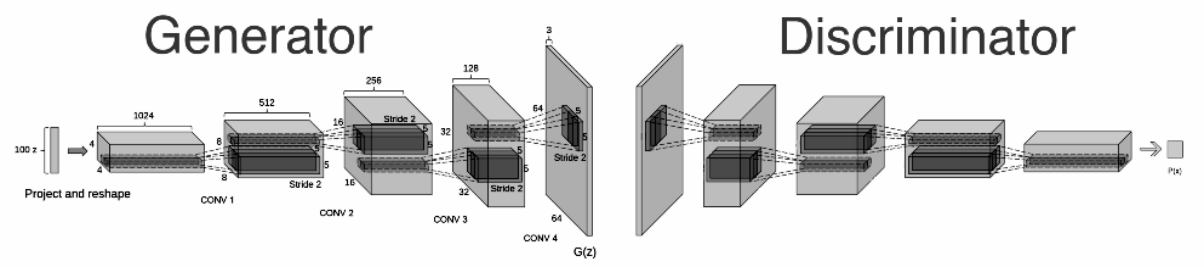


Figure 2.1: Deep Convolutional Generative Adversarial Network

Today, most GANs are loosely based on the former shown DCGAN [12] architecture. Many papers have focused on improving the setup to enhance stability and

performance. Many key insights was given by Salimans *et al*. [8], like Usage of convolution with stride instead of pooling, Usage of Virtual Batch Normalization, Usage of Minibatch Discrimination in DD, Replacement of Stochastic Gradient Descent with Adam Optimizer [6], Usage of one-sided label smoothing.

2.2 InfoGAN

The power of the features encoded in the latent variables was further explored by Chen at al. [4]. They propose an algorithm which is completely unsupervised, unlike previous approaches which involved supervision, and learns interpretable and disentangled representations on challenging datasets. Their approach only adds a negligible computation cost on top of GAN and is easy to train.

2.3 ACGAN

Augustus Odena *et al* [7] came up a improved training of generative adversarial networks and variant of GAN with employing label conditioning that results in image samples exhibiting global coherence. ACGAN uses an auxiliary classifier to control the minimax game between generator and discriminator. In their work they demonstrate that adding more structure to the GAN latent space along with a specialized cost function results in higher quality samples

2.4 WGAN

Another huge development came with the introduction of Wasserstein GANs by Martin Arjovsky [1] . He introduced a new algorithm named WGAN, an alternative to traditional GAN training. In this new model, he showed that the stability of learning can be improved, remove problems like mode collapse, and provide good learning curves useful for debugging and hyperparameter searches.

This recently proposed Wasserstein GAN (WGAN) [1] makes progress toward stable training of GANs, but sometimes can still generate only low-quality images or fail to converge. Ishaan Gulrajani with Martin Arjovsky proposed an alternative in [2] to fix the issues the previous GAN faced. This proposed method performs better than standard WGAN and enables stable training of a wide variety of GAN architectures with almost no hyperparameter tuning, including 101-layer ResNets [11] and language models over discrete data.

2.5 Other GANs

Work by Mehdi Mirza *et al*. [14] introduced the conditional version of GAN which can be constructed by simply feeding the data, y, we wish to condition on to both the generator and discriminator. The CGAN results were comparable with some other networks, but were outperformed by several other approaches – including non-conditional adversarial nets.

Sebastian Nowozin *et al*. [6] discussed the benefits of various choices of divergence functions on training complexity and the quality of the obtained generative models. They show that any f-divergence can be used for training generative neural samplers.

Ming-Yu *et al*. [5] proposed coupled generative adversarial network (CoGAN) for learning a joint distribution of multi-domain images. The existing approaches requires tuples of corresponding images in different domains in the training data set. CoGAN can learn a joint distribution without any tuple of corresponding images.

2.6 Capsule Neural Network

A big breakthrough in the field of Deep Learning came with the introduction of CapsNets or Capsule Networks [3] by the Godfather of Deep Learning, Geoffrey Hinton *et al*. CNNs perform exceptionally great when they are classifying images which are very close to the data set. If the images have rotation, tilt or any other different orientation then CNNs have poor performance. A capsule is a group of neurons whose activity vector represents the instantiation parameters of a specific type of entity such as an object or an object part. They use the length of the activity vector to represent the probability that the entity exists and its orientation to represent the instantiation parameters. Active capsules at one level make predictions, via transformation matrices, for the instantiation parameters of higher-level capsules. When multiple predictions agree, a higher level capsule becomes active. They show that a discrimininatively trained, multi-layer capsule system achieves state-of-the-art performance on MNIST and is considerably better than a convolutional net at recognizing highly overlapping digits. To achieve these results they use an iterative routing-by-agreement mechanism: A lower-level capsule prefers to send its output to higher level capsules whose activity vectors have a big scalar product with the prediction coming from the lower-level capsule.

Chapter 3
Technology

CHAPTER 3

TECHNOLOGY

"I think, therefore I am"

René Descartes,
French philosopher and scientist

Deep learning frameworks offer flexibility with designing and training custom deep neural networks and provide interfaces to common programming language. We used the following frameworks and technologies in our project.

3.1 Tensorflow

TensorFlow is an open source software library for high performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. Originally developed by researchers and engineers from the Google Brain team within Google's AI organization, it comes with strong support for machine learning and deep learning and the flexible numerical computation core is used across many other scientific domains. TensorFlow, as the name indicates, is a framework to define and run computations involving tensors. A tensor is a generalization of vectors and matrices to potentially higher dimensions. Internally, TensorFlow represents tensors as n-dimensional arrays of base datatypes. TensorFlow uses a dataflow graph to represent your computation in terms of the dependencies between individual operations. This leads to a low-level programming model in which you first define the dataflow graph, then create a TensorFlow session to run parts of the graph across a set of local and remote devices.

We use Tensorflow when we need access to low level API such as metric functions or auto gradient functions.

3.2 Keras

Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation. It puts user experience front and center. Keras follows best practices for reducing cognitive load: it offers consistent & simple APIs, it minimizes the number of user actions required for common use cases, and it provides clear and actionable feedback upon user error. A model is understood as a sequence or a graph of standalone, fully-configurable modules that can be plugged together with as little restrictions as possible. In particular, neural layers, cost functions, optimizers, initialization schemes, activation functions, regularization schemes are all standalone modules that you can combine to create new models. Can easily create new modules allows for total expressiveness, making Keras suitable for advanced research.

We use Keras as our primary Deep learning library. Most of our code uses keras.

3.3 PyTorch

PyTorch is a relatively new framework. PyTorch provides Tensors that can live either on the CPU or the GPU, and accelerate compute by a huge amount. It also provides a wide variety of tensor routines to accelerate and fit your scientific computation needs such as slicing, indexing, math operations, linear algebra, reductions. PyTorch has a unique way of building neural networks: using and replaying a tape recorder. Most frameworks such as TensorFlow, Theano, Caffe and CNTK have a static view of the world. One has to build a neural network, and reuse the same structure again and again. Changing the way the network behaves means that one has to start from scratch. With PyTorch, it uses a technique called Reverse-mode auto-differentiation, which allows you to change the way your network behaves arbitrarily with zero lag or overhead. Its inspiration comes from several research papers on this topic, as well as current and past work such as autograd, autograd, Chainer, etc. While this technique is not unique to PyTorch, it's one of the fastest implementations of it to date.

We have a simple proof of concept implementation of our project in PyTorch, which can easily be extended to other GANs.

3.4 Google Colaboratory

Colaboratory is a research tool for machine learning education and research. Colaboratory is a Google research project created to help disseminate machine learning education and research. It's a Jupyter notebook environment that requires no setup to use and runs entirely in the cloud. We can use GPU as a backend for free for 12 hours at a time. The GPU used in the backend is Nvidia Tesla K80. Colaboratory notebooks are stored in Google Drive and can be shared just as you would with Google Docs or Sheets. Colaboratory supports both Python2 and Python3 for code execution. It has Intel Xeon 2vCPU running at 2.2 GHz, 13 GB RAM and 33 GB storage space. We used Google Colaboratory extensively for our project. We trained and tested all of our models on colaboratory. It provides an average 10 times speed-up than running on a local machine.

3.5 Matplotlib

Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shells, the Jupyter notebook, web application servers, and four graphical user interface toolkits. We made use of Matplotlib to visualize the various graphs. The output images and data of the training were also obtained Matplotlib.

3.6 Flask

Flask is a micro web framework written in Python and based on the Werkzeug toolkit and Jinja2 template engine. We used Flask to port our python demonstration code onto a webapp. It forms an intermediary between the python code on the server and the front-end HTML and JavaScript.

3.7 OpenCV

OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products. Being a BSD-licensed product, OpenCV makes it easy for businesses to utilize and modify the code.

The library has more than 2500 optimized algorithms, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. These algorithms can be used to detect and recognize faces, identify objects, classify human actions in videos, track camera movements, track moving objects, extract 3D models of objects, produce 3D point clouds from stereo cameras, stitch images together to produce a high resolution image of an entire scene, find similar images from an image database, remove red eyes from images taken using flash, follow eye movements, recognize scenery and establish markers to overlay it with augmented reality, etc.

CHAPTER 4
SYSTEM REQUIREMENTS

CHAPTER 4

SYSTEM REQUIREMENTS

"Any A.I. smart enough to pass a Turing test is smart enough to know to fail it."

Ian McDonald,
River of Gods

4.1 Functional Requirements

The system must, at the minimum, fulfill certain basic requirements.

1. Take a latent space vector or noise as input.
2. Learn joint probability distribution of training images.
3. Generate a realistic image of the training distribution that doesn't belong to training set.
4. Use a binary Capsule network classifier to differentiates real or fake image.

4.2 Non Functional Requirements

The system should have following non functional requirements. These specify the quality of the system.

1. Efficient, Fast forward and backward propagation.
2. Curb data usage while maintaining high quality results.

4.3 Development Requirement

System requirements for training and demonstrating of the models

4.3.1 Training System Requirements

For training the model,

1. python 3.4 or above
2. Tensorflow 1.7 or above
3. Keras 2.1.6 or above
4. Numpy, Scipy, Matplotlib

5. OpenCV 2
6. CPU 1.6 GHz or above
7. Nvdia GPU with CUDA compatibility 3.5 or above
8. RAM 8GB or above
9. Disk storage 20GB or above

4.3.2 Demonstration System Requirements

For evaluating the model or just forward-propagation,

1. python 3.4 or above
2. Flask
3. Browser - Google Chrome
4. Tensorflow 1.7 or above
5. Keras 2.1.6 or above
6. Numpy, Scipy, Matplotlib
7. OpenCV 2
8. CPU 1 GHz or above
9. RAM 4GB or above
10. Disk storage 20GB or above

Chapter 5
Proposed Architecture

CHAPTER 5

PROPOSED ARCHITECTURE

"Artificial intelligence, in fact, is obviously an intelligence transmitted by conscious subjects, an intelligence placed in equipment."

Pope Benedict XVI

The generator will use noise as input to generate faces. We will use random data as this noise. This ensures the data is unique and across the spectrum while retaining a normal distribution.

The CapsNet making up the discriminator consists of a small convolutional network to convert low level data in the form of pixels into an artifact called "pose". These poses can be anything, like nose, ear, eye, etc. These poses are then passed on as input to the later lower layers consisting of components called Capsules. A capsule is analogous to the human brain containing different modules to handle different tasks. The brain has a mechanism to route the information among the modules, to reach the best modules that can handle the information.

A capsule is a nested set of neural layers. Each capsule is able to handle one particular pose and communicate its calculation to other capsules which can use that calculation. This calculation is in the form of a probability prediction of the current pose that takes place in its logistic unit. This working is fundamentally different from convolutional networks, which utilizes Max Pooling. Max pooling selects the most active input node from the next layer to pass on the information. CapsNet on the other hand selects the next capsule based on which capsule would be capable of handling that information. This is called Dynamic routing. This results in invariance of information to the position and orientation of features in an object while ignoring the invariance in very low level features as, at the pixel level, this does not matter.

5.1 Generator

We use a deep convolutional generator model, which is similar to what DCGAN uses as generator. It starts with a latent vector or noise of shape 100 which connects to a densely connected neural layer. We then use Reshape layer which reshapes it to an (8, 8, 128) matrix and which is then sent to the batch normalization layer. We later perform up-sampling by using DeConv (De-Convolutional layer: transposed convolutional layer) layer. DeConv layer is internally implemented by an up-sampling layer and a convolutional layer. We perform DeConv two more times to get the shape of the image as expected, which should be (64, 64, 3), which stands for a 64x64 RGB image.

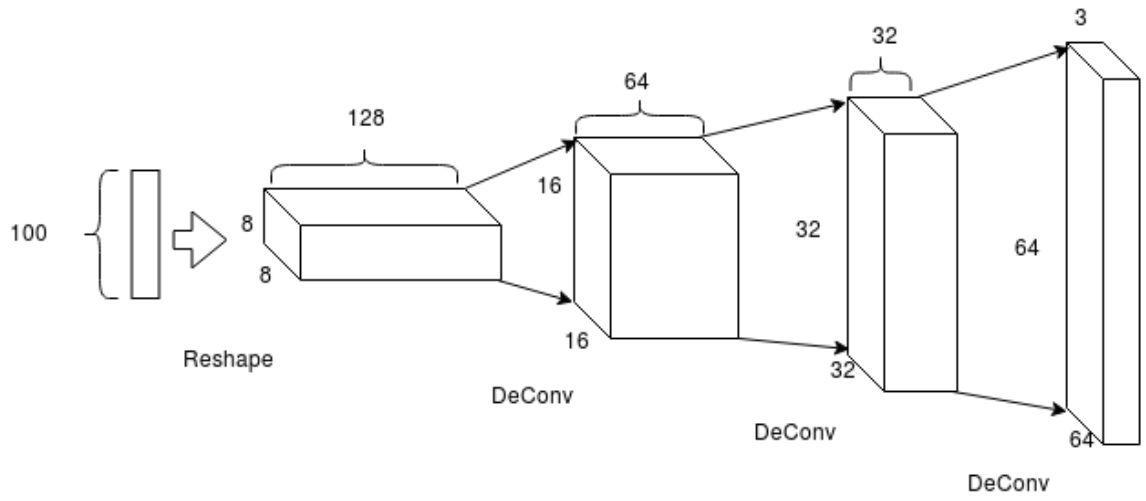


Figure 5.1: Generator architecture

5.2 Discriminator

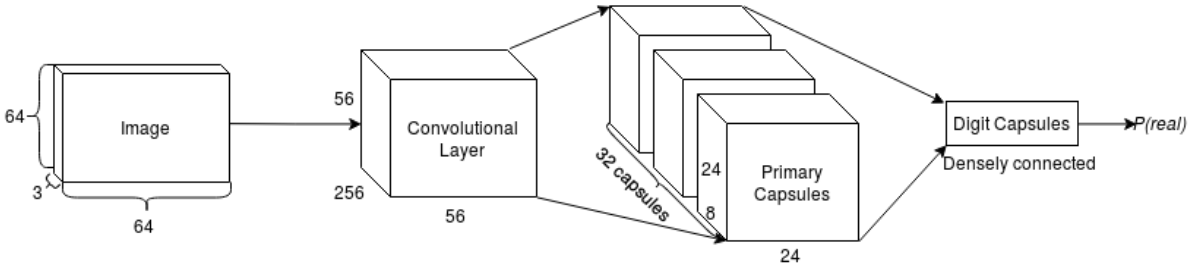


Figure 5.2: Discriminator architecture

The discriminator in the original models is replaced by our modification of CapsNet. We use a binary classifier CapsNet to distinguish between real and fake images.

The CapsNet has 2 parts: encoder and decoder. The first 3 layers are encoder, and the second 3 are decoder:

Layer 1. Convolutional layer Layer 2. PrimaryCaps layer Layer 3. DigitCaps layer Layer 4. Fully connected #1 Layer 5. Fully connected #2 Layer 6. Fully connected #3

In our implementation, we do not use the decoder layers as we do not need the reconstruction aspects of the network for classification. Hence only the encoder layers are used.

The encoding part of the network takes as input a digital image of 64x64 and learns to encode it into a vector of 16 dimensions of instantiation parameters, this is where the capsules do their job. The output of the network during prediction is a 10-dimensional vector of the lengths of the DigitCaps' outputs.

5.2.1 Layer 1: Convolutional Layer

Input: 64x64 image (three color channel).
Output: 56x56x256 tensor.
Number of parameters: 62464.

The work of the convolutional layer consists of detecting the basic functions in the 2D image. In the CapsNet system, the convolutional layer has 256 kernels of size 9x9x1 and stride of 1, followed by the activation function, ReLU. To calculate the number of parameters, we must also remember that each kernel in a convolutional layer has 1 bias term. Therefore, this layer has (9 x 9 x 3 x 256 + 256 =) 62464 trainable parameters in total.

5.2.2 Layer 2: PrimaryCaps Layer

Input: 56x56x256 tensor.
Output: 24x24x8x32 tensor.
Number of parameters: 5308672.

This layer has 32 primary capsules whose job is to take basic features detected by the convolutional layer and produce combinations of the features. The layer has 32 "primary capsules" that are very similar to convolutional layer in their nature. Each capsule applies eight 9x9x256 convolutional kernels to the 56x56x256 input volume and therefore produces 24x24x8 output tensor. Since there are 32 such capsules, the output volume has shape of 24x24x8x32. Doing calculation similar to the one in the previous layer, we get (9 x 9 x 256 x 256 + 256 =) 5308672 trainable parameters in this layer.

5.2.3 Layer 3: DigitCaps Layer

Input: 24x24x8x32 tensor.
Flattened to: 147456
Output: 1x1 matrix.
Number of parameters: 1497600.

We have a 3 hidden-layer densely connected neural network which takes the flattened input to give a binary classification output of 1x1. Each hidden layer consists of 160 neurons. From the flattened input we get the input of size 147456 which is connected densely to the first hidden layer. Thus the weights in the first layer turn out to be (147456 x 160 + 160 =) 23593120 in number. The first hidden layer is connected to the second hidden layer densely, similarly the second and the third. Therefore there are (160 x 160 + 160 =) 25760 trainable parameters for the second and third hidden layers.

The last last hidden layer is connected to a output layer with one neuron, which essentially gives the binary classification result. Its parameters are (160 x 1 + 1 =) 161 in number.

Chapter 6
Implementation

CHAPTER 6

IMPLEMENTATION

"The portion of evolution in which animals developed eyes was a big development. Now computers have eyes."

Jeff Dean,
Lead of Google Brain

The first step is to implement the state-of-the-art in image regeneration to guage the improvements. We use DCGAN to start of with. The results of the training and testing will be recorded to compare it with the results of our CapsNet-based approach later. We will be using CapsNet as the underlying technology to implement our GAN (CapsGAN). The goal is to replace the CNN inside DCGAN with CapsNet and compare the results. The GAN internally consists of two components - a generator and a discriminator - which we build out of CapsNet. The discriminator is initially trained separately to distinguish real and fake data, and later they work together to improve upon their performance by acting as adversaries.

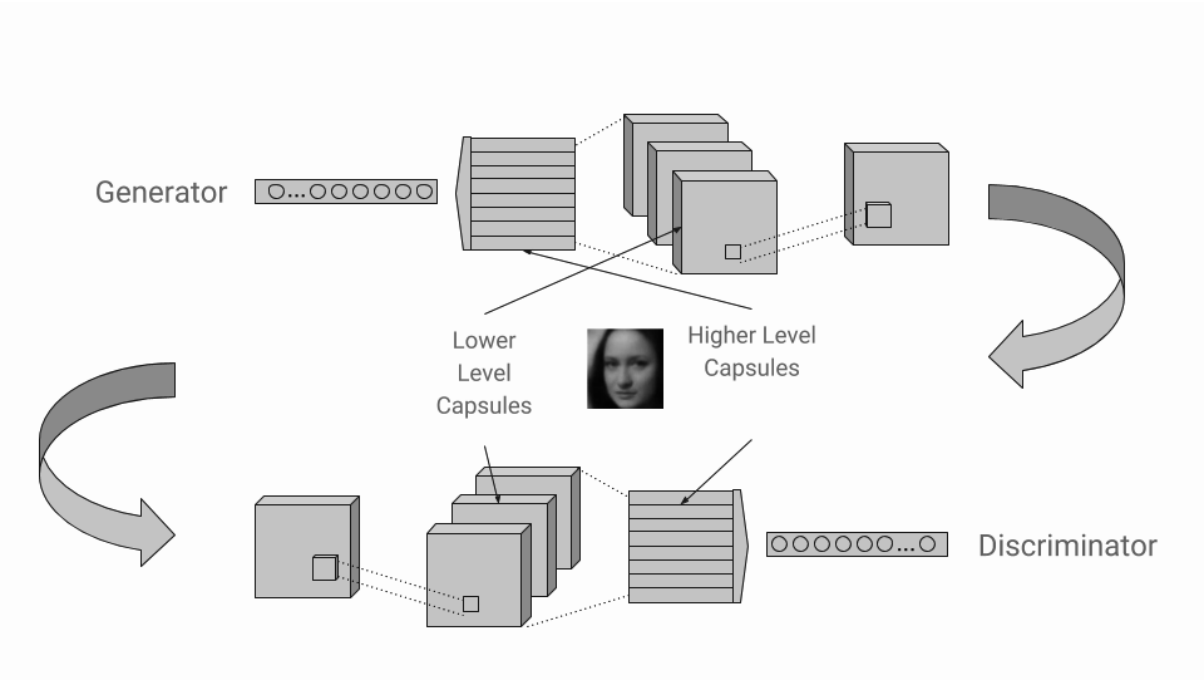


Figure 6.1: Proposed architecture

During the course of our research we were forced to conclude that building a CapsNet based generator was not feasible. A fundamental aspect of CapsNet is dynamic routing which is not possible to replicate in the generator, that is, dynamic routing cannot be inverted. Hence we implemented just the discriminator in CapsNet.

We concentrated on four networks: DCGAN, WGAN, ACGAN and InfoGAN. Our training laboratory was Colloboratory - the cloud machine learning research platform.

We trained each network individually for 20,000 epochs each. For our preliminary training we used the MNIST dataset. The MNIST dataset is a large dataset of handwritten digits commonly used for image processing training tasks. Each of the networks had it's discriminator augmented with the CapsNet code. The networks with the CapsNet discriminator were then individually trained on the same dataset, for 20,000 epochs each. Overall, it took us a few days to train all the networks and gather all the data.

6.1 Network Architectures

The following are the network architectures of the generator and the discriminator.

6.1.1 Generator

Layer (type)	Output Shape	Param #
dense_5 (Dense)	(None, 8192)	827392
reshape_1 (Reshape)	(None, 8, 8, 128)	0
batch_normalization_3 (Batch	(None, 8, 8, 128)	512
up_sampling2d_1 (UpSampling2	(None, 16, 16, 128)	0
conv2d_1 (Conv2D)	(None, 16, 16, 128)	147584
activation_1 (Activation)	(None, 16, 16, 128)	0
batch_normalization_4 (Batch	(None, 16, 16, 128)	512
up_sampling2d_2 (UpSampling2	(None, 32, 32, 128)	0
conv2d_2 (Conv2D)	(None, 32, 32, 64)	73792
activation_2 (Activation)	(None, 32, 32, 64)	0
batch_normalization_5 (Batch	(None, 32, 32, 64)	256
up_sampling2d_3 (UpSampling2	(None, 64, 64, 64)	0
conv2d_3 (Conv2D)	(None, 64, 64, 32)	18464
activation_3 (Activation)	(None, 64, 64, 32)	0
batch_normalization_6 (Batch	(None, 64, 64, 32)	128
conv2d_4 (Conv2D)	(None, 64, 64, 3)	867
activation_4 (Activation)	(None, 64, 64, 3)	0

Total params: 1,069,507
Trainable params: 1,068,803
Non-trainable params: 704

6.1.2 Discriminator

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 64, 64, 3)	0	
conv1 (Conv2D)	(None, 56, 56, 256)	62464	input_1[0][0]
leaky_re_lu_1 (LeakyReLU)	(None, 56, 56, 256)	0	conv1[0][0]

```
batch_normalization_1 (BatchNor (None, 56, 56, 256)  1024        leaky_re_lu_1[0][0]
__________________________________________________________________________________________________
primarycap_conv2 (Conv2D)       (None, 24, 24, 256)  5308672     batch_normalization_1[0][0]
__________________________________________________________________________________________________
primarycap_reshape (Reshape)    (None, 18432, 8)     0           primarycap_conv2[0][0]
__________________________________________________________________________________________________
primarycap_squash (Lambda)      (None, 18432, 8)     0           primarycap_reshape[0][0]
__________________________________________________________________________________________________
batch_normalization_2 (BatchNor (None, 18432, 8)     32          primarycap_squash[0][0]
__________________________________________________________________________________________________
flatten_1 (Flatten)             (None, 147456)       0           batch_normalization_2[0][0]
__________________________________________________________________________________________________
uhat_digitcaps (Dense)          (None, 160)          23593120    flatten_1[0][0]
__________________________________________________________________________________________________
softmax_digitcaps1 (Activation) (None, 160)          0           uhat_digitcaps[0][0]
__________________________________________________________________________________________________
dense_1 (Dense)                 (None, 160)          25760       softmax_digitcaps1[0][0]
__________________________________________________________________________________________________
multiply_1 (Multiply)           (None, 160)          0           uhat_digitcaps[0][0]
                                                                 dense_1[0][0]
__________________________________________________________________________________________________
leaky_re_lu_2 (LeakyReLU)       (None, 160)          0           multiply_1[0][0]
__________________________________________________________________________________________________
softmax_digitcaps2 (Activation) (None, 160)          0           leaky_re_lu_2[0][0]
__________________________________________________________________________________________________
dense_2 (Dense)                 (None, 160)          25760       softmax_digitcaps2[0][0]
__________________________________________________________________________________________________
multiply_2 (Multiply)           (None, 160)          0           uhat_digitcaps[0][0]
                                                                 dense_2[0][0]
__________________________________________________________________________________________________
leaky_re_lu_3 (LeakyReLU)       (None, 160)          0           multiply_2[0][0]
__________________________________________________________________________________________________
softmax_digitcaps3 (Activation) (None, 160)          0           leaky_re_lu_3[0][0]
__________________________________________________________________________________________________
dense_3 (Dense)                 (None, 160)          25760       softmax_digitcaps3[0][0]
__________________________________________________________________________________________________
multiply_3 (Multiply)           (None, 160)          0           uhat_digitcaps[0][0]
                                                                 dense_3[0][0]
__________________________________________________________________________________________________
leaky_re_lu_4 (LeakyReLU)       (None, 160)          0           multiply_3[0][0]
__________________________________________________________________________________________________
dense_4 (Dense)                 (None, 1)            161         leaky_re_lu_4[0][0]
==================================================================================================
Total params: 29,042,753
Trainable params: 29,042,225
Non-trainable params: 528
__________________________________________________________________________________________________
```

6.2 Demonstration

The training of the models took place on Colaboratory, over Keras. Keras provided for a fast implementation of the code and high level abstraction. The output model was in the format of H5. This could not be directly used as part of semantic inpainting as semantic inpainting requires changing low level architectural details which Keras does not allow. Hence we converted the H5 model to TensorFlow protocol buffer, which is TensorFlow's model saving format, the code for which can be found in 7.2.1.

For the semantic inpainting, we take as input an image. The image is loaded onto a canvas with an ability to mask a part of the image. Once the mask is set, our processing starts. The following process is based on Raymond A. Yeh *et al.* [9]

The masked image is a Hadamard product of the mask component, M, and the original input image, y.

$$MaskedImage = \boldsymbol{M} \odot \boldsymbol{y} \tag{6.1}$$

Suppose we've found an image from the generator $G(\hat{z})$ for some $\hat{z}$ that gives a reasonable reconstruction of the missing portions. The completed pixels $(1 - M) \odot$

$G(\hat{z})$ can be added to the original pixels to create the reconstructed image:

$$\boldsymbol{x}_{\text{reconstructed}} = \boldsymbol{M} \odot \boldsymbol{y} + (1 - \boldsymbol{M}) \odot \boldsymbol{G}(\hat{z}) \tag{6.2}$$

Now all that is needed is to find a $G(\hat{z})$ that does a good enough job of completing the image. We will consider a loss function, a smaller value of which means that z is more suitable for completion. The total loss function will be a sum of two loss functions: Contextual and Perceptual.

6.2.1 Contextual Loss

To keep the same context as the input image, make sure the known pixel locations in the input image y are similar to the pixels in $G(z)$. We need to penalize $G(z)$ for not creating a similar image for the pixels that we know about. Formally, we do this by element-wise subtracting the pixels in y from $G(z)$ and looking at how much they differ:

$$\boldsymbol{L}_{\text{contextual}}(\boldsymbol{z}) = ||\boldsymbol{M} \odot \boldsymbol{G}(\boldsymbol{m}) + (1 - \boldsymbol{M}) \odot \boldsymbol{G}(\hat{z})||_1 \tag{6.3}$$

where $||x||_1 = \sum_i |x_i|$ is the l_1 norm of some vector x.

6.2.2 Perceptual Loss

To recover an image that looks real, let's make sure the discriminator is properly convinced that the image looks real. We'll do this with the same criterion used in training the network:

$$\boldsymbol{L}_{\text{perceptual}}(\boldsymbol{z}) = \boldsymbol{log}(1 - \boldsymbol{D}(\boldsymbol{G}(\boldsymbol{z}))) \tag{6.4}$$

6.2.3 Total Loss

We now find $\hat{z}$ with a combination of the contextual and perceptual losses:

$$\boldsymbol{L}(\boldsymbol{z}) = \boldsymbol{L}_{\text{contextual}}(\boldsymbol{z}) + \lambda \boldsymbol{L}_{\text{perceptual}}(\boldsymbol{z}) \tag{6.5}$$

$$\hat{z} = arg \min_z \boldsymbol{L}(\boldsymbol{z}) \tag{6.6}$$

where λ is a hyper-parameter that controls how important the contextual loss is relative to the perceptual loss. Then as before, the reconstructed image fills in the missing values of $\boldsymbol{y}$ with $\boldsymbol{G}(\boldsymbol{z})$:

$$\boldsymbol{x}_{\text{reconstructed}} = \boldsymbol{M} \odot \boldsymbol{y} + (1 - \boldsymbol{M}) \odot \boldsymbol{G}(\hat{z}) \tag{6.7}$$

6.2.4 Projected Gradient Descent

For minimizing the loss function we use projected gradient descent. Its different from gradient descent in the sense, at a basic level, projected gradient descent is just a more general method for solving a more general problem. Gradient descent minimizes a function by moving in the negative gradient direction at each step. There is no constraint on the variable.

$$\text{Problem 1: } \min_x f(x)$$

$$x_{k+1} = x_k - t_k \nabla f(x_k) \tag{6.8}$$

On the other hand, projected gradient descent minimizes a function subject to a constraint. At each step we move in the direction of the negative gradient, and then "project" onto the feasible set.

$$\text{Problem 2: } \min_x f(x) \text{ subject to } x \in C$$

$$y_{k+1} = x_k - t_k \nabla f(x_k)$$

$$x_{k+1} = \arg\min_{x \in C} \|y_{k+1} - x\| \tag{6.9}$$

Chapter 7
Code Snippets

CHAPTER 7

CODE SNIPPETS

"By far the greatest danger of Artificial Intelligence is that people conclude too early that they understand it."

Eliezer Yudkowsky,
Machine Intelligence Research Institute

7.1 GAN Training

The internal architectures of all four GANs are similarly designed. Here we take DCGAN to showcase the code.

The main code consists of a class (DCGAN) which contains the following six functions:

1. Initialization: Calls build_generator and build_discriminator and makes a combined model
2. Build_Generator: Creates a generator model
3. Build_Discriminator: Creates a discriminator model
4. Train: Takes the input images and starts training, prints training progress with metrics
5. Save_Imgs: Saves a grid of generated images at specific epochs
6. Save_models: Saves the current model to disk

7.1.1 Initialization

```
def __init__(self):
    # Input shape
    self.img_rows = 64
    self.img_cols = 64
    self.channels = 3
    self.img_shape = (self.img_rows, self.img_cols, self.channels)
    self.latent_dim = 100

    optimizer = Adam(0.0002, 0.5)

    # Build and compile the discriminator
    self.discriminator = self.build_discriminator()
    self.discriminator.compile(loss='binary_crossentropy',
        optimizer=optimizer,
        metrics=['accuracy'])

    # Build the generator
    self.generator = self.build_generator()
```

```
    # The generator takes noise as input and generates imgs
    z = Input(shape=(100,))
    img = self.generator(z)

    # For the combined model only train the generator
    self.discriminator.trainable = False

    # The discriminator takes generated images as
    # input and determines validity
    valid = self.discriminator(img)

    # The combined model  (stacked generator and discriminator)
    self.combined = Model(z, valid)
    self.combined.compile(loss='binary_crossentropy', optimizer=optimizer)
```

7.1.2 Building Generator

```
def build_generator(self):
    """
    Build generator which takes noise (a tensor of size 100) as input,
    and produces an RGB image of size (64 x 64) .
    """

    # Create a model in which one can add layers sequentially
    model = Sequential()

    # Add a densely connected layer to the model,
    # activation function of ReLu
    model.add(Dense(128 * 8 * 8, activation="relu",
        input_shape=(self.latent_dim,)))
    model.add(Reshape((8, 8, 128)))

    # DeConv layer one starts
    model.add(BatchNormalization(momentum=0.8))
    model.add(UpSampling2D())
    model.add(Conv2D(128, kernel_size=3, padding="same"))
    model.add(Activation("relu"))

    # DeConv layer two starts
    model.add(BatchNormalization(momentum=0.8))
    model.add(UpSampling2D())
    model.add(Conv2D(64, kernel_size=3, padding="same"))
    model.add(Activation("relu"))

    # DeConv layer three starts
    model.add(BatchNormalization(momentum=0.8))
    model.add(UpSampling2D())
    model.add(Conv2D(32, kernel_size=3, padding="same"))
    model.add(Activation("relu"))

    # Final output layer
    model.add(BatchNormalization(momentum=0.8))
    model.add(Conv2D(self.channels, kernel_size=3, padding="same"))
    model.add(Activation("tanh"))

    # Print model summary
    model.summary()

    # Use functional API to make model
    noise = Input(shape=(self.latent_dim,))
    img = model(noise)

    # Return functional model
    return Model(noise, img)
```

7.1.3 Building Discriminator

```
def build_discriminator(self):
    """
    Discriminator takes real/generated images and outputs its prediction.
    """

    # Define input shape for our network
    img = Input(shape=self.img_shape)
```

```
    # First ConvLayer outputs a 56x56x256 matrix
    x = Conv2D(filters=256, kernel_size=9, strides=1,
        padding='valid', name='conv1')(img)
    x = LeakyReLU()(x)
    x = BatchNormalization(momentum=0.8)(x)

    # Capsule architecture starts

    # First layer: PrimaryCaps
    x = Conv2D(filters=8 * 32, kernel_size=9, strides=2,
        padding='valid', name='primarycap_conv2')(x)

    # Primary capsule has collections of activations which denote orientation
    # while intensity of the vector which denotes the presence of the digit)
    x = Reshape(target_shape=[-1, 8], name='primarycap_reshape')(x)

    # Output a number between 0 and 1 for each capsule
    # where the length of the input decides the amount
    x = Lambda(squash, name='primarycap_squash')(x)
    x = BatchNormalization(momentum=0.8)(x)

    # Second layer: DigitCaps
    # This is a modified form of the standard CapsNet DigitCaps architecture
    # where we have replaced the multiple capsules with a single capsule of
    # densely connected neural network.
    x = Flatten()(x)

    # Dynamic Routing
    # uhat = prediction vector, u * w
    # w = weight matrix but will act as a dense layer
    # u = output from a previous layer
    uhat = Dense(160, kernel_initializer='he_normal',
        bias_initializer='zeros', name='uhat_digitcaps')(x)

    # softmax will make sure that each weight c_ij is a non-negative number
    # and their sum equals to one
    c = Activation('softmax', name='softmax_digitcaps1')(uhat)

    # s_j (output of the current capsule level) = uhat * c
    c = Dense(160)(c) # compute s_j
    x = Multiply()([uhat, c])

    # Squashing the capsule outputs creates severe blurry artifacts,
    # thus we replace it with Leaky ReLu.
    s_j = LeakyReLU()(x)


    c = Activation('softmax', name='softmax_digitcaps2')(s_j)
    c = Dense(160)(c)
    x = Multiply()([uhat, c])
    s_j = LeakyReLU()(x)

    c = Activation('softmax', name='softmax_digitcaps3')(s_j)
    c = Dense(160)(c)
    x = Multiply()([uhat, c])
    s_j = LeakyReLU()(x)

    # Final dense layer output a binary classification
    pred = Dense(1, activation='sigmoid')(s_j)
    return Model(img, pred)
```

7.1.4 Train

```
def train(self, epochs, batch_size=128, save_interval=50):

    half_batch = int(batch_size / 2)

    for epoch in range(epochs):
        # ---------------------
        #  Train Discriminator
        # ---------------------
        cnt=0
        train_datagen = ImageDataGenerator(rescale=1./255)
        train_generator = train_datagen.flow_from_directory('data',
            target_size=(64, 64),batch_size=half_batch,class_mode=None)
        for x in train_generator:
```

```
            # Sample noise and generate a half batch of new images
            noise = np.random.normal(0, 1, (half_batch, 100))
            gen_imgs = self.generator.predict(noise)

            # Train the discriminator
            # (real classified as ones and generated as zeros)
            d_loss_real = self.discriminator.train_on_batch
                (x, np.ones((half_batch, 1)))
            d_loss_fake = self.discriminator.train_on_batch
                (gen_imgs, np.zeros((half_batch, 1)))
            d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

            # ---------------------
            #  Train Generator
            # ---------------------

            # Sample generator input
            noise = np.random.normal(0, 1, (batch_size, 100))

            # Train the generator
            # (wants discriminator to mistake images as real)
            g_loss = self.combined.train_on_batch
                (noise, np.ones((batch_size, 1)))

            # Plot the progress
            if(cnt%save_interval==0):
              print ("%d [D loss: %f, acc.: %.2f%%] [G loss: %f]"
                % (epoch, d_loss[0], 100*d_loss[1], g_loss))

            # If at save interval => save generated image samples
            if(cnt%save_interval==0):
              self.save_imgs(cnt)
            cnt+=1
```

7.1.5 Saving Images

```
def save_imgs(self, epoch):
    r, c = 5, 5
    noise = np.random.normal(0, 1, (r * c, 100))
    gen_imgs = self.generator.predict(noise)

    # Rescale images 0 - 1
    gen_imgs = 0.5 * gen_imgs + 0.5

    fig, axs = plt.subplots(r, c)
    #fig.suptitle("DCGAN: Generated digits", fontsize=12)
    cnt = 0
    for i in range(r):
        for j in range(c):
            axs[i,j].imshow(gen_imgs[cnt, :,:,0], cmap='gray')
            axs[i,j].axis('off')
            cnt += 1
    fig.savefig("images/images_%d.png" % epoch)
    plt.close()
```

7.1.6 Saving Models

```
def save_model(self):

    def save(model, model_name):
        model_path = "saved_model/%s.json" % model_name
        weights_path = "saved_model/%s_weights.hdf5" % model_name
        options = {"file_arch": model_path,
                   "file_weight": weights_path}
        json_string = model.to_json()
        open(options['file_arch'], 'w').write(json_string)
        model.save_weights(options['file_weight'])

    save(self.generator, "generator")
    save(self.discriminator, "discriminator")
    save(self.combined, "adversarial")
```

7.2 Demonstration

7.2.1 Converting Models

This code converts the Keras H5 model to the TensorFlow protocol buffers.

```
def convert_to_pb(weight_file,json_file,input_fld='',output_fld=''):

    import os
    import os.path as osp
    from tensorflow.python.framework import graph_util
    from tensorflow.python.framework import graph_io
    from keras.models import model_from_json
    from keras import backend as K
    import tensorflow as tf

    # weight_file is a .hdf5 keras model file
    output_node_names_of_input_network = ["pred0"]
    output_node_names_of_final_network = 'output_node'

    # change filename to a .pb tensorflow file
    output_graph_name = json_file[:-4]+'pb'

    weight_file_path = osp.join(input_fld, weight_file)
    json_file_path=osp.join(input_fld,json_file)
    js_file = open(json_file_path, 'r')
    model_js= js_file.read()
    js_file.close()
    net_model = model_from_json(model_js)
    # load weights into new model
    net_model.load_weights(weight_file_path)


    num_output = len(output_node_names_of_input_network)
    pred = [None]*num_output
    pred_node_names = [None]*num_output

    for i in range(num_output):
        pred_node_names[i] = output_node_names_of_final_network+str(i)
        pred[i] = tf.identity(net_model.output[i], name=pred_node_names[i])

    sess = K.get_session()

    constant_graph = graph_util.convert_variables_to_constants(
        sess, sess.graph.as_graph_def(), pred_node_names)
    graph_io.write_graph(
        constant_graph, output_fld, output_graph_name, as_text=False)
    print('saved the constant graph (ready for inference) at: ',
         osp.join(output_fld, output_graph_name))

    return output_fld+output_graph_name


# tf_model_path = convert_to_pb(
# 'generator_weights.hdf5','generator.json','saved_model/','tf_model/')
```

7.2.2 Semantic Inpainting

This code implements semantic inpainting for demonstration purposes.

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import cv2

IMAGE_SHAPE=[28,28,1]
ITERATIONS=1000

gen_input_name="input_2:0"
gen_output_name="output_node0:0"
dis_input_name="input_1:0"
dis_output_name="output_node0:0"

momentum, lr =0.9,0.01
```

```
# read  graph definition from protocol buffer
gen_graph_def = tf.GraphDef()
with open('generator.pb', "rb") as f:
    gen_graph_def.ParseFromString(f.read())


z=tf.placeholder(tf.float32,shape=(1,100))
gen_imgs,=tf.import_graph_def(gen_graph_def,input_map={gen_input_name:z},
     return_elements=[gen_output_name])
print("loaded generator")

dis_graph_def = tf.GraphDef()
with open('discriminator.pb', "rb") as f:
    dis_graph_def.ParseFromString(f.read())
dis_imgs=tf.reshape(gen_imgs,shape=[1]+IMAGE_SHAPE)
print(dis_imgs.shape)


valid,=tf.import_graph_def(dis_graph_def,input_map={dis_input_name:dis_imgs},
     return_elements=[dis_output_name])
print("loaded discriminator")

mask = tf.placeholder(tf.float32, IMAGE_SHAPE, name='mask')
image = tf.placeholder(tf.float32, IMAGE_SHAPE, name='image')
contextual_loss = tf.reduce_sum(
    tf.contrib.layers.flatten(
        tf.abs(tf.multiply(mask, gen_imgs) - tf.multiply(mask, image))), 1)
perceptual_loss = tf.log(1-valid)
complete_loss = contextual_loss + 0.1*perceptual_loss
grad_complete_loss = tf.gradients(complete_loss, z)



with tf.Session() as sess:
    scale = 0.3
    mask_val=np.ones(IMAGE_SHAPE)
    l = int(28*scale)
    u = int(28*(1.0-scale))
    mask_val[l:u, l:u, :] = 0.0


    in_image=(cv2.imread('img_269.jpg',0))

    # expand dimension if its gray scale image
    if IMAGE_SHAPE[2]==1:
        in_image=in_image.reshape(IMAGE_SHAPE)

    in_image=(in_image.astype(np.float32)-127.5)/127.5

    masked_image=(1+np.multiply(in_image,mask_val))/2

    if IMAGE_SHAPE[2]==1:
        masked_image=masked_image[:,:,0]

    cv2.imwrite('front/image.png',masked_image*255)

    zhats=np.random.normal(0, 1, (1, 100))
    v=0
    for i in range(ITERATIONS):
        fd={
        mask:mask_val,
        image:in_image,
        z:zhats
        }
        outputs=[complete_loss, grad_complete_loss, gen_imgs]
        loss,grad,g_imgs=sess.run(outputs,feed_dict=fd)
        v_prev = np.copy(v)
        v = momentum*v - lr*grad[0]
        zhats += -momentum * v_prev + (1+momentum)*v
        zhats = np.clip(zhats, 0, 1)
        if(i%10==0):
            print("Iteration {}".format(i))

        # write current status
        completed_img=np.multiply(g_imgs,1-mask_val)+
```

```
            np.multiply(in_image,mask_val)
        completed_img=(1+completed_img)*127.5

        if IMAGE_SHAPE[2]==1:
            completed_img=completed_img[:,:,0]

        cv2.imwrite('front/image.png',completed_img)

    g_imgs=sess.run(gen_imgs,feed_dict={z:zhats})

    completed_img=np.multiply(g_imgs,1-mask_val)+
        np.multiply(in_image,mask_val)
    completed_img=(1+completed_img)/2

    original_image=(1+in_image)/2

    cmap=None

    if IMAGE_SHAPE[2]==1:
        completed_img=completed_img[:,:,0]
        original_image=original_image[:,:,0]
        cmap='gray'


    fig, axs = plt.subplots(1, 3)
    axs[0].set_title("Original")
    axs[0].imshow(original_image,cmap=cmap)
    axs[0].axis('off')
    axs[1].set_title("Masked")
    axs[1].imshow(masked_image,cmap=cmap)
    axs[1].axis('off')
    axs[2].set_title("Completed")
    axs[2].imshow(completed_img,cmap=cmap)
    axs[2].axis('off')
    plt.show()
```

Chapter 8
Execution and Results

CHAPTER 8

EXECUTION AND RESULTS

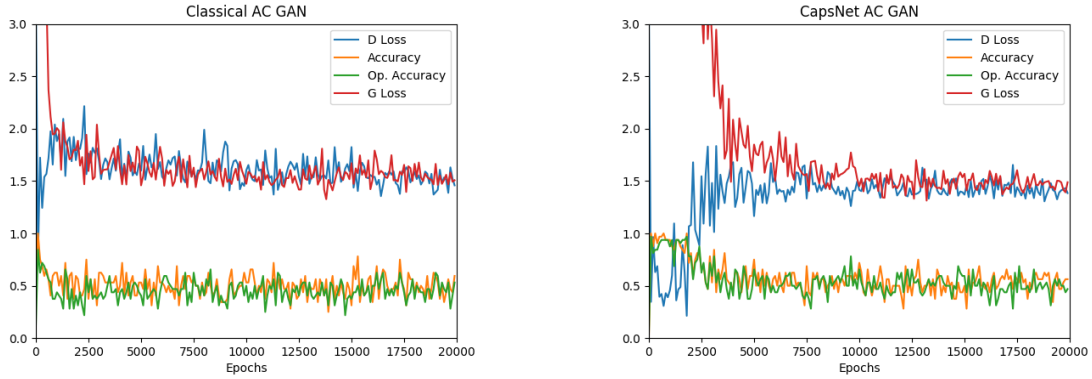
" We dont have better algorithms, we just have more data"

Peter Norvig,
Director of Research at Google Inc.

8.1 Training

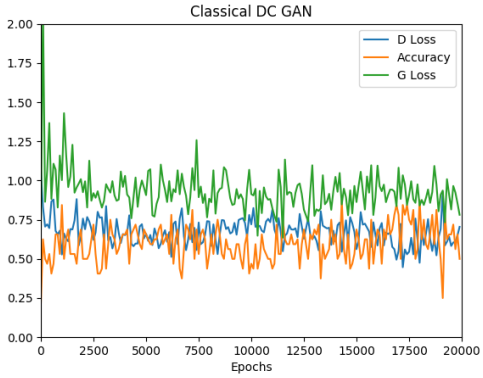
Our research consisted of four networks: ACGAN, DCGAN, InfoGAN and WGAN. Each of the networks was trained for 20,000 epochs each. The training period for the individual networks took anywhere between a few hours and a couple of days. For each network, we logged a few key metrics, mainly the Generator Loss (GLoss), Discriminator Loss (DLoss) and the Accuracy (Acc).

Below we show graphically the performance of the (four) classical networks as compared to the CapsNet discriminator augmented networks.

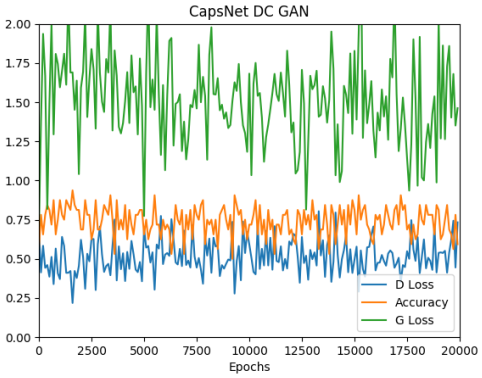


(a) Classical ACGAN (b) CapsNet augmented ACGAN

Figure 8.1: ACGAN metrics

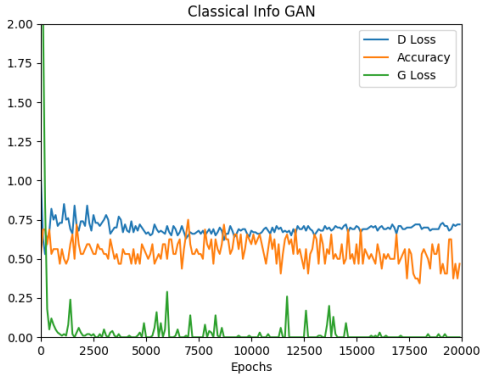


(a) Classical DCGAN

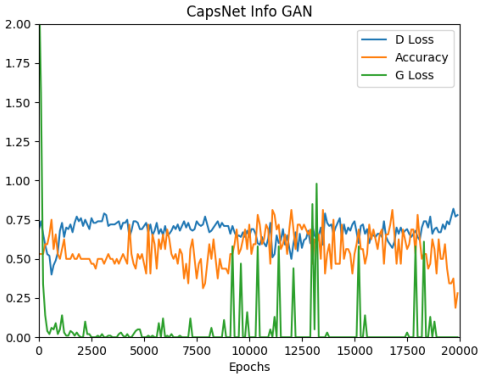


(b) CapsNet augmented DCGAN

Figure 8.2: DCGAN metrics

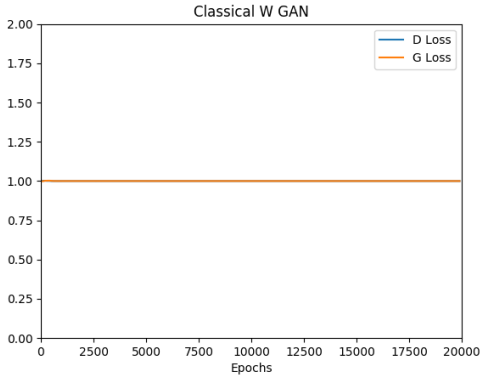


(a) Classical InfoGAN

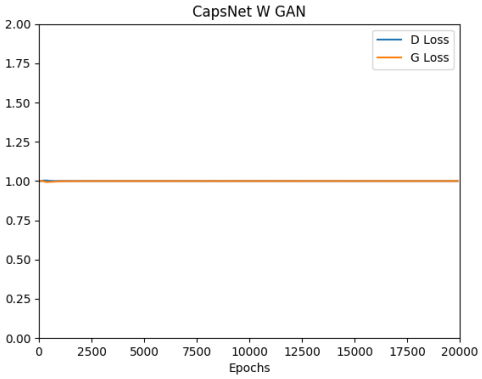


(b) CapsNet augmented InfoGAN

Figure 8.3: InfoGAN metrics



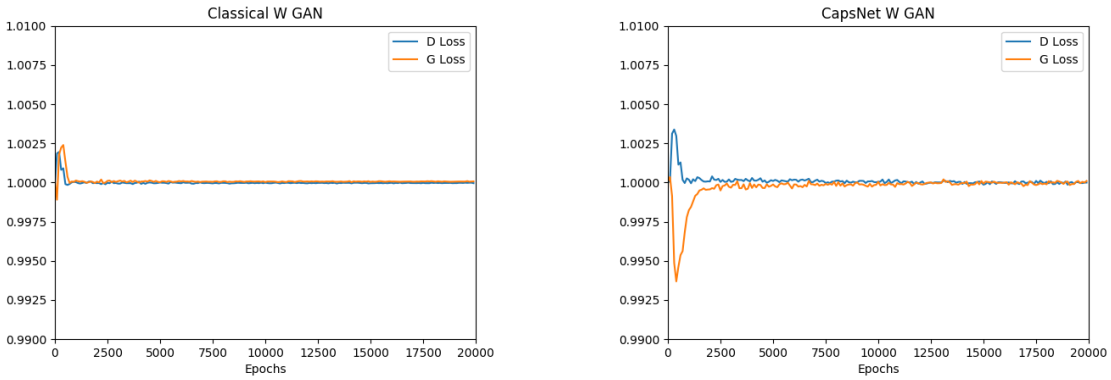
(a) Classical WGAN



(b) CapsNet augmented WGAN

Figure 8.4: WGAN metrics

The metrics of WGAN are at a smaller scale, so below is a comparison at the lower scale.



(a) Classical WGAN (b) CapsNet augmented WGAN

Figure 8.5: WGAN metrics - zoomed

A preliminarily analysis of the data shows us that our CapsNet augmented networks, hence-forth referred to as the network name prefixed with "Caps", perform comparably with the classical architectures.

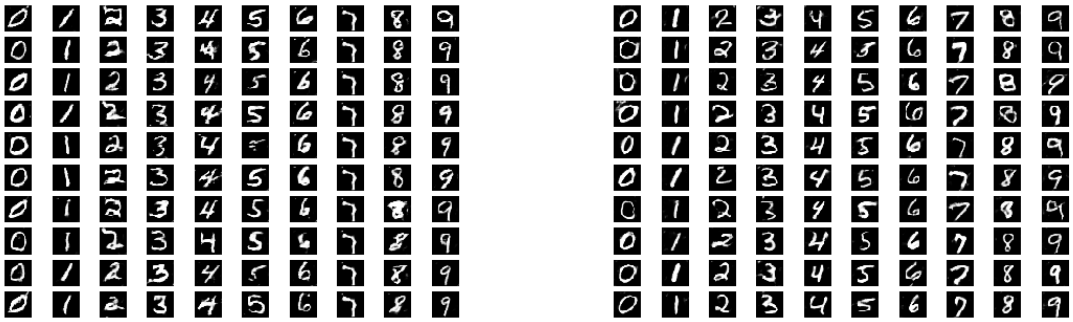
Under ACGAN we see that CapsACGAN starts off with very high variance in GLoss and DLoss. Over the course of 20,000 epochs, the variance gradually reduces to match the Classical ACGAN metrics at the end. Accuracy of CapsACGAN, on the other hand, quickly stabilizes to meet classical ACGAN metrics. As a side note, ACGAN was the fastest trained network, taking roughly three hours to complete 20,000 epochs.

DCGAN and InfoGAN are interesting in the sense that they show a remarkable shift in the GLoss metric. Even though it seems as if augmenting DCGAN with CapsNet discriminator leads to increase in GLoss, A closer look reveals that increased variation leads to faster learning of the generator network. Accuracy and DLoss follow their classical counterpart closely.

WGAN happens to be the best and state-of-the-art. Consequently, the variance in the metrics were at a much smaller scale, that is, the network is designed to be more stabilized and balanced but this also leads to slower learning, we can see that adding CapsNet discriminator speeds this up by adding slightly more variance while the network is still balanced. We had to zoom-in to notice the difference between the metrics. We see that an initial burst of high variance quickly stabilizes to quickly trace a path closely matching the classical architecture.

8.2 Generation

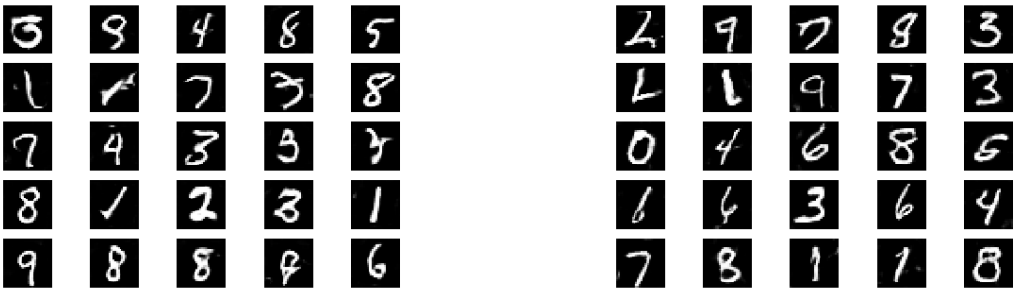
At the end of every epoch we saved the outputs of the generator. Here we show the outputs of generator of four networks after 20,000 epochs for comparison.



(a) Classical ACGAN (b) CapsNet augmented ACGAN

Figure 8.6: ACGAN outputs

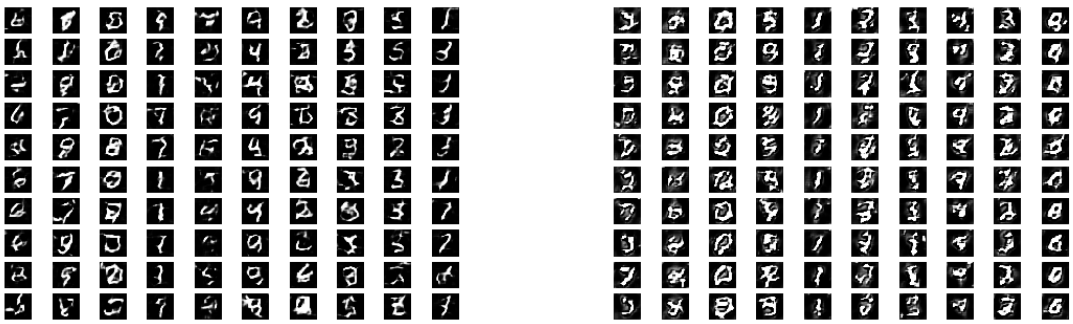
As expected from the metrics the ACGAN and CapsACGAN produce very similar outputs.



(a) Classical DCGAN (b) CapsNet augmented DCGAN

Figure 8.7: DCGAN outputs

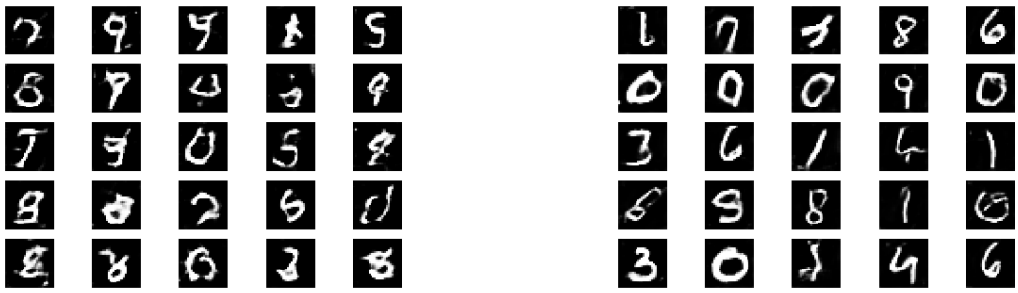
The outputs of DCGAN and CapsDCGAN are almost indistinguishable but a closer look reveals that CapsDCGAN produces more clear outputs and more percentage of CapsDCGAN outputs look real.



(a) Classical InfoGAN (b) CapsNet augmented InfoGAN

Figure 8.8: InfoGAN outputs

Even though the CapsInfoGAN output looks structurally more better and has learned faster but it is more noisier than that of the InfoGAN.



(a) Classical WGAN (b) CapsNet augmented WGAN

Figure 8.9: WGAN outputs

Finally in the WGAN outputs we can clearly see that CapsWGAN produces clear and better outputs, hence has learned more quickly than the WGAN.

8.3 Demonstration

Initially for demonstration of semantic in-painting we used the MNIST model of CapsDCGAN that we used comparison in the previous sections. We had promising results as seen in the figure 8.10.

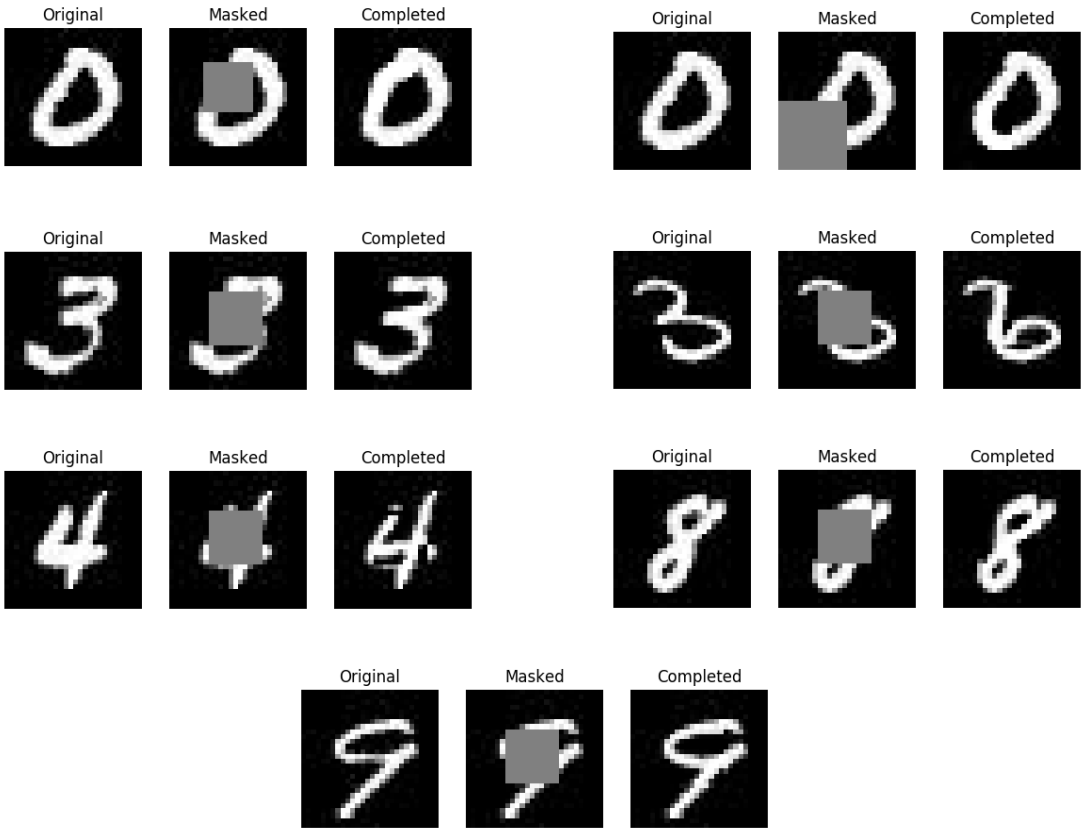


Figure 8.10: MNIST examples

For the completion of faces however the process was more complex. We initially trained CapsDCGAN on LFW dataset [16] for 50 epochs. Since LFW contains only 13,233 images, the model suffered from low generalization. So we decided to use the large CelebA dataset [10] for training. CelebA consists of 202,599 images of 10,177 celebrities. We trained CapsDCGAN on CelebA for only 9 epochs and still obtained realistic images. The following are the completion outputs from CelebA model in figure 8.11.

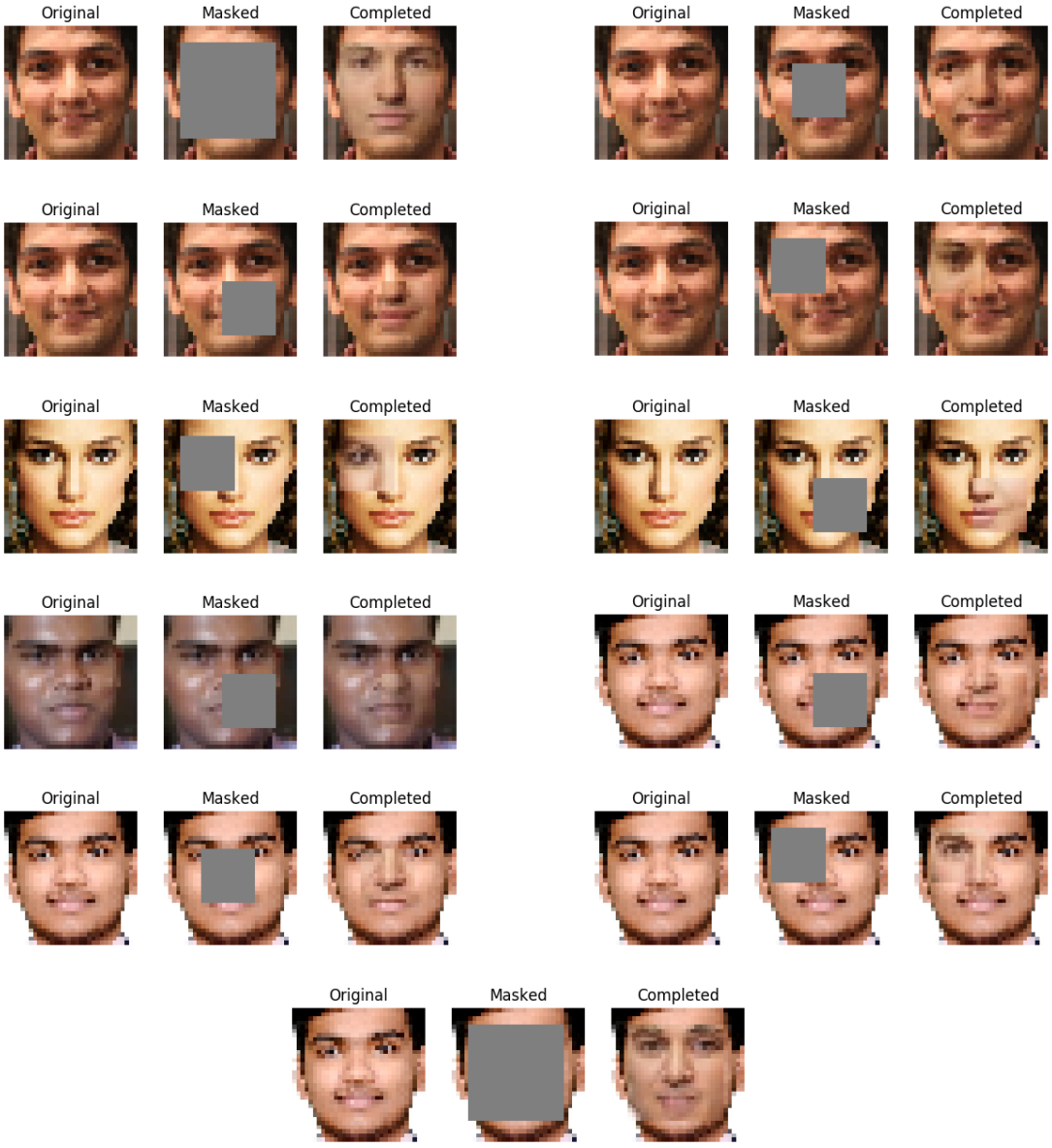


Figure 8.11: Face examples

Chapter 9
Conclusion

CHAPTER 9

CONCLUSION

"A year spent in artificial intelligence is enough to make one believe in God"

Alan Perlis,
First Turing award recipient

During the course of this project, we wished to replicate the results of the existing state-of-the-art in Generative Models. We implemented a few different versions of GANs with CapsNet. Our motivating assumption was that CapsNet would provide a performance improvement. We based this on the idea that it is more capable of understanding the variances in objects. This in turn should lead to lower data requirements during training of the model and consequently lower power consumption.

We provide a comparison between our novel CapsNet-based approach and other implementations of GAN for the same task. To observe this we augment the code of a few GANs, namely ACGAN, InfoGAN, DCGAN and WGAN, by implementing the discriminator with CapsNet. We decided to work with a few standard metrics such as Discriminator Loss, Generator Loss and Accuracy to measure its training performance. The data while training was captured and visualized in the form of graphs.

In conclusion, we can confidently state that augmenting the GANs with CapsNet was a fruitful endeavor. The CapsNet helped to reduce th training overhead considerably when compared to classical networks while providing remarkably similar results. Our research shows that embedding CapsNet into the GAN does not degrade its performance and, in certain cases, improves upon it.

REFERENCES

[1] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 214–223, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[2] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. Improved training of wasserstein gans. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5767–5777. Curran Associates, Inc., 2017.

[3] Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton. Dynamic routing between capsules. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 3856–3866. Curran Associates, Inc., 2017.

[4] Xi Chen, Xi Chen, Yan Duan, Rein Houthooft, John Schulman, Ilya Sutskever, and Pieter Abbeel. Infogan: Interpretable representation learning by information maximizing generative adversarial nets. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 2172–2180. Curran Associates, Inc., 2016.

[5] Ming-Yu Liu and Oncel Tuzel. Coupled generative adversarial networks. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 469–477. Curran Associates, Inc., 2016.

[6] Sebastian Nowozin, Botond Cseke, and Ryota Tomioka. f-gan: Training generative neural samplers using variational divergence minimization. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 271–279. Curran Associates, Inc., 2016.

[7] Augustus Odena, Christopher Olah, and Jonathon Shlens. Conditional image synthesis with auxiliary classifier gans. *arXiv preprint arXiv:1610.09585*, 2016.

[8] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, Xi Chen, and Xi Chen. Improved techniques for training gans. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 2234–2242. Curran Associates, Inc., 2016.

[9] Raymond A. Yeh, Chen Chen, Teck-Yian Lim, Mark Hasegawa-Johnson, and Minh N. Do. Semantic image inpainting with perceptual and contextual losses. *CoRR*, abs/1607.07539, 2016.

[10] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[12] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *CoRR*, abs/1511.06434, 2015.

[13] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc., 2014.

[14] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets. *CoRR*, abs/1411.1784, 2014.

[15] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[16] Gary B Huang, Manu Ramesh, Tamara Berg, and Erik Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical report, Technical Report 07-49, University of Massachusetts, Amherst, 2007.