TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING

Khwopa College of Engineering
Libali, Bhaktapur
Department of Computer Engineering

A REPORT ON

Nepali Handwritten Letter Generation Using Generative Adversarial Network

Submitted in partial fulfillment of the requirements for the degree

BACHELOR OF COMPUTER ENGINEERING

Submitted by

Aakash Raj Dhakal	KCE074BCT002
Asmin Karki	KCE074BCT012
Basant Babu Bhandari	KCE074BCT013
Laxman Maharjan	KCE074BCT020

Under Supervision of
Er.Shiva Kumar Shrestha
Department of Computer Engineering

Khwopa College of Engineering
Libali, Bhaktapur
10 Feb 2021

Certificate of Approval

The undersigned certify that the final year project entitled **"Nepali Handwritten Letter Generation using Generative Adversarial Network "** submitted by Aakash Raj Dhakal, Asmin Karki, Basant Babu Bhandari and Laxman Maharjan to the Department of Computer Engineering in partial fulfillment of requirement for the degree of Bachelor of Engineering in Computer Engineering. The project was carried out under special supervision and within the time frame prescribed by the syllabus. We found the students to be hardworking, skilled, bona fide and ready to undertake any commercial and industrial work related to their field of study and hence we recommend the award of Bachelor of Computer Engineering degree.

.....
Er. Shiva Kumar Shrestha
(Project Supervisor)

.....
.....
(Project External)

.....
Er. Shiva Kumar Shrestha
Head of Department
Department of Computer Engineering, KhCE

Copyright

The author has agreed that the library, Khowpa College of Engineering and Management may make this report freely available for inspection. Moreover, the author has agreed that permission for the extensive copying of this project report for scholarly purpose may be granted by supervisor who supervised the project work recorded herein or, in absence the Head of The Department wherein the project report was done. It is understood that the recognition will be given to the author of the report and to Department of Computer Engineering, KhCE in any use of the material of this project report. Copying or publication or other use of this report for financial gain without approval of the department and author's written permission is prohibited. Request for the permission to copy or to make any other use of material in this report in whole or in part should be addressed to: Head of the Department of Computer Engineering

Khwopa College of Engineering(KhCE)
Liwali, Bhaktapur
Nepal

Acknowledgement

We take this opportunity to express our deepest and sincere gratitude to our supervisor Er. Shiva Kumar Shrestha, for his insightful advice, motivating suggestions, invaluable guidance, help and support in successful completion of this project and also for his constant encouragement and advice throughout our Bachelor's program.We are deeply indebted to our teacher Er. Bindu Bhandari for boosting our efforts and morale by their valuable advices and suggestion regarding the project and supporting us in tackling various difficulties.In Addition, we also want to express our gratitude towards Er. Dinesh Gothe for providing the most important advice and giving realization of the practical scenario of the project.

Aakash Raj Dhakal	KCE074BCT002
Asmin Karki	KCE074BCT012
Basant Babu Bhandari	KCE074BCT013
Laxman Maharjan	KCE074BCT020

Abstract

The Generative modeling became recently one of the most important field in deep learning, and the Generative Adversarial Networks is the new research line in this field, the GANs have proven their ability to generate a high resolution of images and achieved remarkable success for computer vision in general, by the same way, we assume that this tool will be able to generate accurate Nepali letters. Our model uses Deep Convolutional Generative Adversarial Network architecture to generate the Nepali Handwritten Images after training on real-life datasets. It uses padding and strides to get better and good result than normal Generative network. Real-life datasets contains 36 different classes of Handwritten Images of size 64x64 which are feed to our model. We defined the Convolutional Neural Network Architecture for Generator and Discriminator with neccessary hyperparameters and then trained the model to generate output. The expected benefit of this work explores the potential to serve the Nepali calligraphy. As an impact for further studies in the future, this work may provide an insight into the possibility of generating new kinds of Nepali calligraphy.

Keywords: *Nepali Handwritten letter Generation, Nepali Language, Neural Network, Machine Learning, General Adversarial Network*

Contents

Certificate of Approval i
Copyright i
Acknowledgement ii
Abstract iii
List of Figures vi
List of Symbols and Abbreviation viii

1 INTRODUCTION **1**
1.1 Background 1
1.2 Problem Statement 2
1.3 Objective 2

2 LITERATURE REVIEW **3**
2.1 Handwritten Digits Generation 3
2.2 Bangla Handwritten Digit Generation 3
2.3 Generating Arabic Letters using GANs 3

3 REQUIREMENT ANALYSIS **4**
3.1 SOFTWARE REQUIREMENT 4
3.1.1 Python 4
3.1.2 Torch 4
3.1.3 Matplotlib 4
3.2 FUNCTIONAL REQUIREMENT 4
3.2.1 Generation of Devnagari Alphabets 4
3.2.2 Build Identifiable Characters 4
3.2.3 Precise and Understandable 4
3.3 NON-FUNCTIONAL REQUIREMENT 5
3.3.1 Accuracy 5
3.3.2 Easily Modifiable 5
3.3.3 Speed 5
3.3.4 Reliable 5
3.4 FEASIBILITY STUDY 6
3.4.1 Economic Feasibility 6
3.4.2 Technical Feasibility 6
3.4.3 Operational Feasibility 6

4 SYSTEM DESIGN AND ARCHITECTURE **7**
4.1 System Block Diagram 7
4.2 Usecase Diagram 8
4.3 Sequence Diagram 9

5 METHODOLOGY **10**
5.1 SOFTWARE DEVELOPMENT APPROACH 10
5.1.1 Rapid Prototyping 10
5.1.2 Rapid Prototyping Process 10
5.2 DATA COLLECTION 11
5.3 DATA PREPARATION 12
5.3.1 Transform 13
5.3.2 Normalize 13
5.3.3 Denormalize 13
5.4 Model Development 13
5.4.0.1 Our Model Creation 13
5.4.0.2 Our Model Architecture 14
5.4.0.3 Generator Model 14
5.4.0.4 Discriminator Model 15
5.5 TRAINING 15
5.5.1 Part 1 - Train the Discriminator 16
5.5.2 Part 2 - Train the Generator 16

6 RESULT AND DISCUSSION **17**
6.1 Result 17
6.1.1 Result of Training for different Hyperparameters 18
6.1.1.1 Hyperparameters 18
6.1.2 Graph Analysis 19

7 CONCLUSION AND RECOMMENDATION **21**

BIBLIOGRAPHY **23**

Appendix **24**
A Screen shots of code 24
B Outcomes 31

List of Figures

1.1 Basic GAN Architecture 1

4.1 Standard GAN Model Block Diagram 7
4.2 Usecase Diagram of Handwritten Text Generation GAN Model 8
4.3 Sequence Diagram of Handwritten Text Generation GAN model 9

5.1 Rapid Prototyping Model for Software Development 11
5.2 Activity Diagram for Data Preparation of Handwritten Text Generation 12
5.3 Mathematical representation of Normalization 13
5.4 Our Model Architecture 14
5.5 Handwritten Letter Generative CNN 15

6.1 Synthesized Intermediate Images of Text Generation 17
6.2 Synthesized Image For above Hyperparameters 18
6.3 Generator and Discriminator Loss For Hyperparameters 19
6.4 Final Output Obtained From the Our Model 20

List of Abbreviation

CNN	Convolutional Neural Network
CTC	Connectionist Temporal Classification
GAN	Generative Adversarial Network
GPU	Graphical Processing Unit
DCGAN	Deep Convolutional Generative Adversarial Network

Chapter 1

INTRODUCTION

1.1 Background

A generative adversarial network (GAN) is a class of machine learning frameworks designed by Ian Goodfellow and his colleagues in 2014. Two neural networks contest with each other in a game. Given a training set, this technique learns to generate new data with the same statistics as the training set. For example, a GAN trained on photographs can generate new photographs that look at least superficially authentic to human observers, having many realistic characteristics. Though originally proposed as a form of generative model for unsupervised learning, GANs have also proven useful for semi-supervised learning, fully supervised learning,and reinforcement learning. We use supervised learning, one trains the machine with well-labeled data. This allows for producing output based on previous experience. This implementation maps the input variables to an output variable and uses an algorithm to learn the relationship between them. This involves learning to predict a label associated with the data. In this implementation, Devanagari script(rounded shapes within squared outlines and a horizontal line that runs along the top of all characters) characters are fed into the GAN as input. The network will then try to generate the characters as accurately as possible.

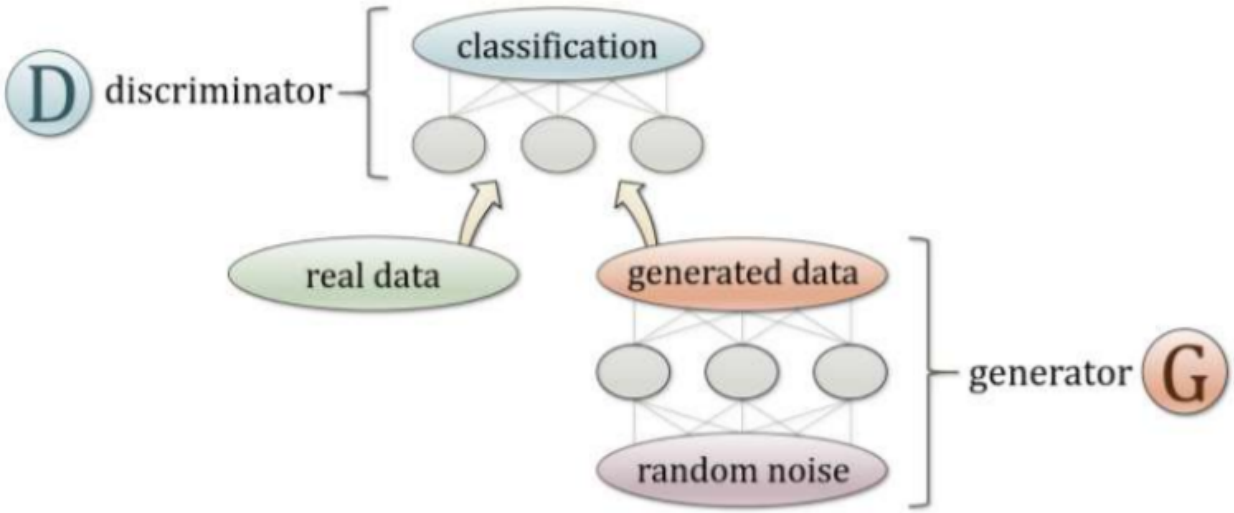


Figure 1.1: Basic GAN Architecture

1.2 Problem Statement

Having computers able to recognize text from images is an old problem that has many practical applications, such as automatic content search on scanned documents. Transcribing printed text is now a reliable technology. However, automatically recognizing handwritten text is still a hard and open problem. Today, in order to deal with such a complex problem, state-of-the-art solutions are all based on deep neural networks and the CTC loss. The supervised training of these neural networks requires large amounts of annotated data; in our case, images of handwritten text with corresponding transcripts. However, annotating images of text is a costly, time-consuming task. Therefore, we use available Devanagari character dataset to generate images of handwritten Devanagari characters. GAN offer powerful framework for generative modeling. This architecture enables the generation of diverge characters. The model discriminator is trained to classify real and fake image so that generator can mapped a accurate image of the character.

-

1.3 Objective

The main aim of this project is:

- To Generate Nepali Handwritten Letters using GAN

Chapter 2

LITERATURE REVIEW

Different researches have been performed on GAN. Researches that are related our project are as follows:

2.1 Handwritten Digits Generation

The First research paper for GANs was submitted by Ian Goodfellow in 2014 [4]. In this paper, he used MNIST dataset for testing his proposed framework and generated the handwritten digits. In his result he made no claim that the result samples are better than samples generated by existing methods, but believed that these samples are at least competitive with the better generative models.Using GAN network there are audio generation[[13]][[14]][[9]][[14]], video generation network and many more[[7]][[8]][[10]][[6]][[3]][[1]].

2.2 Bangla Handwritten Digit Generation

This paper is about generating the Bangla Handwritten Digit using the DCGAN architecture [5]. In this paper, to achieve the goal by they used the three most popular Bangla handwritten datasets CMATERdb, BanglaLekha-Isolated, ISI and their own dataset Ekush. The proposed DCGAN successfully generate Bangla digits which makes it a robust model to generate Bangla handwritten digits from random noise. The losses in each dataset is shown below along the output.

2.3 Generating Arabic Letters using GANs

This is the master Thesis Paper is about generating the Arabic Letters using DCGAN architecture [2]. The dataset is composed of pictures for Arabic alphabet (from 'alef' to 'yeh' 480 images per character) handwritten with images size 32x32 (for dcgan the standard image size is supposed to be 64x64), because it's the only dataset available for handwritten Arabic letters some changes of standard architecture of generator and discriminator of the DCGAN to adapt it for size 32x32. The losses in each dataset is shown below along the output.

Chapter 3

REQUIREMENT ANALYSIS

3.1 SOFTWARE REQUIREMENT

Our text generation system requires Python,PyTorch which are described below:

3.1.1 Python

Python is used as programming language because it has large comprehensive standard library and framework for Machine learning.

3.1.2 Torch

Torch is used for neural network architecture, normalization of data and calculate the loss function in this project.

3.1.3 Matplotlib

Matplotlib is python library used for visualizing the images in this project. .

3.2 FUNCTIONAL REQUIREMENT

3.2.1 Generation of Devnagari Alphabets

The basic function of our project is to generate the devnagari alphabets using the Generative adversarial networks.

3.2.2 Build Identifiable Characters

The letter that is to be displayed should be identifiable to the human readable form. And it should to be correctly oriented and displayed.

3.2.3 Precise and Understandable

The characters that are displayed should be precise and have accuracy to the limit that any people understanding Devnagari could understand it.

3.3 NON-FUNCTIONAL REQUIREMENT

These requirements are not needed by the system but are essential for the better performance of model. The points below focus on the non-functional requirement of the system.

3.3.1 Accuracy

The system is to be maximum accuracy as possible. It should have maximum accuracy as possible.

3.3.2 Easily Modifiable

The system is to be easily modifiable by the network. The Devnagari characters are not only something that we can train for the network.

3.3.3 Speed

The system is supposed to have faster speed and better performance.

3.3.4 Reliable

The GAN model is to be reliable to maximum limit and thus be able to generate the digit better and reliably.

3.4 FEASIBILITY STUDY

The following points describes the feasibility of the project.

3.4.1 Economic Feasibility

Simply, economic feasibility is about the monetary value of project.The overall economic expenditure is on the device and the effort which is more precious than the money. The network are build using the proper devices and the device is the major expenditure to our project. Computational power is something that is required to more for the system.

3.4.2 Technical Feasibility

It is one of the major concern about the techniques and tools used for the project. The project is to be trained with the lots of data across the network. Our Generative adversarial networks model is the main model that is to be trained for the technical purpose. The complexity lies in the training the network through the system. A proper network is to be needed for the system. And the processor and the system is to be of required capability. And of course the devices and the system belongs to the technical requirements of the system that is must for the project.

3.4.3 Operational Feasibility

The project operational feasibility is about the project operation and its obstacles to encounter with it. The project can be operational just after training from labeled data using generative adversarial networks. The operation of the project is the basic operation and its operation need to be feasible to the system. And we can create feasible system using the network.

Chapter 4

SYSTEM DESIGN AND ARCHITECTURE

We developed a Nepali Handwritten letter Generator by building the GAN model and feeding it with lots of training data-set and finally it is able to output the realistic Nepali Handwritten letter.

4.1 System Block Diagram

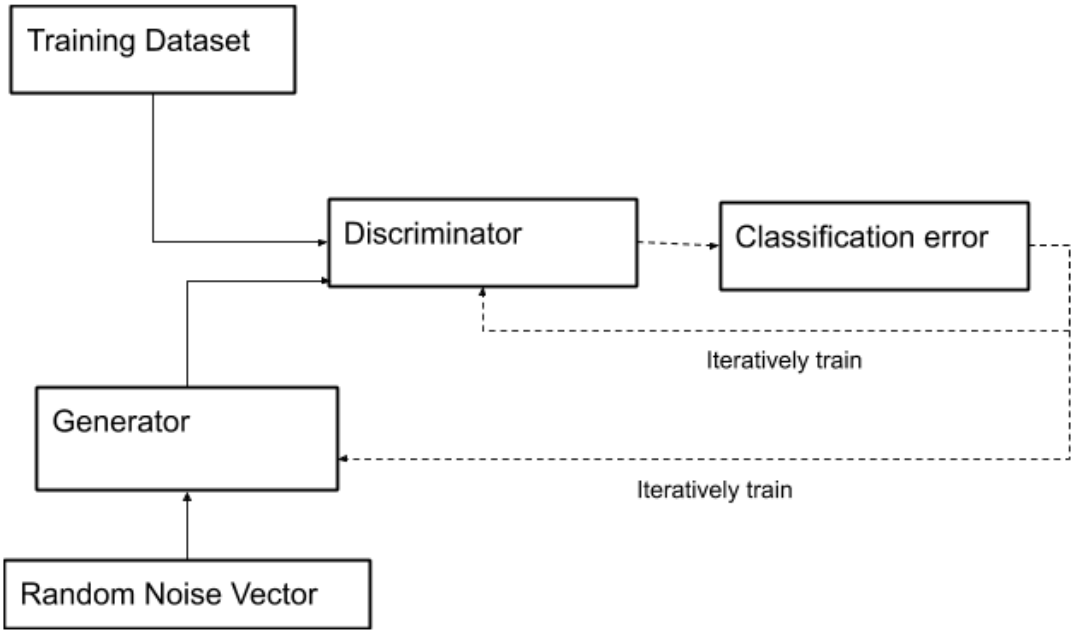


Figure 4.1: Standard GAN Model Block Diagram

4.2 Usecase Diagram

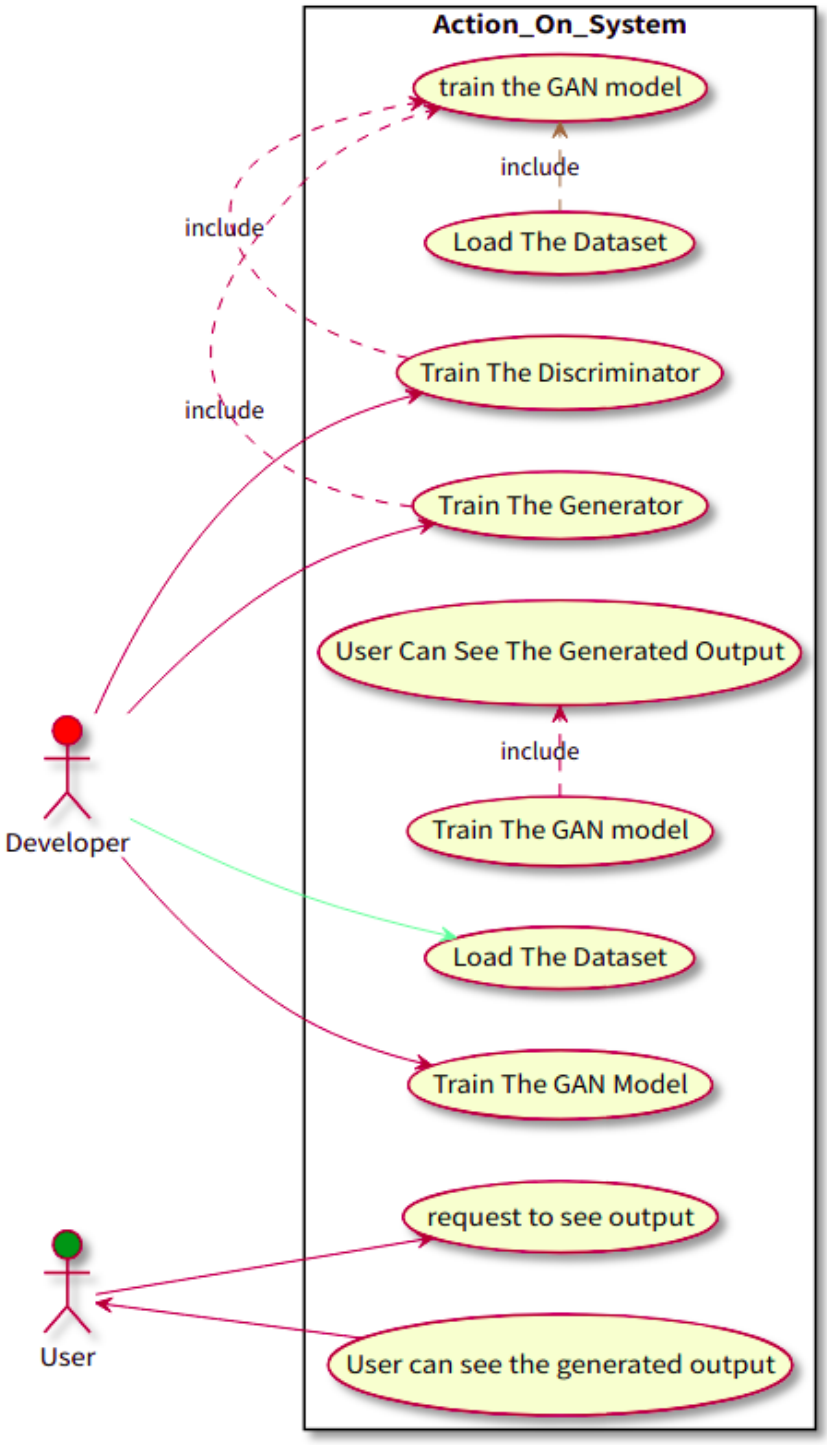


Figure 4.2: Usecase Diagram of Handwritten Text Generation GAN Model

4.3 Sequence Diagram

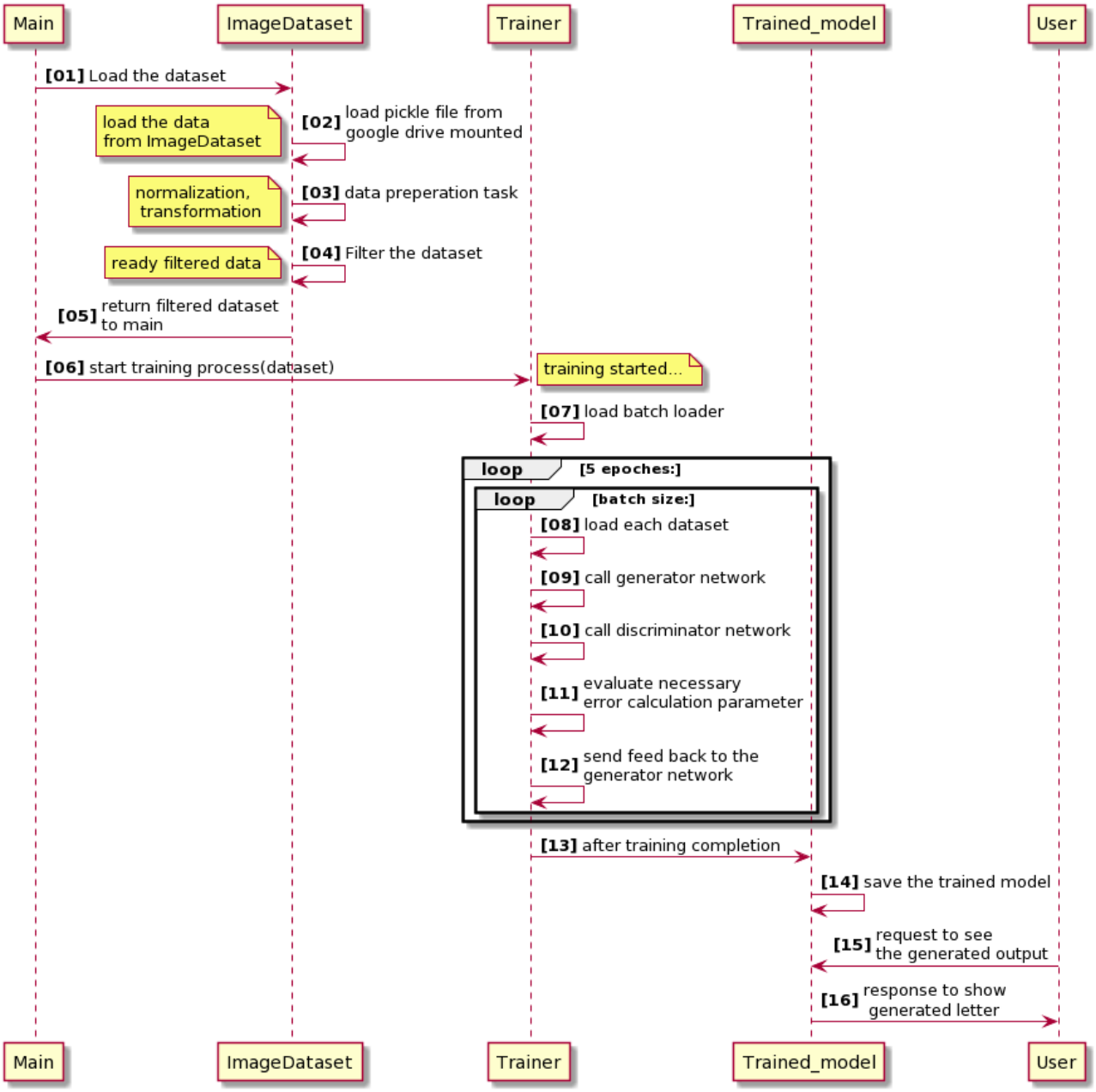


Figure 4.3: Sequence Diagram of Handwritten Text Generation GAN model

Chapter 5

METHODOLOGY

5.1 SOFTWARE DEVELOPMENT APPROACH

5.1.1 Rapid Prototyping

Rapid Prototyping applies an iterative approach to the design stage of an app or website. The objective is to quickly improve the design using regularly updated prototypes and multiple short cycles. This saves time and money by solving common design issues before development begins, helps businesses to reach market quicker, and puts the focus of development on the needs of the end-user.

5.1.2 Rapid Prototyping Process

The rapid prototyping process involves three simple steps:

- **Prototype** – The team creates an initial prototype. This is a visual representation of the design specifications as set out in the requirements document.
- **Review** – The creators share the prototype with other team members, stakeholders, and focus groups made up the intended end-users. Everyone evaluates both the design and usability before submitting feedback.
- **Refine and Iterate** – The feedback is used to create a new iteration of the prototype. The process then cycles round to Step 2 for further feedback. This continues until there are no more changes or a specified cut-off is reached.

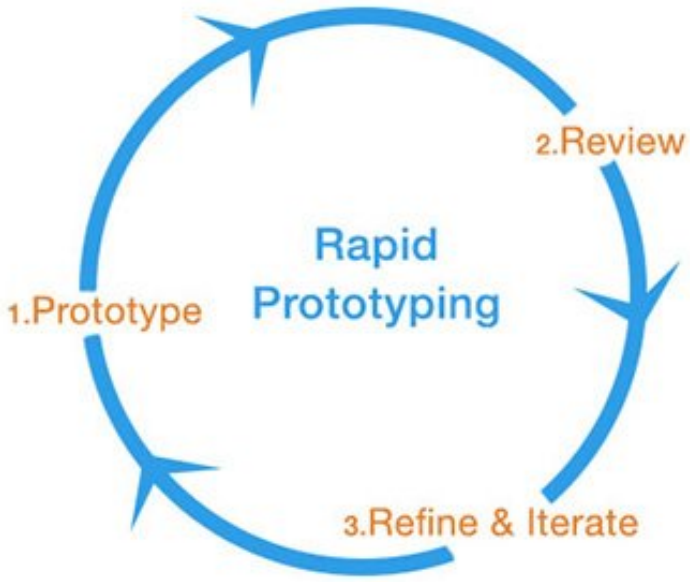


Figure 5.1: Rapid Prototyping Model for Software Development

5.2 DATA COLLECTION

Initially, we found the dataset as:

- **Numerals** (288 samples per class, 10 classes)
- **Vowels** (221 samples per class, 12 classes)
- **Consonants** (205 samples per class, 36 classes)

which is collected by Ashok Kumar Pant [11] who profiles you can check at `https://www.kaggle.com/ashokpant/devanagari-character-dataset` and we use paper of 2014 origin gan research paper.We made a generative adversarial network by which is densely connected node and we try a lot to get the desired output but this network does not give satisfactory results for 36 class of dataset using old dataset. Then we decided to go with a strong and more advanced algorithm called deep convolution GAN.

This is an image database of Handwritten Devanagari characters. There are 46 classes of characters with 2000 examples each. This is the link to the data set repository. `http://archive.ics.uci.edu/ml/datasets/Devanagari+Handwritten+Character+Dataset` Now, using the augmentation technique, we increase the size of the dataset by 5 times and we make it 10000 samples per class.

According to the 2016 research paper, we construct a deep convolution network and we feed an augmented dataset of 10000 samples per class which give more satisfactory results than the previous one and by tuning the hyperparameter we are able to produce this output as you can see in output section which is a computer-generated letter from generator saved model.

5.3 DATA PREPARATION

After the data has been collected, it was processed to make into correct format so that the neural networks can understand both the inputs and outputs. The block diagram (figure 6.2) shows the data preparation steps of the project.

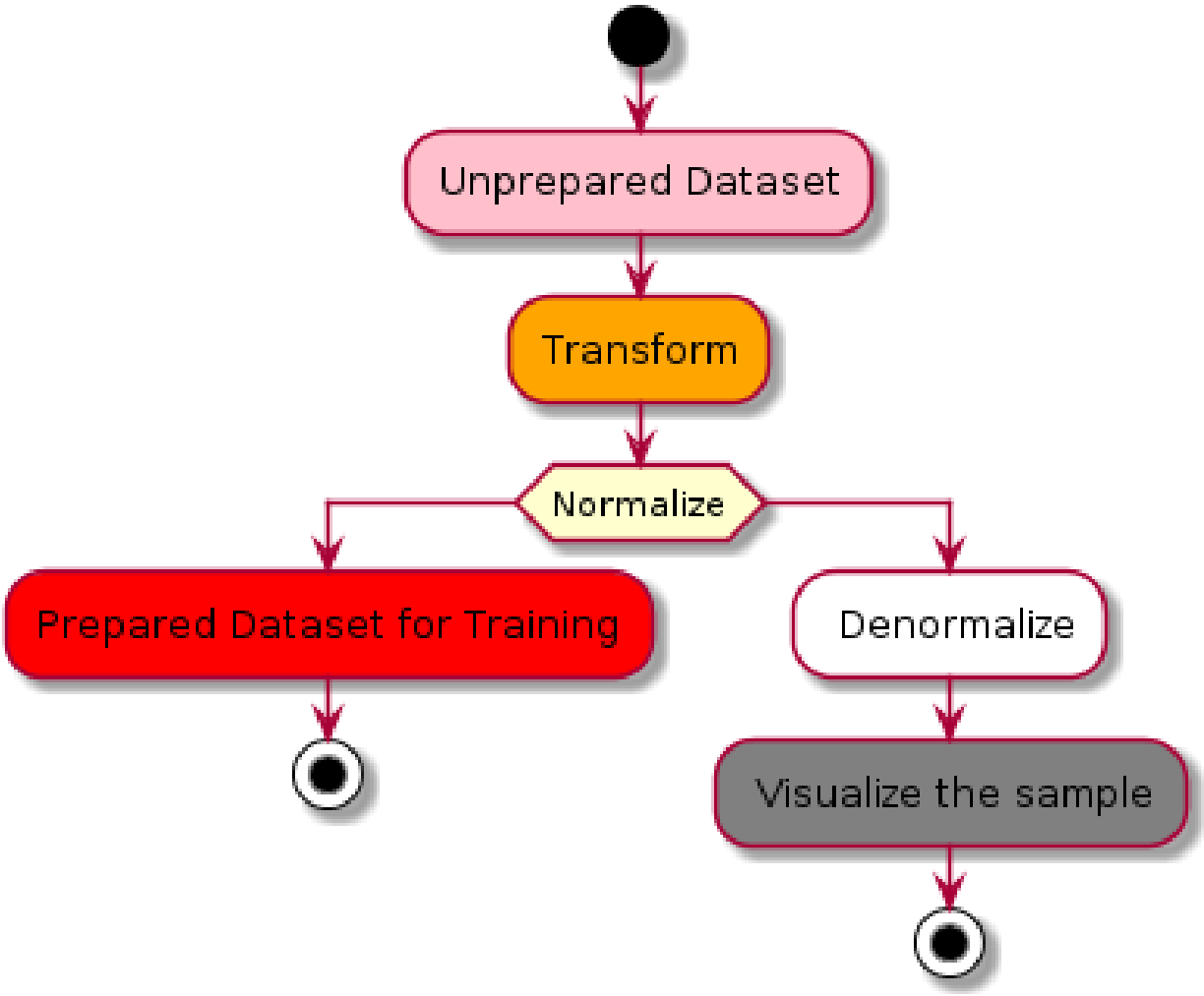


Figure 5.2: Activity Diagram for Data Preparation of Handwritten Text Generation

5.3.1 Transform

Data transformation is the process in which you take data from its raw, siloed and normalized source state and transform it into data that's joined together, dimensionally modeled, denormalized, and ready for analysis.

5.3.2 Normalize

Normalization is the process of changing the range from one to another. In our project we use normalization to map the image pixel value between -1 to 1.

Min-Max Normalization:

$$\hat{X}[:,i] = \frac{X[:,i] - \min(X[:,i])}{\max(X[:,i]) - \min(X[:,i])}$$

Figure 5.3: Mathematical representation of Normalization

5.3.3 Denormalize

The reverse process of normalization is denormalization. Denormalization is done to recover the original data which is normalized earlier.

5.4 Model Development

As above mentioned, various algorithms and models were developed for building the Nepali letter Generation using Generative adversarial network [12]. Following topics describes all the algorithms and models used in the project in detail;

5.4.0.1 Our Model Creation

Firstly, we attempt to make the model through the GAN model which was based on the Ian Godfellow model, but due to the failure in expectation of desired output through the model, we choosed DCGAN that used convolutional neural network rather than than the artificial neural network.

Torch library is used in the process. Pickle is used for data storage and retrieval. The generator and discriminator is defined to train the model. Firstly the generator is defined with four layers. Along all the layers, they have kernel size of four,stride two and padding of one for all.Leaky-ReLU is used as activation function at first three layers and sigmoid is used for last layer. The layer is obtained as (Nx1x1x1). Secondly the discriminator is defined with four layers.Along all the layers, they have kernel size of four,stride two and padding of one for all except the first. ReLU is used as activation function at first three layers and tanh is used for last layer. We obtain the output in the form of (Nxchannels_imgx64x64).Then the hyper parameters are defined. The hyperparameters are learning rate, batch_size, image_size, channel_image, channel_noise, number_of_epochs and number_of_pixels. The channels for discriminator and generator is defined then.

Afterwards,datasets are loaded. And the numpy array is converted to tensor form.The datasets is filtered. And feature scaling is done to get the required dimension of the image. And the

optimizer is setup as Adam optimimzer and training is done to get the model perform desired work. Then the loaded model is saved to get the desired output.

5.4.0.2 Our Model Architecture

For model development of DCGAN we need to development two CNN network called discriminator network and generator network. In discriminator network five convolution layer, leakyrely as activation function except at the final layer to avoid vanishing gradient problem.For the output layer of discriminator network we use sigmoid as activation function and that for generator network we use tanh as activation function.The generator network consist of file convolution layer. The summary of our model we used is as below:

```
Discriminator(
  (net): Sequential(
    (0): Conv2d(1, 16, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (1): LeakyReLU(negative_slope=0.2)
    (2): Conv2d(16, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (3): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2)
    (5): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (6): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2)
    (8): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (9): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2)
    (11): Conv2d(128, 1, kernel_size=(4, 4), stride=(2, 2))
    (12): Sigmoid()
  )
)
Generator(
  (net): Sequential(
    (0): ConvTranspose2d(256, 256, kernel_size=(4, 4), stride=(1, 1))
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU()
    (6): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (7): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU()
    (9): ConvTranspose2d(64, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (10): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU()
    (12): ConvTranspose2d(32, 1, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (13): Tanh()
  )
)
```

Figure 5.4: Our Model Architecture

5.4.0.3 Generator Model

The input to the generator is typically a vector or a matrix which is used as a seed for generating an image. Once again, to keep things simple, we'll use a feed forward neural network with 3 layers, and the output will be a vector of size 4096, which can be transformed to a 64×64 pixels image. This is the random input to the Generator model This is a Generator Network Model. We use the tanh( ) activation function for the output layer of the generator. Note that since the outputs of the tanh( ) activation lie in the range [-1,1], we have applied the same transformation to the images in the training dataset. Let's generate an output vector using the generator and view it as an image by transforming and de-normalizing the output.

The ReLU activation (Nair Hinton, 2010) is used in the generator except for the output

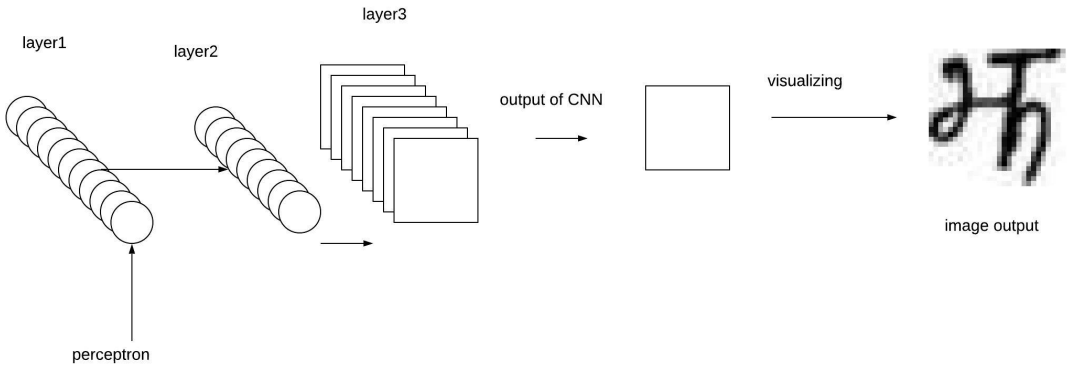


Figure 5.5: Handwritten Letter Generative CNN

layer which uses the Tanh function. We observed that using a bounded activation allowed the model to learn more quickly to saturate and cover the colour space of the training distribution. Within the discriminator we found the leaky rectified activation (Maas et al., 2013) (Xu et al., 2015) to work well, especially for higher resolution modelling.

5.4.0.4 Discriminator Model

The discriminator takes an image as input, and tries to classify it as "real" or "generated". In this sense, it's like any other neural network. While we can use a CNN for the discriminator, we'll use a simple feed forward network with 2 linear layers to keep things since. We'll treat each 64×64 image as a vector of size 4096. Input for the discriminator model will be 4096. This is a Discriminator Network Model. We use the Leaky ReLU activation for the discriminator. The output of the discriminator is a single number between 0 and 1, which can be interpreted as the probability of the input image being fake i.e. generated.

Different from the regular ReLU function, Leaky ReLU allows the pass of a small gradient signal for negative values. As a result, it makes the gradients from the discriminator flows stronger into the generator. Instead of passing a gradient (slope) of 0 in the back-prop pass, it passes a small negative gradient.

5.5 TRAINING

Finally, now that we have all of the parts of the GAN framework defined, we can train it. Be mindful that training GANs is somewhat of an art form, as incorrect hyperparameter settings lead to mode collapse with little explanation of what went wrong. Here, we will closely follow Algorithm 1 from Goodfellow's paper, while abiding by some of the best practices shown in ganhacks. Namely, we will "construct different mini-batches for real and fake" images, and also adjust G's objective function to maximize logD(G(z)). Training is split up into two main parts. Part 1 updates the Discriminator and Part 2 updates the Generator.

5.5.1 Part 1 - Train the Discriminator

Recall, the goal of training the discriminator is to maximize the probability of correctly classifying a given input as real or fake. In terms of Goodfellow, we wish to "update the discriminator by ascending its stochastic gradient". Practically, we want to maximize log(D(x))+log(1D(G(z))). Due to the separate mini-batch suggestion from ganhacks, we will calculate this in two steps. First, we will construct a batch of real samples from the training set, forward pass through D, calculate the loss (log(D(x))), then calculate the gradients in a backward pass. Secondly, we will construct a batch of fake samples with the current generator, forward pass this batch through D, calculate the loss (log(1D(G(z)))), and accumulate the gradients with a backward pass. Now, with the gradients accumulated from both the all-real and all-fake batches, we call a step of the Discriminator's optimizer.

5.5.2 Part 2 - Train the Generator

As stated in the original paper, we want to train the Generator by minimizing log(1D(G(z))) in an effort to generate better fakes. As mentioned, this was shown by Goodfellow to not provide sufficient gradients, especially early in the learning process. As a fix, we instead wish to maximize log(D(G(z))). In the code we accomplish this by: classifying the Generator output from Part 1 with the Discriminator, computing G's loss using real labels as GT, computing G's gradients in a backward pass, and finally updating G's parameters with an optimizer step. It may seem counter-intuitive to use the real labels as GT labels for the loss function, but this allows us to use the log(x) part of the BCELoss (rather than the log(1x) part) which is exactly what we want.

Chapter 6

RESULT AND DISCUSSION

6.1 Result

We obtained the exact output of the Nepali written text generation with the output given in the figure below. The output of the GAN generation along with the fake image is given in the figure below.

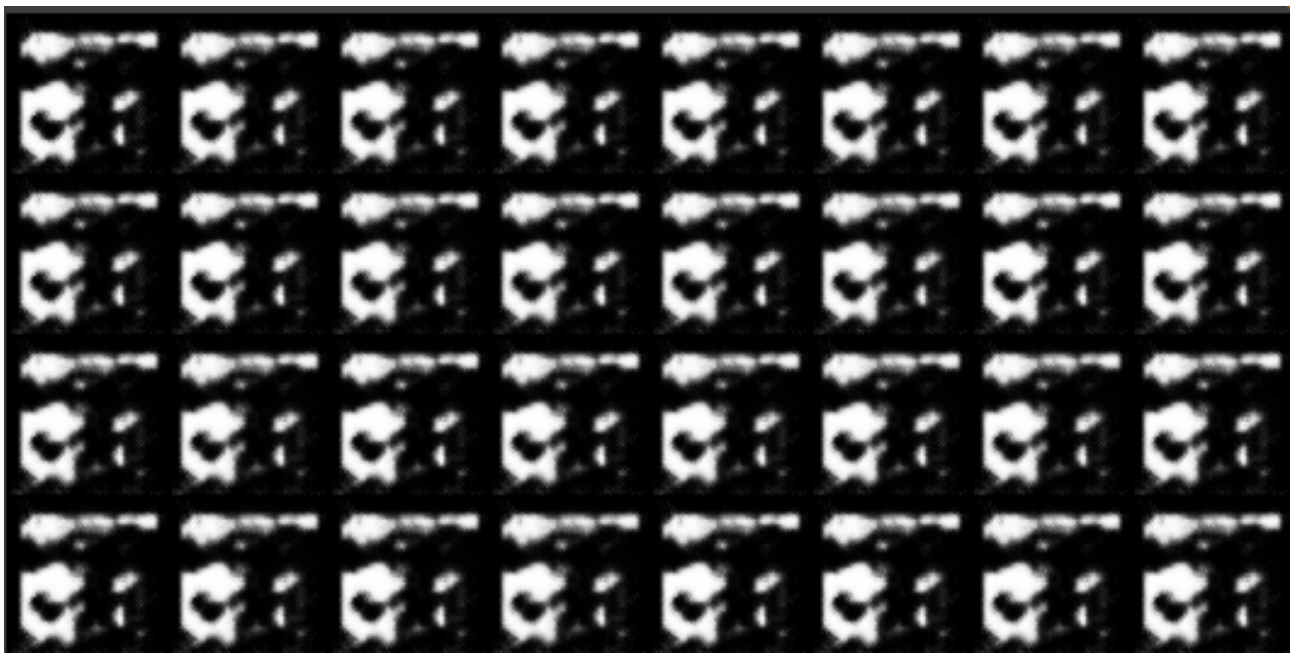


Figure 6.1: Synthesized Intermediate Images of Text Generation

this above result say the intermediate batch of image generated by partially trained generator while training in the mid way.

6.1.1 Result of Training for different Hyperparameters

These are the some best output determined by human eye inspection, generated with the help of trained generator for two different set of hyperparameters is shown in the figure below:

6.1.1.1 Hyperparameters

- Learning rate = 0.0005
- Batch size = 64
- Image size = 64
- Channels image = 1
- Channels noise = 512
- Number of epoch = 1
- Number of pixel = 64
- Features of Discriminator = 16
- Features of Generator = 16

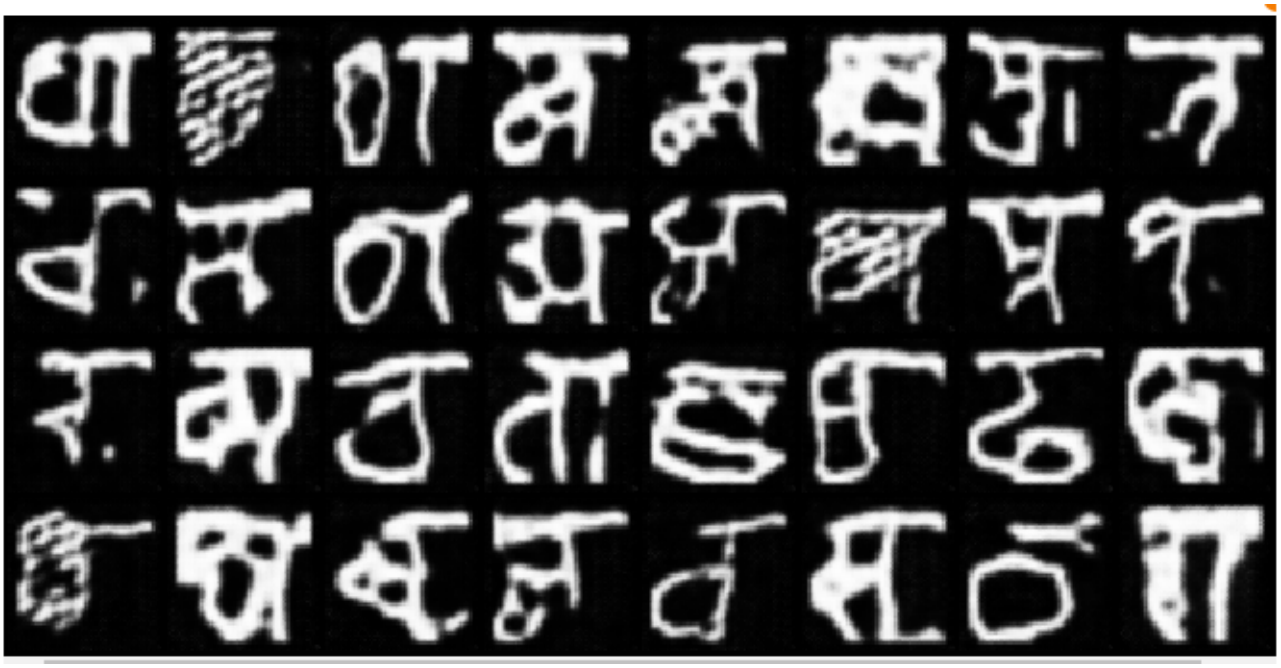


Figure 6.2: Synthesized Image For above Hyperparameters

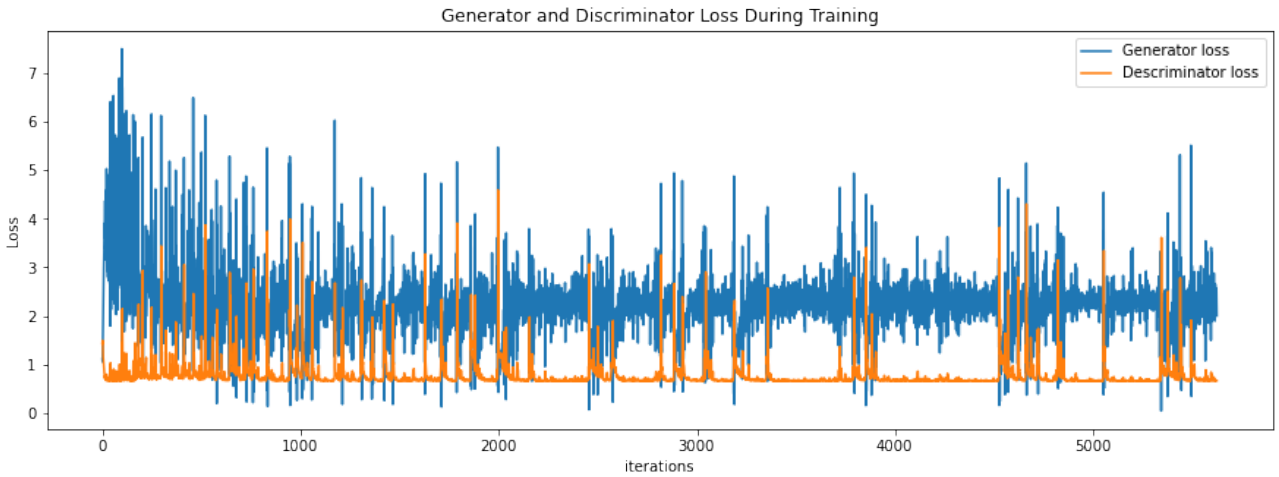


Figure 6.3: Generator and Discriminator Loss For Hyperparameters

A GAN can have two loss functions: one for generator training and one for discriminator training.In the loss schemes we'll look at here, the generator and discriminator losses derive from a single measure of distance between probability distributions. In both of these schemes, however, the generator can only affect one term in the distance measure: the term that reflects the distribution of the fake data. So during generator training we drop the other term, which reflects the distribution of the real data.

6.1.2 Graph Analysis

The most popular graph in GAN based network is loss graph, where we plot descriminator and generator loss while training the network. Here we use Binary Cross Entropy Loss function. We can clearly observe from both graph, initially for untrained network of generator has highest loss cause it is not trained yet. But as iteteration goes on increasing the generator network learns how to generate images similer to the real dataset, the loss of generator gradually decrease. In case of discriminator the loss lise between o to 1. For this type of trend of generator network, a single image generated by trained generator is shown below:

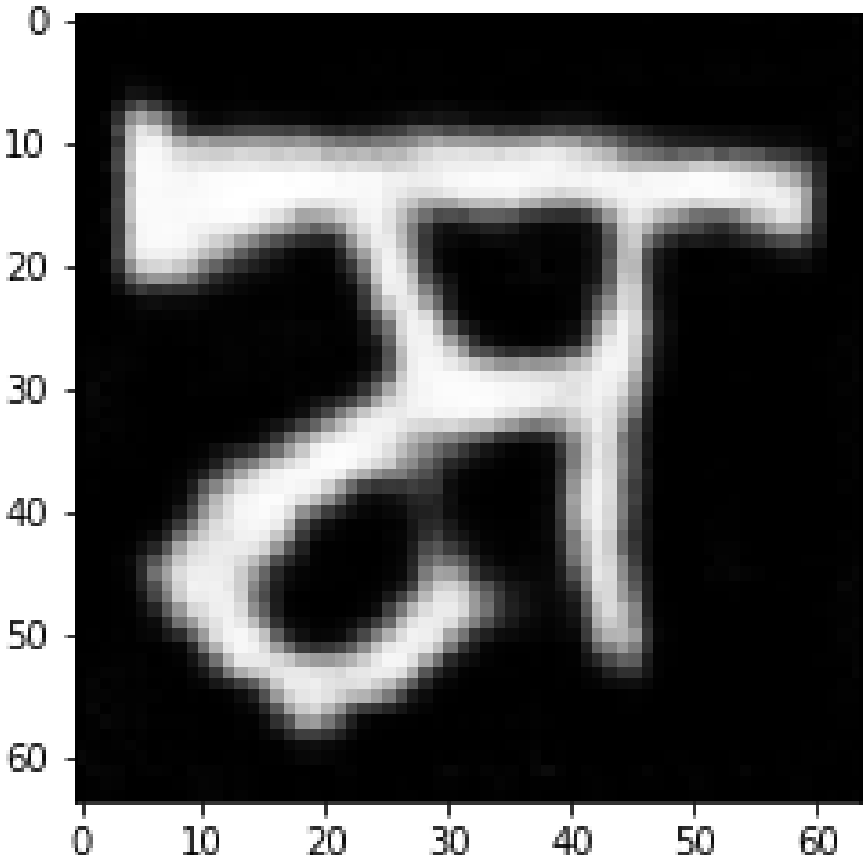


Figure 6.4: Final Output Obtained From the Our Model

Chapter 7

CONCLUSION AND RECOMMENDATION

We demonstrated that generative adversarial networks, and in particular their deep convolutional variants, function exceptionally well as generative models. We described our DCGAN model implementations and showed the realistic Nepali handwritten text generation.
This paper has successfully demonstrated the implementation of our DCGAN model on real-life datasets. Even though the generated characters were noisy, further fine tuning can be done to give significantly better result. Furthermore, our model architecture can be modified and trained to generate a new kind of Nepali calligraphy.
DCGAN model was trained for 5625 iterations. At each iterations, the output of the model was stored. Despite noise the generated characters was readable to human eye.
Therefore, It is clear that generative models offer more representational power than their discriminative counterparts. We foresee great future success with GANs, DCGANs, and generative models in general. Our recommendation after doing hand written letter generation using DCGAN project are:

1. Generation of large hand written generation.
2. Font generation using generated letter from generator.
3. Recognition of generated letter by small child while learning.
4. Development of other artistic DCGAN model by replacing the dataset.
5. DCGAN based Data Augmentation.

Bibliography

[1] Bojanowski, P., Joulin, A., Lopez-Paz, D., and Szlam, A. (2019). Optimizing the latent space of generative networks.

[2] CHEBOUAT, A. (2018). Generating arabic letters using generative adversarial networks. In *International Conference on Recent Trends in Image Processing and Pattern Recognition*. UNIVERSITY KASDI-MERBAH OUARGLA.

[3] Cherian, A. and Sullivan, A. (2019). Sem-gan: Semantically-consistent image-to-image translation. In *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 1797–1806.

[4] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680.

[5] Haque, S., Shahinoor, S. A., Rabby, A. S. A., Abujar, S., and Hossain, S. A. (2018). Onko-gan: Bangla handwritten digit generation with deep convolutional generative adversarial networks. In *International Conference on Recent Trends in Image Processing and Pattern Recognition*, pages 108–117. Springer.

[6] Karnewar, A., Wang, O., and Iyengar, R. S. (2019). MSG-GAN: multi-scale gradient GAN for stable image synthesis. *CoRR*, abs/1903.06048.

[7] Kumar, R., Kumar, K., Anand, V., Bengio, Y., and Courville, A. (2020). Nu-gan: High resolution neural upsampling with gan.

[8] Liu, J.-Y., Chen, Y.-H., Yeh, Y.-C., and Yang, Y.-H. (2020). Unconditional audio generation with generative adversarial networks and cycle regularization.

[9] Lu, S., Sirojan, T., Phung, B. T., Zhang, D., and Ambikairajah, E. (2019). Da-dcgan: An effective methodology for dc series arc fault diagnosis in photovoltaic systems. *IEEE Access*, 7:45831–45840.

[10] Marafioti, A., Perraudin, N., Holighaus, N., and Majdak, P. (2019). Adversarial generation of time-frequency features with application in audio synthesis. In Chaudhuri, K. and Salakhutdinov, R., editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 4352–4362. PMLR.

[11] Pant, A. K., Panday, S. P., and Joshi, S. R. (2012). Off-line nepali handwritten character recognition using multilayer perceptron and radial basis function neural networks. In *2012 Third Asian Himalayas International Conference on Internet*, pages 1–5. IEEE.

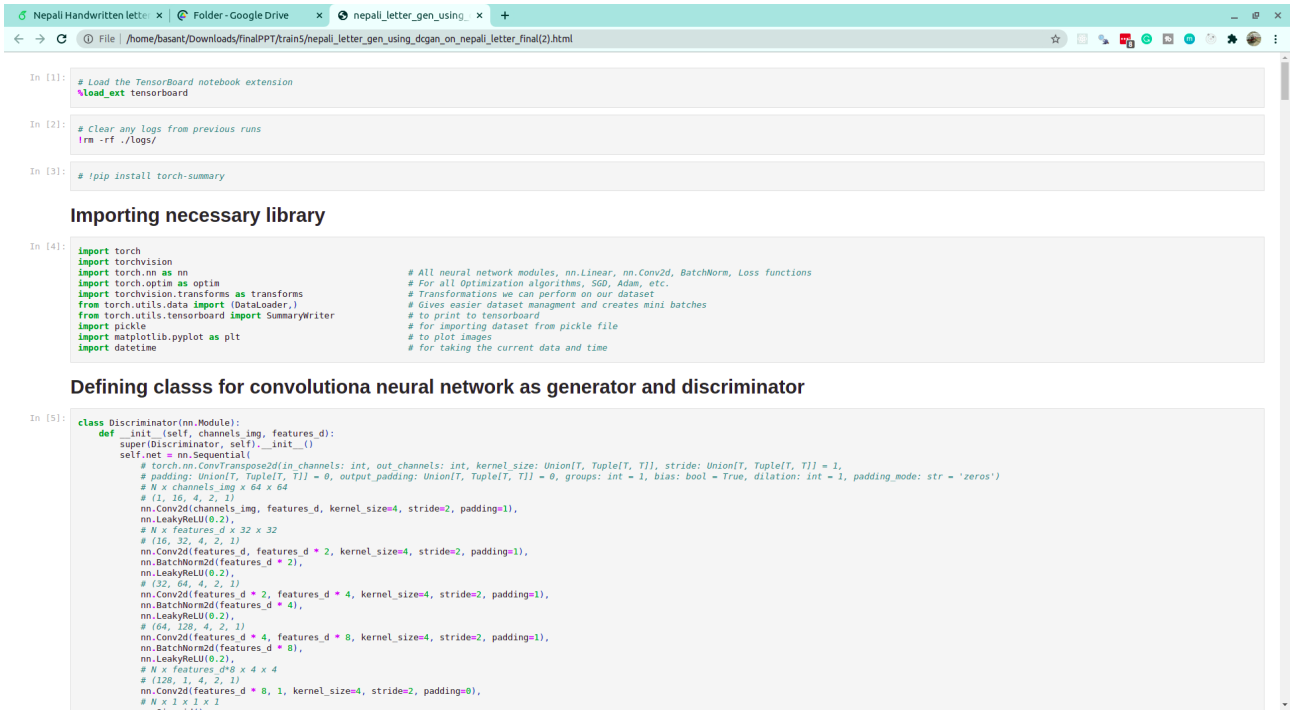
[12] Radford, A., Metz, L., and Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*.

[13] Shmelkov, K., Schmid, C., and Alahari, K. (2018). How good is my gan? In *Proceedings of the European Conference on Computer Vision (ECCV)*.

[14] Wu, Q., Chen, Y., and Meng, J. (2020). Dcgan-based data augmentation for tomato leaf disease identification. *IEEE Access*, 8:98716–98728.

Appendix

A Screen shots of code




```
# Load the TensorBoard notebook extension
%load_ext tensorboard
```

```
# Clear any logs from previous runs
!rm -rf ./logs/
```

```
# !pip install torch-summary
```

Importing necessary library

```
import torch
import torchvision
import torch.nn as nn                                   # All neural network modules, nn.Linear, nn.Conv2d, BatchNorm, Loss functions
import torch.optim as optim                             # For all Optimization algorithms, SGD, Adam, etc.
import torchvision.transforms as transforms             # Transformations we can perform on our dataset
from torch.utils.data import (DataLoader,)              # Gives easier dataset managment and creates mini batches
from torch.utils.tensorboard import SummaryWriter       # to print to tensorboard
import pickle                                           # for importing dataset from pickle file
import matplotlib.pyplot as plt                         # to plot images
import datetime                                         # for taking the current data and time
```

```python
class Discriminator(nn.Module):
    def __init__(self, channels_img, features_d):
        super(Discriminator, self).__init__()
        self.net = nn.Sequential(
            # torch.nn.ConvTranspose2d(in_channels: int, out_channels: int, kernel_size: Union[T, Tuple[T, T]], stride: Union[T, Tuple[T, T]] = 1,
            # padding: Union[T, Tuple[T, T]] = 0, output_padding: Union[T, Tuple[T, T]] = 0, groups: int = 1, bias: bool = True, dilation: int = 1, padding_mode: str = 'zeros')
            # N x channels_img x 64 x 64
            # (1, 16, 4, 2, 1)
            nn.Conv2d(channels_img, features_d, kernel_size=4, stride=2, padding=1),
            nn.LeakyReLU(0.2),
            # N x features_d x 32 x 32
            # (16, 32, 4, 2, 1)
            nn.Conv2d(features_d, features_d * 2, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(features_d * 2),
            nn.LeakyReLU(0.2),
            # (32, 64, 4, 2, 1)
            nn.Conv2d(features_d * 2, features_d * 4, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(features_d * 4),
            nn.LeakyReLU(0.2),
            # (64, 128, 4, 2, 1)
            nn.Conv2d(features_d * 4, features_d * 8, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(features_d * 8),
            nn.LeakyReLU(0.2),
            # N x features_d*8 x 4 x 4
            # (128, 1, 4, 2, 1)
            nn.Conv2d(features_d * 8, 1, kernel_size=4, stride=2, padding=0),
            # N x 1 x 1 x 1
            nn.Sigmoid(),
        )

    def forward(self, x):
        return self.net(x)
```

```python
class Generator(nn.Module):
    def __init__(self, channels_noise, channels_img, features_g):
        super(Generator, self).__init__()

        self.net = nn.Sequential(
            # torch.nn.ConvTranspose2d(in_channels: int, out_channels: int, kernel_size: Union[T, Tuple[T, T]], stride: Union[T, Tuple[T, T]] = 1,
            # padding: Union[T, Tuple[T, T]] = 0, output_padding: Union[T, Tuple[T, T]] = 0, groups: int = 1, bias: bool = True, dilation: int = 1, padding_mode: str = 'zeros')
            # N x channels_noise x 1 x 1
            # (256, 256, 4, 1, 0)
            nn.ConvTranspose2d(channels_noise, features_g * 16, kernel_size=4, stride=1, padding=0),
            nn.BatchNorm2d(features_g * 16),
            nn.ReLU(),
            # N x features_g*16 x 4 x 4
            # (256, 128, 4, 2, 1)
            nn.ConvTranspose2d(features_g * 16, features_g * 8, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(features_g * 8),
            nn.ReLU(),
            # (128, 64, 4, 2, 1)
            nn.ConvTranspose2d(features_g * 8, features_g * 4, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(features_g * 4),
            nn.ReLU(),
            # (64, 32, 4, 2, 1)
            nn.ConvTranspose2d(features_g * 4, features_g * 2, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(features_g * 2),
            nn.ReLU(),
            # (32, 1, 4, 2, 1)
            nn.ConvTranspose2d(features_g * 2, channels_img, kernel_size=4, stride=2, padding=1),
            # N x channels_img x 64 x 64
            nn.Tanh(),
        )

    def forward(self, x):
        return self.net(x)
```

Defining the necessary parameter

```python
# Hyperparameters
lr = 0.0005
batch_size = 64
image_size = 64
channels_img = 1
channels_noise = 512
num_epochs = 4
number_pixel = 64

# For how many channels Generator and Discriminator should use
features_d = 16
features_g = 16
```

Importing the required dataset

```
pickle_in = open("./X_desired_shape_resize_64_10000_each_class.pickle","rb")
X = pickle.load(pickle_in)
```

Converting numpy array into tensor

```
X = torch.Tensor(X)
print(X[0].size())
print(type(X))
print(X.shape)
```

```
torch.Size([1, 64, 64])
<class 'torch.Tensor'>
torch.Size([460000, 1, 64, 64])
```

Filtering dataset

```
print(X[0].shape)
print(X[:][:][0].shape)
```

```
torch.Size([1, 64, 64])
torch.Size([1, 64, 64])
```

```
print(type(X))
print(X.shape)
print(X[0].shape)
```

```
<class 'torch.Tensor'>
torch.Size([460000, 1, 64, 64])
torch.Size([1, 64, 64])
```

```
for i in range(0, 460000,2000):
  temp = X[i]
  plt.imshow(temp[:][:].reshape([number_pixel,number_pixel]), cmap='gray')
  if i== 6000:
    break
```

Feature scaling

```
def norm(x_r):
    x_n = (2.0*x_r/255.0 -1)
    return x_n
```

```
def denorm(x_r):
    x_n = 255.0 *(x_r + 1) / 2.0
    return x_n
```

```
for i in range(0, 360000):
  X_new[i] =norm(X_new[i])
```

```
X_new[0,:, 10:15, 10:15]
```

```
tensor([[[-1., -1., -1., -1., -1.],
         [-1., -1., -1., -1., -1.],
         [-1., -1., -1., -1., -1.],
         [-1., -1., -1., -1., -1.],
         [-1., -1., -1., -1., -1.]]])
```

```
torch.min(X_new[0]), torch.max(X_new[0])
```

```
(tensor(-1.), tensor(1.))
```

```
dataloader = DataLoader(X_new, batch_size=batch_size, shuffle=True)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
G_losses = []
D_losses = []

lossG_real_for_plot = []
lossD_real_for_plot = []
```

```
# Create discriminator and generator
netD = Discriminator(channels_img, features_d).to(device)
netG = Generator(channels_noise, channels_img, features_g).to(device)

# Setup Optimizer for G and D
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(0.5, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(0.5, 0.999))

netG.train()
netD.train()

criterion = nn.BCELoss()


real_label = 1
fake_label = 0

fixed_noise = torch.randn(64, channels_noise, 1, 1).to(device)
writer_real = SummaryWriter(f"runs/GAN_NEPALI/test_real")
writer_fake = SummaryWriter(f"runs/GAN_NEPALI/test_fake")
step = 0
```

```
CHECKPOINT_FILENAME = "./checkPoint_holder/my_checkpoint.pth.tar"
def save_checkpoint(state,filename=CHECKPOINT_FILENAME):
  print("******************************Saving check point ******************************")
  torch.save(state,filename)
```

```
print("Starting Training...")

for epoch in range(num_epochs):
    for batch_idx, (data) in enumerate(dataloader):
        data = data.to(device)

        ### Train Discriminator: max log(1 - D(G(z)))
        netD.zero_grad()
        label = (torch.ones(batch_size) * 0.9).to(device)
        output = netD(data).reshape(-1)
        lossD_real = criterion(output, label)
        D_x = output.mean().item()

        noise = torch.randn(batch_size, channels_noise, 1, 1).to(device)
        fake = netG(noise)
        label = (torch.ones(batch_size) * 0.1).to(device)

        output = netD(fake.detach()).reshape(-1)
        lossD_fake = criterion(output, label)

        lossD = lossD_real + lossD_fake
        lossD.backward()
        optimizerD.step()


        ### Train Generator: max log(D(G(z)))
        netG.zero_grad()
        label = torch.ones(batch_size).to(device)
        output = netD(fake).reshape(-1)

        lossG = criterion(output, label)
        lossG.backward()
        optimizerG.step()



        # Print losses ocassionally and print to tensorboard
        if batch_idx % 50 == 0:
            step += 1
            print(f"Epoch [{epoch}/{num_epochs}] Batch {batch_idx}/{len(dataloader)} \Loss D: {lossD:.4f}, loss G: {lossG:.4f} D(x): {D_x:.4f}")
            SummaryWriter('runs/gen_loss').add_scalar('generator loss with epoch', lossG/100.0, (epoch*len(dataloader) + batch_idx))
            SummaryWriter('runs/dis_loss').add_scalar('discriminator loss with epoch', lossD/100.0, epoch*len(dataloader) + batch_idx)

            # to save the model
#             checkpoint = {
#                         'state_dict_g': netG.state_dict(),
#                         'optimizer_g': optimizerG.state_dict(),
#                         'state_dict_d': netG.state_dict(),
#                         'optimizer_d': optimizerG.state_dict(),
#                         'loss_g': lossG,
#                         'loss_d': lossD,
#                         'd_out': D_x

#                         }
            #save_checkpoint(checkpoint)
```

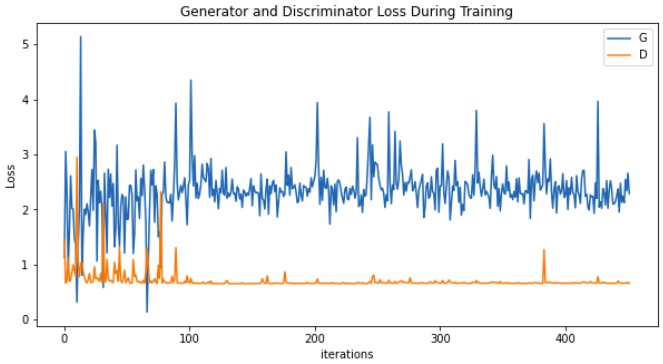
```
#                         }
          #save_checkpoint(checkpoint)



          # Save Losses for plotting later
          G_losses.append(lossG.item())
          D_losses.append(lossD.item())

          lossG_real_for_plot.append(lossD_real.item())
          lossD_real_for_plot.append(lossD_fake.item())


          with torch.no_grad():
              fake = netG(fixed_noise)
              # torchvision.utils.make_grid(tensor: Union[torch.Tensor, List[torch.Tensor]], nrow: int = 8, padding: int = 2, normalize: bool = False, range: Optional[Tuple[int, int]] = No
              img_grid_real = torchvision.utils.make_grid(data[:32], normalize=True)
              img_grid_fake = torchvision.utils.make_grid(fake[:32], normalize=True)
              writer_real.add_image("Nepali letter Real Images", img_grid_real, global_step=step)
              writer_fake.add_image("Nepali letter Fake Images", img_grid_fake, global_step=step)
```

```
Starting Training...
Epoch [0/4] Batch 0/5625 \Loss D: 1.4393, loss G: 1.1236 D(x): 0.4052
Epoch [0/4] Batch 50/5625 \Loss D: 0.6600, loss G: 3.0540 D(x): 0.8749
Epoch [0/4] Batch 100/5625 \Loss D: 0.6908, loss G: 2.3402 D(x): 0.9512
Epoch [0/4] Batch 150/5625 \Loss D: 1.1187, loss G: 0.9304 D(x): 0.5127
Epoch [0/4] Batch 200/5625 \Loss D: 0.6938, loss G: 1.6391 D(x): 0.9205
Epoch [0/4] Batch 250/5625 \Loss D: 0.7199, loss G: 2.6099 D(x): 0.7961
Epoch [0/4] Batch 300/5625 \Loss D: 0.8543, loss G: 2.0138 D(x): 0.7263
Epoch [0/4] Batch 350/5625 \Loss D: 0.9953, loss G: 2.0050 D(x): 0.7332
Epoch [0/4] Batch 400/5625 \Loss D: 0.8900, loss G: 1.4683 D(x): 0.6885
Epoch [0/4] Batch 450/5625 \Loss D: 0.7841, loss G: 1.3633 D(x): 0.7125
Epoch [0/4] Batch 500/5625 \Loss D: 2.9512, loss G: 0.3163 D(x): 0.0675
Epoch [0/4] Batch 550/5625 \Loss D: 1.0097, loss G: 1.3355 D(x): 0.6642
Epoch [0/4] Batch 600/5625 \Loss D: 0.7755, loss G: 2.0374 D(x): 0.8930
Epoch [0/4] Batch 650/5625 \Loss D: 1.0247, loss G: 5.1385 D(x): 0.9580
Epoch [0/4] Batch 700/5625 \Loss D: 1.0297, loss G: 0.8040 D(x): 0.5269
Epoch [0/4] Batch 750/5625 \Loss D: 0.8669, loss G: 1.5792 D(x): 0.6827
Epoch [0/4] Batch 800/5625 \Loss D: 0.7617, loss G: 2.0067 D(x): 0.7344
Epoch [0/4] Batch 850/5625 \Loss D: 0.6970, loss G: 1.9084 D(x): 0.8147
Epoch [0/4] Batch 900/5625 \Loss D: 0.6650, loss G: 2.1026 D(x): 0.8915
Epoch [0/4] Batch 950/5625 \Loss D: 0.7232, loss G: 1.9502 D(x): 0.7695
Epoch [0/4] Batch 1000/5625 \Loss D: 0.8370, loss G: 1.6991 D(x): 0.7615
Epoch [0/4] Batch 1050/5625 \Loss D: 0.6692, loss G: 2.2303 D(x): 0.8799
Epoch [0/4] Batch 1100/5625 \Loss D: 0.6785, loss G: 2.7231 D(x): 0.9318
Epoch [0/4] Batch 1150/5625 \Loss D: 0.6884, loss G: 1.9825 D(x): 0.8207
Epoch [0/4] Batch 1200/5625 \Loss D: 0.9571, loss G: 3.4437 D(x): 0.9433
Epoch [0/4] Batch 1250/5625 \Loss D: 0.7408, loss G: 3.2087 D(x): 0.9427
Epoch [0/4] Batch 1300/5625 \Loss D: 0.7591, loss G: 1.0599 D(x): 0.7332
Epoch [0/4] Batch 1350/5625 \Loss D: 0.7289, loss G: 2.5309 D(x): 0.9391
Epoch [0/4] Batch 1400/5625 \Loss D: 0.6948, loss G: 2.1213 D(x): 0.8063
Epoch [0/4] Batch 1450/5625 \Loss D: 0.8348, loss G: 2.3322 D(x): 0.8501
Epoch [0/4] Batch 1500/5625 \Loss D: 0.6782, loss G: 2.0167 D(x): 0.8539
Epoch [0/4] Batch 1550/5625 \Loss D: 2.1129, loss G: 0.5816 D(x): 0.8745
```

```
print(netD)
print(netG)
```

```
Discriminator(
  (net): Sequential(
    (0): Conv2d(1, 16, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (1): LeakyReLU(negative_slope=0.2)
    (2): Conv2d(16, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (3): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2)
    (5): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (6): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2)
    (8): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (9): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2)
    (11): Conv2d(128, 1, kernel_size=(4, 4), stride=(2, 2))
    (12): Sigmoid()
  )
)
Generator(
  (net): Sequential(
    (0): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(1, 1))
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU()
    (6): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (7): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU()
    (9): ConvTranspose2d(64, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (10): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU()
    (12): ConvTranspose2d(32, 1, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (13): Tanh()
  )
)
```

```
plt.figure(figsize=(10,5))
plt.title("Generator and Discriminator Loss During Training")
plt.plot(G_losses,label="G")
plt.plot(D_losses,label="D")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()
```




```
# plt.figure(figsize=(10,5))
# plt.title("Generator and Discriminator Loss During Training")
# plt.plot(lossG_real_for_plot,label="G_real_score")
# plt.plot(lossD_real_for_plot,label="D_fake_score")
# plt.xlabel("iterations")
# plt.ylabel("score")
# plt.legend()
# plt.title('Score in GAN');
# plt.show()
```

```
fake = netG(fixed_noise)
print(fake.shape)
```

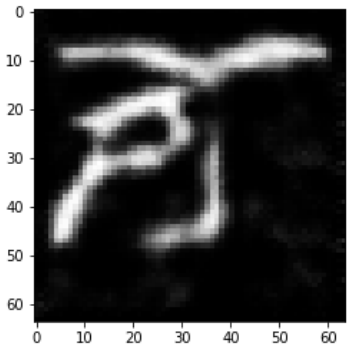
```
torch.Size([64, 1, 64, 64])
```

```
print(fake[0][0].shape)
print(type(fake[0][0]))
```

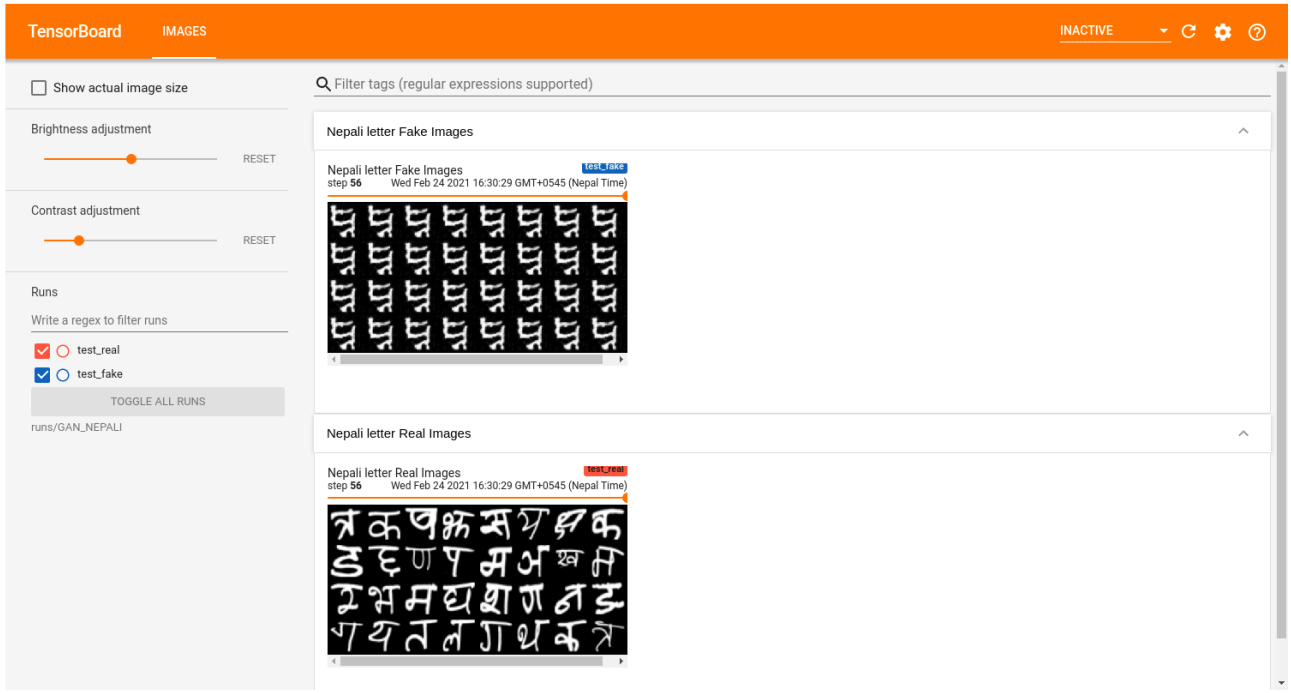
```
torch.Size([64, 64])
<class 'torch.Tensor'>
```

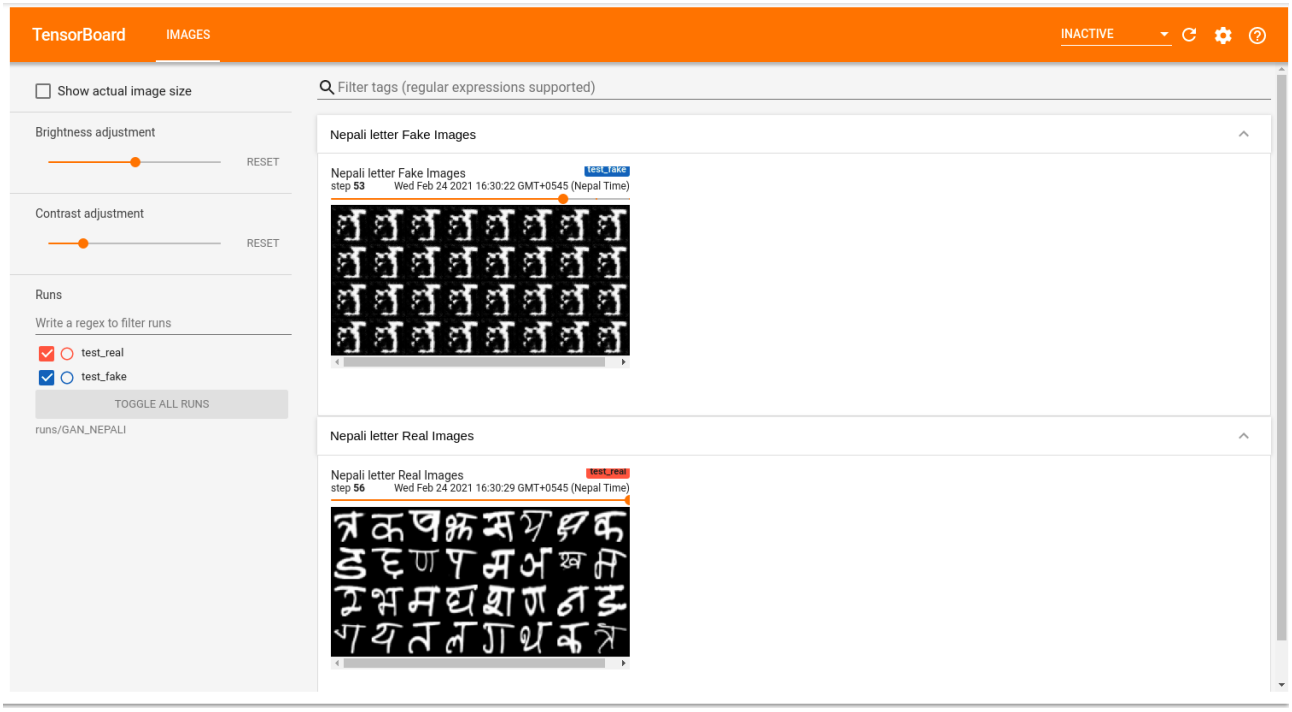
```
plt.imshow(fake[10][0].cpu().data.numpy(),  cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7f84e2358080>
```



B Outcomes


TensorBoard
IMAGES
INACTIVE
Show actual image size
Brightness adjustment
RESET
Contrast adjustment
RESET
Runs
Write a regex to filter runs
test_real
test_fake
TOGGLE ALL RUNS
runs/GAN_NEPALI
Filter tags (regular expressions supported)
Nepali letter Fake Images
Nepali letter Fake Images
step 56
Wed Feb 24 2021 16:30:29 GMT+0545 (Nepal Time)
test_fake
Nepali letter Real Images
Nepali letter Real Images
step 56
Wed Feb 24 2021 16:30:29 GMT+0545 (Nepal Time)
test_real


TensorBoard
IMAGES
INACTIVE
Show actual image size
Brightness adjustment
RESET
Contrast adjustment
RESET
Runs
Write a regex to filter runs
test_real
test_fake
TOGGLE ALL RUNS
runs/GAN_NEPALI
Filter tags (regular expressions supported)
Nepali letter Fake Images
Nepali letter Fake Images
step 53
Wed Feb 24 2021 16:30:22 GMT+0545 (Nepal Time)
test_fake
Nepali letter Real Images
Nepali letter Real Images
step 56
Wed Feb 24 2021 16:30:29 GMT+0545 (Nepal Time)
test_real