EEE443 Neural Network Project Final Report
Text to Image Generation

Mustafa Yasir Altunhan[1], Berk Demirkan[2], Umutcan Aral[2], Tuğcan Ünalan[2]

(Date: 11.06.2023)

[1]Computer Science Department, Bilkent University, 06800 Ankara, Turkey
[2]Electrical and Electronic Engineering, Bilkent University, 06800 Ankara, Turkey

Abstract - *In this project, A text-to-image synthesis assignment is provided. Deep Neural Networks (DNN) are utilized in conjunction with sentence embedding and image processing techniques to complete the task. The definition of image to text synthesis, the possible problems and the possible issues that can come with these problems is thoroughly described and literature review on how to solve these problems and overcome the hardships of the project is done in the introduction section. The methodologies that can be used for the project are covered in the methods section. The simulation outcomes for training and test data with various settings are analyzed in the results section. The broad overview of the suggested approaches, their findings, and benefits and drawbacks are explored in the discussion section.*

1. Introduction

The scientific community is nowadays very interested in text-to-image synthesis, which intends to automatically generate images based on the given text description. The creation of visually realistic and semantically appropriate images can be listed as one of the main difficulties in text-to-image synthesis. In order to handle the local and global visual information and generate more visually realistic and compatible images, a number of frameworks were proposed because image data distribution is in high-dimensional space and cannot be directly estimated. Contrarily, semantic consistency refers to the agreement between the content of the image and the text description [1].

In this project, we are going to use three methods which are DCGAN, GAN-CLS with Skip-Thought Vectors and StackGAN that will enable us to generate realistic images from the given texts. We will train DCGAN and its encoder from scratch whereas due to the limited time and resources, we will use pre-trained models and encoders for both GAN-CLS with Skip-Thought Vectors and StackGAN as our backbone models and fine-tune them for our flowers dataset. We intentionally have chosen our pre-trained models to have already been trained on Oxford flowers dataset so that we do not need a high number of iterations to fine-tune the network for our dataset since the backbone training and fine-tuning is done to the same dataset.

2. Methods

2, A. Dataset

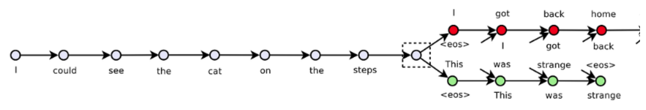
The Oxford Flowers Dataset was developed by the University of Oxford, a prestigious organization at the forefront of ground-breaking research in a variety of fields. It was created specifically by Nilsback and Zisserman, who were motivated by the enormous variety and aesthetic appeal of flowers seen in natural ecosystems. Their goal was to create a dataset that included a diversity of flower species and would serve as a useful tool for developing and comparing picture recognition systems [2][3]. This dataset consists of 102 flower categories where each category has between 40 to 258 images. Captions for a given image are needed to get the relation between captions and images so text_c10 directory is used for this purpose. It has multiple caption text files where each file has 10 captions for its corresponding image.

2, B. Models

Before explaining how the experiments were set up, first the models that were used in the experiments are presented.

Skip-Thought Vectors

In order to process a caption and produce the image, we first need to get a vectorial representation of the caption. Some captions were long whereas others were short but at the end, we need to get a fixed size vector to process a caption. Therefore, it was decided that Skip-Thought vectors would be a good option. Skip-Thought vectors were developed by Ryon Kiros and other researchers whose initial aim was to be able to create useful embeddings for sentences that are not just tuned for a single task and do not need labeled data. Skip-Thought vectors were created using encoder-decoder models. The encoder takes a sentence as input and outputs a vector. Multiple encoder types are available which are uni-skip, bi-skip and combine-skip. Uni skip reads the sentence forwards whereas bi-skip reads the sentence forwards and backwards and then concatenates the results. There are two decoders both of which take the vector as input. The first decoder tries to predict the previous sentence whereas the second decoder tries to predict the next sentence . Both the encoder and decoder are constructed using RNNs. By this way the model can predict the relation between the words which gives a semantic perspective to a sentence [4][5]. A diagram indicating the input sentence and the predicted previous and next sentences are given below:



Given a sentence (the grey dots), skip-thought attempts to predict the preceding sentence (red and the next sentence (green dots). Figure from the paper.

Figure 1 - Skip-Thought Vectors Prediction [4]

GAN (Generative Adversarial Network)

GAN consists of two main components named generator and discriminator. The generator's job is to create new data that resembles a target distribution such as a realistic cat whereas the discriminator tries to learn how to distinguish real data from generated data that look like samples from the target distribution. Throughout the training process, the generator tries to learn how to trick the discriminator whereas the discriminator tries to learn how to detect a generated image from real ones [6]. Concurrently, D and G play the optimization game given below:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim P_{\text{real}}(x)}[\log(D(x))] + \mathbb{E}_{z\sim P_{\text{noise}}(z)}[\log(1-D(G(z)))]$$

The Objective Function of GAN [5]

The figure below represents the general architecture of a GAN:

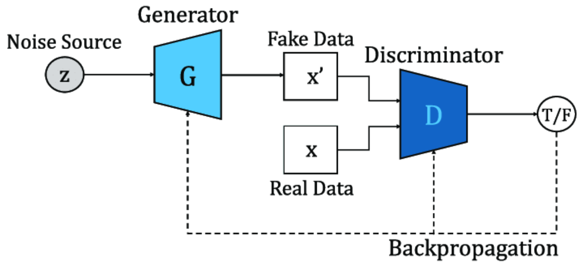
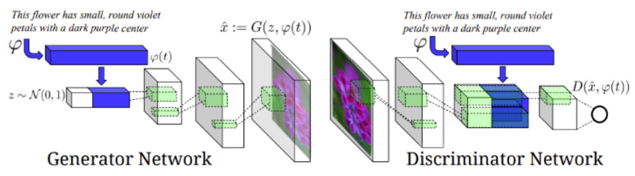


Figure 2 - GAN Architecture [6]

GAN-CLS (Generative Adversarial Network Conditional Latent Space)

The GAN-CLS algorithm combines two important concepts that are GAN and classification. GAN can be used to produce a specific type of image such as a human face given a noise vector as input. However, in this project, we are required not to only produce a realistic image but an image that also resembles its caption. Therefore, the generator should learn to produce a realistic image that is highly correlated with the given caption whereas the discriminator should be able to distinguish not only fake images from real ones but also whether given images are highly correlated to the given caption. The general architecture is given below:



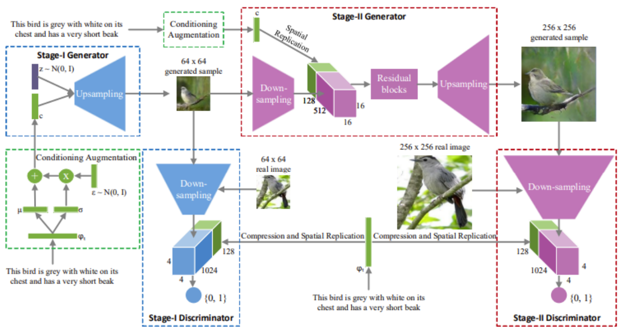
This flower has small, round violet petals with a dark purple center
φ
$\varphi(t)$
$z \sim \mathcal{N}(0,1)$
$\hat{x} := G(z, \varphi(t))$
This flower has small, round violet petals with a dark purple center
φ
$D(\hat{x}, \varphi(t))$
Generator Network
Discriminator Network

Figure 3 - GAN-CLS Architecture [5]

As seen from figure 3, the input for the generator is constructed by concatenating the vector representation of the caption together with a randomly produced noise vector. After the discriminator image is down sampled the produced image, the resulting output is concatenated with the vector representation of the input. After that, the output of D(x,Φ(t)) is calculated. The aim of the discriminator is that looking at the vector representation of the caption, it should be able to distinguish a real image with the right label from a generated image with arbitrary caption or generated image mismatching its caption [6].

StackGAN

Instead of producing a realistic high-resolution image directly, StackGAN model tries to accomplish this using two GANs named Stage-I GAN and Stage-II GAN. [8].



This bird is grey with white on its chest and has a very short beak
Conditioning Augmentation
Stage-II Generator
c
Spatial Replication
256 x 256 generated sample
Stage-I Generator
z ~ N(0, I)
Upsampling
c
64 x 64 generated sample
Down-sampling
128
512
16
16
Residual blocks
Upsampling
Conditioning Augmentation
ε ~ N(0, I)
μ
σ
φt
This bird is grey with white on its chest and has a very short beak
Down-sampling
64 x 64 real image
256 x 256 real image
Down-sampling
Compression and Spatial Replication
Compression and Spatial Replication
128
4
1024
4
{0, 1}
Stage-I Discriminator
φt
This bird is grey with white on its chest and has a very short beak
128
1024
4
4
{0, 1}
Stage-II Discriminator

Figure 4 - StackGAN architecture [7]

Stage-I GAN

It sketches the primitive shape and the basic colors of the shape given the text embedding of the caption [8]. The sequence of steps performed in Stage-I GAN are given below.

The text embedding of the caption is obtained using encoder. To address the high dimensionality problem of the text, conditioning augmentation is used. It randomly samples latent variables from a gaussian distribution depending on the text embedding. The generator takes the text embedding and conditioned Gaussian variables to generate a low-resolution image that more or less captures the primitive shape of a realistic image. The generator is trained to minimize the $L(G_0)$, which includes adversarial loss and regularization term to enforce smoothness. On the other hand, the objective of discriminator is to be able to distinguish between real images and fake images. Therefore, it maximizes the objective function $L(D_0)$ which consists of loss for real image and generated image.

Stage-II GAN

It corrects the defects in the low-resolution image and completes details of the object by reading the text description again [7]. Its goal is to produce high resolution realistic images given low resolution images generated in stage I. The sequence of steps performed in Stage-II GAN are given below.

The generator takes the low-resolution image from stage-I, text embedding and gaussian conditioning variables sampled from the same pre-trained text encoder used in stage-I to generate a high-resolution image. Like stage I, the generator is trained to minimize the objective function L(G) which includes adversarial loss and regularization term. The discriminator is trained to maximize the objective function L(D) so that it can distinguish between real and fake images. L(D) consists of loss for real images and generated images.

3. Metrics

3, A GAN Losses

The two parts of a GAN network have distinct loss functions that are used during training to optimize their respective performances. The generator's loss quantifies how well it is able to fool the discriminator. The discriminator's loss on the other hand measures its ability to discern real data from the generator's synthetic data.

The generator's loss function is typically formulated as a binary cross-entropy loss. This loss penalizes the generator when the discriminator accurately distinguishes its generated samples from the real ones. The generator aims to minimize this loss to better trick the discriminator. Conversely, the discriminator's loss function is also a binary cross-entropy loss, but from a different perspective. It tries to maximize its ability to correctly classify both real and fake samples. As such, it seeks to minimize its own loss function by accurately telling real samples from synthetic ones. This creates a competitive dynamic between the generator and discriminator.

However, these loss values alone aren't necessarily a good indication of performance of the overall GAN during training. The loss functions of the generator and the discriminator in GANs are not designed to decrease over time in the way they would in a typical supervised learning scenario. Instead, they represent a minimax game where the discriminator and generator are continually improving to outperform each other. Moreover, it's possible for a GAN to experience mode collapse, where the generator only produces a limited variety of samples, even though the discriminator's loss might be high and the generator's loss might be low. As such, the GAN losses alone are not sufficient to evaluate the quality of the generated samples or the diversity of the generated dataset. Therefore, other evaluation methods such as the Fréchet inception distance (FID) are usually used to assess the quality and diversity of the generated samples.

3,B FID (Fréchet inception distance)

FID is a metric for evaluating the quality of generated images and was specifically developed to evaluate the performance of GANs [8] [9]. The score was proposed as an improvement over the existing inception score (IS).

The FID score is based on the Fréchet distance, which is a measure of dissimilarity between two multivariate Gaussian distributions. The FID score compares the statistics of feature representations extracted from real and generated images using inception network. Here is how FID score is calculated:

First, a pre-trained inception vector is used as a feature extractor. The network is typically trained on a large dataset of real images and has learned to extract meaningful features from them. After real and generated images are passed through the network, activations for one of the intermediate layers are extracted to serve as representative features of the images. After that, the mean and covariance of generated and real images are calculated. After that, Fréchet distance between two gaussian distributions are calculated using the means and covariance of generated images and fake images. Finally, the FID score is obtained by scaling Fréchet distance so that Fréchet distance value is normalized.

A lower FID score indicated that generated images are closer in distribution to the real images which implies that the generated images are realistic and have high quality, diversity. However, higher FID score indicated a high dissimilarity between two given gaussian distributions so it implied GAN that was trained has poor performance and does not produce high resolution realistic images.

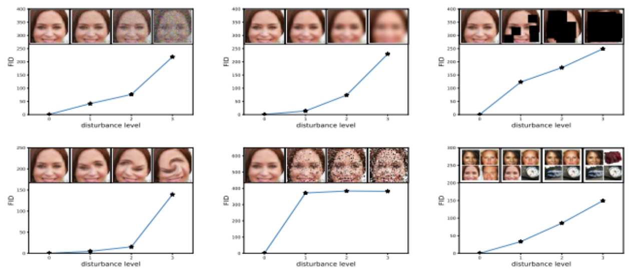


Figure 5 - FID Scores of Human Faces having different amount of distortion [9]

4. Results and Analysis

4, A DCGAN

The progressive improvement of image generation by the DCGAN was illustrated by the generated samples across multiple epochs.

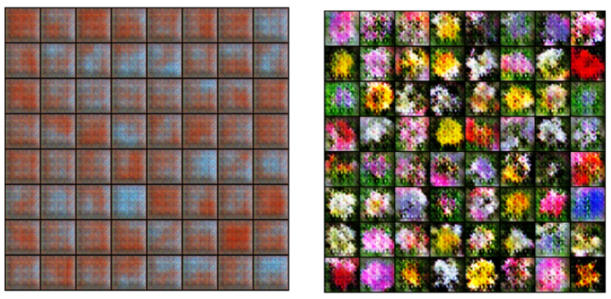


Figure 6: Epoch 1 (Left) and Epoch 10 (Right) Generated Images

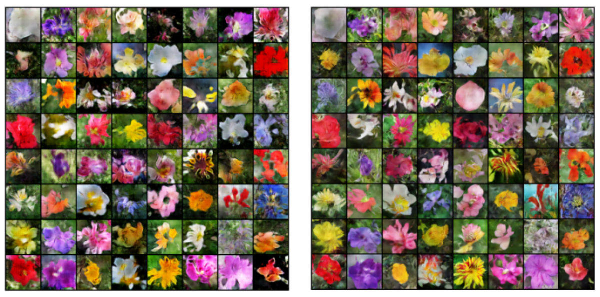


Figure 7: Epoch 50 (Left) and Epoch 200 (Right) Generated Images

The first epoch samples were quite noisy, exhibiting no structural semblance to the real samples. However, by epoch 10, discernable patterns began to emerge and there was a noticeable improvement in the images. Structures resembling real samples were identifiable, although they still appeared quite noisy and had unrelated patterns. Upon reaching epoch 50, the images showed a marked advancement in quality, with more detailed structures and less noise. The generated images are in quality, with more detailed structures and less noise. The target categories were much clearer, with relatively accurate representation, suggesting a more accurate mapping from the latent space to the data distribution. At epoch 200, the images were even more refined and showed a high degree of similarity with the real images, demonstrating the effectiveness of the DCGAN-CLS model in learning complex data distributions. The model seemed to capture the major structural and textual components of the images, denoting a good understanding of the underlying data distribution.

The generator and discriminator losses plotted over the training iterations provided insights into the adversarial relationship between the two network components.

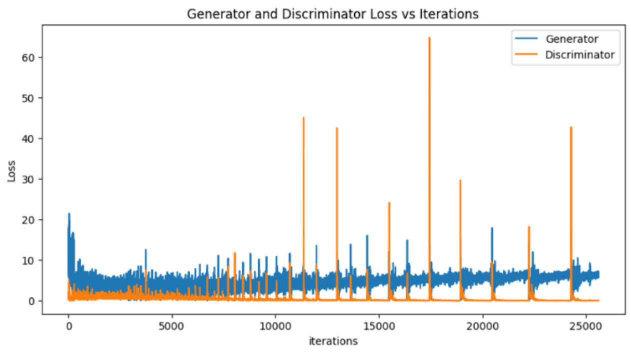


Figure 8: Generator and Discriminator Losses During Training

Initially, the discriminator loss was relatively low, indicating its superiority over the generator. On the other hand, the generator's loss was initially high, reflecting the poor quality of generated images. With increased training iterations, the generator's loss decreased, illustrating its increasing ability to produce more realistic images.

Starting at roughly 10,000 iterations the generator and discriminator losses started to oscillate around their respective stable values with discriminator loss being lower than the generator loss.

The Fréchet inception distance (FID) Scores plotted over training iterations offer an objective metric for evaluating the quality and diversity of generated images.

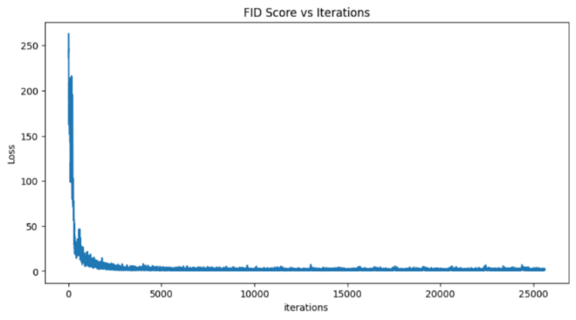


Figure 9: FID Score During Training

High initial FID scores corresponded to the poor image quality at early epochs. The FID score dramatically dropped as training proceeded, indicating an improvement in the quality of the generated images. By epoch 200, the FID score was relatively low, denoting the high quality of the generated images and the model's capability to accurately represent the real data distribution.

Due to the primitive nature of the network, the process of sampling the generator with prompts works yields successful results intermittently, requiring multiple attempts to generate a relevant image. This suggests a suboptimal connectivity between the conditional latent space and the generated image. Nevertheless, it's important to note that the sampling process has indeed improved compared to a GAN that lacks a conditional latent space.

The low FID scores of the network and high quality of the generated images together with the difficulty of prompted generation may be an indication that the model behaves more like a standard GAN.

4, B GAN-CLS with Skip-Thought Vectors

This model consists of two parts: GAN-CLS as its core training network and skip-thought vectors to embed captions as vectors. For the GAN-CLS part, it is almost identical to the DCGAN implementation. The core difference between DCGAN and this implementation is the usage of pretrained skip-thought vectors [5]. The goal of using skip-thought vectors is to generate a sentence embedding that captures the meaning and the context of the sentence. The problem with using the DCGAN model was that we were not able to train a perfectly working embedding network that captures the semantic and meaning of the sentence. Thus, we decided to use pre-trained skip-thought vectors that capture the semantic of the given caption very well. We did not have enough time to train the GAN-CLS algorithm from scratch for more than 200 epochs. Therefore, we decided to use a pre-trained model that was specifically trained for the flowers dataset for 200 epochs as our backbone network and fine-tuned it for 35 epochs [11]. Since it was already trained for flowers dataset, the fine-tuning process was saturated, performance increased very slowly after 30 epochs, so we stopped the algorithm early.

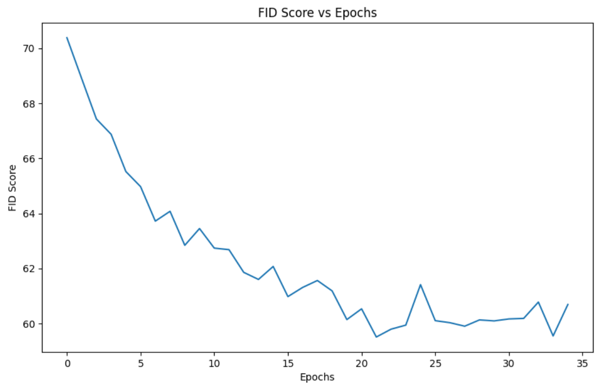


Figure 10: FID Score of GAN-CLS During Training

4, C Stack GAN

This model consists of two parts: StackGan as its core training network and char-CNN-RNN text embeddings. Realizing how well we performed using pre-trained embedding networks to convert captions into vectorial representations in GAN-CLS, we decided that it would be a good idea to use pre-trained embedding networks in StackGAN as well. However, unlike the pretrained skip-thought vectors where they were trained on a large scale of texts from many contexts, we decided to use vector embeddings specifically generated for flowers dataset [12]. After obtaining vectors for each caption, we performed conditioning augmentation and after that, we concatenated the input with the noise vector. Realizing that we have a limited amount of time and resources whereas StackGAN needs to be trained for both Stack1 and Stack2 where each stack needs more than 300 for good results, we decided to use a pretrained model where each stack was already trained for 600 epochs specifically for the flowers dataset [12]. Fine-tuning was performed for approximately 35 iterations since the network was already trained for a large number of iterations for the same task. As you can see from the FID score given below, the FID score stopped changing significantly after iteration 25 so the training stopped early.

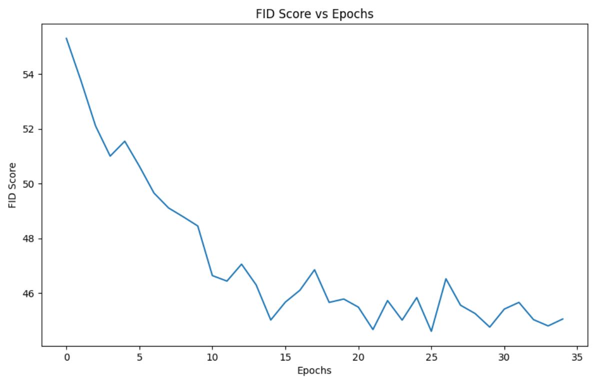


Figure 11: FID Score of StackGAN During Training

5. Example Outputs

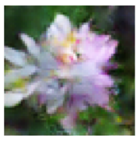
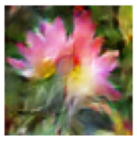
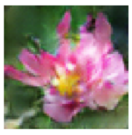
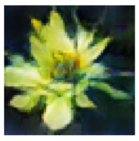
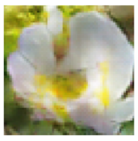
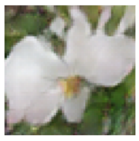
	DCGAN (multiple tries)	GAN-CLS with Skip-Thought Vectors	StackGAN
This flower has pink petals and a yellow center			
The petals of this flower are white in color and its center is yellow			

Table 1: The Outputs of Three Different Models Given Captions

6. Discussion

6,A.Comparison Of Networks

As you can see from the pictures given above, for the DCGAN, we were able to produce images that more or less resemble the realistic images. In most of the papers we have seen, researchers use an encoder that was already trained with a massive amount of data to map captions into a vector space. However, in our implementation, we have not used a pre-trained encoder but rather tried to train our encoder together with the GAN. Since, we have only used the captions of the flowers to train our encoder, expectedly, the encoder was not able to learn the semantic features of captions fully. Therefore, although we were able to produce realistic images for flowers, many times they were not very highly related with the captions provided.

For the GAN-CLS algorithm with skip-thought vectors, the images produced by our network were both realistic and related to its prompts. That was what we expected because in this model we used pre-trained skip-thought vectors that were trained on hundreds of texts and novels on the internet [reference]. By using skip-thought vectors, we were able to capture the semantics of the captions and put it in a fixed dimensional vector. For the GAN-CLS model, we did not have enough time to train it from scratch so we used a pretrained model and fine-tuned it over 35 iterations. Therefore, we obtained very realistic produced images that we would not be able to get if we were to train from scratch as we did not have enough time to train for over 200 iterations.

For the StackGan implementation, we got the best FID score and the produced images were the most realistic ones. We believe it was due to two reasons. First one is that instead of using one GAN architecture, 2 GANs stacked above each other helped the network learn how to generate the image in two stages rather than one. In the first stage, the network tried to learn the general characteristic shape of the flower image given caption whereas the second stage tried to learn how to convert a low resolution image with a not realistic background to a high resolution with a realistic background which looks more realistic. Therefore, using a two-staged architecture facilitated the task of the GAN and enabled it to focus different sub-task in different stages which improved the performance. The second reason why we got the best performance is that we used a pre-trained model that was trained over 600 iterations for both stages and fine tuned it with 35 iterations. Since the pre-trained network was already trained for flowers dataset, it facilitated fine-tuning and fine-tuning was stopped early as it stopped to increase its performance significantly after iteration 25.

Although DCGAN gave the best FID score, it was chosen to be the worst among 3 models because it could not give flower pictures that were very related to the prompt. The reason we believe it got this FID score is that since the FID score looks for similarity between two gaussian distributions, it did not consider whether the prompts were related to the images produced or not.

7. References

[1] Author links open overlay panelYong Xuan Tan a et al., “Text-to-image synthesis with self-supervised learning,” Pattern Recognition Letters, https://www.sciencedirect.com/science/article/abs/pii/S0167865522001064. (accessed Jun. 11, 2023).

[2] “Flower datasets,” Visual Geometry Group - University of Oxford, https://www.robots.ox.ac.uk/~vgg/data/flowers/ (accessed Jun. 11, 2023).

[3] L. E. M. Yusnu, “Oxford 102 flower dataset,” Kaggle, https://www.kaggle.com/datasets/nunenuh/pytorch-challange-flower-dataset (accessed Jun. 11, 2023).

[4] “Deep Learning Reading Group: Skip-thought vectors,” KDnuggets, https://www.kdnuggets.com/2016/11/deep-learning-group-skip-thought-vectors.html (accessed Jun. 11, 2023).

[5] R. Kiros et al., “Skip-thought vectors,” arXiv.org, https://arxiv.org/abs/1506.06726 (accessed Jun. 11, 2023).

[6] S. Reed et al., “Generative adversarial text to image synthesis,” arXiv.org, https://arxiv.org/abs/1605.05396 (accessed Jun. 11, 2023).

[7] A. M. Tonello, N. A. Letizia, D. Righini, and F. Marcuzzi, “Machine learning tips and tricks for power line communications,” *IEEE Access*, vol. 7, pp. 82434–82452, 2019. doi:10.1109/access.2019.2923321

[8] H. Zhang et al., “Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks,” arXiv.org, https://arxiv.org/abs/1612.03242 (accessed Jun. 11, 2023).

[9] J. Brownlee, “How to implement the Frechet Inception Distance (FID) for evaluating Gans,” MachineLearningMastery.com, https://machinelearningmastery.com/how-to-implement-the-frechet-inception-distance-fid-from-scratch (accessed Jun. 11, 2023).

[10] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter, “Gans trained by a two time-scale update rule converge to a local Nash equilibrium,” arXiv.org, https://arxiv.org/abs/1706.08500 (accessed Jun. 11, 2023).

[11] Paarthneekhara, “Paarthneekhara/text-to-image: Text to image synthesis using thought vectors,” GitHub, https://github.com/paarthneekhara/text-to-image/tree/master (accessed Jun. 11, 2023).

[12] Hanzhanggit, “Hanzhanggit/Stackgan,” GitHub, https://github.com/hanzhanggit/StackGAN (accessed Jun. 11, 2023).

8. Appendices

Appendix A: “dcgan.ipynb”

```
[1]
%matplotlib inline
%env CUBLAS_WORKSPACE_CONFIG=:4096:8

import argparse
import os
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as datasets
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML
from tqdm import tqdm
from torch.utils.data import Dataset, DataLoader
import pickle
from skimage import io
from pytorch_fid import fid_score, inception
from PIL import Image
import matplotlib
from torchmetrics.image.fid import FrechetInceptionDistance
from torchsummary import summary

print("Random Seed: ", 999)
random.seed(999)
torch.manual_seed(999)
torch.use_deterministic_algorithms(True) # Needed for reproducible
```

```
results

device = "cuda:0"

[2]
dataroot = "data/flowers102"
batch_size = 128
image_size = 64

n_latent = 100
n_wordemb = 100 # word embedding size
n_textemb = 100 # sentence embedding size
n_vocab = 1004 # vocab size
n_feature = 64 # feature vector size
epochs = 20
learning_rate = 0.0002

[3]
class CustomDataset(Dataset):
    def __init__(self, path_to_pickle, root_dir, transform=None):
        assert os.path.exists(path_to_pickle)
        with open(path_to_pickle, "rb") as test_fh:
            self.image_cap_pairs = pickle.load(test_fh)

        self.root_dir = root_dir
        self.transform = transform

    def __len__(self):
        return len(self.image_cap_pairs)

    def __getitem__(self, idx):
        if torch.is_tensor(idx):
            idx = idx.tolist()

        right_img_path = os.path.join(self.root_dir,
self.image_cap_pairs[idx, 0])
        with open(right_img_path, "rb") as fh:
          right_image = Image.open(fh)
          right_image = right_image.convert("RGB")
          right_image = self.transform(right_image)

       wrong_img_path = os.path.join(self.root_dir,
self.image_cap_pairs[idx, 1])
       with open(wrong_img_path, "rb") as fh:
          wrong_image = Image.open(fh)
          wrong_image = wrong_image.convert("RGB")
          wrong_image = self.transform(wrong_image)

       caption = self.image_cap_pairs[idx, 2]
       caption = torch.tensor(caption).int()

       return right_image, wrong_image, caption

[4]
dataset = CustomDataset(
   "train.pkl",
   "data/flowers/train_64",
   transform=transforms.Compose(
      [
         # transforms.PILToTensor(),
         transforms.Resize(image_size),
         transforms.CenterCrop(image_size),
         transforms.ToTensor(),
         transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
      ]
   ),
)

dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
shuffle=True)


# Plot some training images
real_batch = next(iter(dataloader))
```

```
plt.figure(figsize=(8, 8))
plt.axis("off")
plt.title("Training Images")
plt.imshow(
    np.transpose(
        vutils.make_grid(
            real_batch[0].to(device)[:64], padding=2, normalize=True
        ).cpu(),
        (1, 2, 0),
    )
)
print(len(dataset))

[5]
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)

[6]
class TextEncoder(nn.Module):
    def __init__(self, hidden_dim=100):
        super().__init__()

        self.embed = nn.Embedding(n_vocab, n_wordemb)
        self.gru = nn.GRU(n_wordemb, hidden_dim, batch_first=False)
        self.relu = nn.ReLU()
        self.fc = nn.Linear(hidden_dim, n_textemb)

    def forward(self, text):
        batch_size = text.shape[0]

        embedded = self.embed(text)
        out_gru, _ = self.gru(embedded, None)
        out_relu = self.relu(out_gru[:, -1])
        context_vector = self.fc(out_relu)
        return context_vector.view(batch_size, n_textemb, 1)

[7]
# Generator Code
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.textEncoder = TextEncoder()
        self.layer1 = nn.Sequential(
            nn.ConvTranspose2d(
                n_latent + n_textemb, n_feature * 8, 4, 1, 0, bias=False
            ),
            nn.BatchNorm2d(n_feature * 8),
            nn.ReLU(True),
        )
        self.layer2 = nn.Sequential(
            nn.ConvTranspose2d(n_feature * 8, n_feature * 4, 4, 2, 1,
bias=False),
            nn.BatchNorm2d(n_feature * 4),
            nn.ReLU(True),
        )
        self.layer3 = nn.Sequential(
            nn.ConvTranspose2d(n_feature * 4, n_feature * 2, 4, 2, 1,
bias=False),
            nn.BatchNorm2d(n_feature * 2),
            nn.ReLU(True),
        )
        self.layer4 = nn.Sequential(
            nn.ConvTranspose2d(n_feature * 2, n_feature, 4, 2, 1,
bias=False),
            nn.BatchNorm2d(n_feature),
            nn.ReLU(True),
        )
        self.output = nn.Sequential(
            nn.ConvTranspose2d(n_feature, 3, 4, 2, 1, bias=False), nn.Tanh()
        )

    def forward(self, input_noise, caption=None):
        batch_size = input_noise.size(0)
        if caption is None:
            caption = torch.zeros(batch_size, 17, dtype=int, device=device)

        caption_encoding = self.textEncoder(caption).unsqueeze(-1)
        feature_vector = torch.cat((input_noise, caption_encoding), 1)

        x1 = self.layer1(feature_vector)
        x2 = self.layer2(x1)
        x3 = self.layer3(x2)
        x4 = self.layer4(x3)
        return self.output(x4)

[8]
# Create the generator
netG = Generator().to(device)
netG.apply(weights_init)
summary(netG, (100, 1, 1))

[9]
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        self.textEncoder = TextEncoder()
        self.layer1 = nn.Sequential(
            nn.Conv2d(3, n_feature, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(n_feature, n_feature * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(n_feature * 2),
            nn.LeakyReLU(0.2, inplace=True),
        )
        self.layer3 = nn.Sequential(
            nn.Conv2d(n_feature * 2, n_feature * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(n_feature * 4),
            nn.LeakyReLU(0.2, inplace=True),
        )
        self.layer4 = nn.Sequential(
            nn.Conv2d(n_feature * 4, n_feature * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(n_feature * 8),
            nn.LeakyReLU(0.2, inplace=True),
        )

        self.output = nn.Sequential(
            nn.Conv2d(n_feature * 8 + n_textemb, 1, 4, 1, 0, bias=False),
nn.Sigmoid()
        )

    def forward(self, input_image, caption=None):
        batch_size = input_image.size(0)
        if caption is None:
            caption = torch.zeros(batch_size, 17, dtype=int, device=device)

        caption_encoding =
self.textEncoder(caption).unsqueeze(-1).repeat(1, 1, 4, 4)

        x1 = self.layer1(input_image)
        x2 = self.layer2(x1)
        x3 = self.layer3(x2)
        x4 = self.layer4(x3)

        x4_ = torch.cat((x4, caption_encoding), 1)

        x5 = self.output(x4_)
        return x5

[10]
# Create the Discriminator
netD = Discriminator().to(device)
```

```
netD.apply(weights_init)
summary(netD, (3, 64, 64))

[11]
# Initialize the ``BCELoss`` function
criterion = nn.BCELoss()

# Create batch of latent vectors that we will use to visualize
#  the progression of the generator
fixed_noise = torch.randn(64, n_latent, 1, 1, device=device)

# Establish convention for real and fake labels during training
real_label = 1.
wrong_label = 0.
fake_label = 0.

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

[12]
torch.backends.cudnn.benchmark = True
# Training Loop

# Lists to keep track of progress
img_list = []
G_losses = []
D_losses = []
fids = []

fid = FrechetInceptionDistance(feature=64, normalize=True).to(device)

# For each epoch
for epoch in range(1, epochs + 1):
    # For each batch in the dataloader
```

```
  tqdm_dataloader = tqdm(dataloader)
  for i, data in enumerate(tqdm_dataloader, 1):
    tqdm_dataloader.set_description_str(f"Epoch: {epoch}/{epochs}")

    caption = data[2].to(device) if np.random.random() > 0.1 else None

    ############################
    # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
    ###########################
    ## Train with all-real batch
    netD.zero_grad()
    # Format batch
    real_image = data[0].to(device)
    b_size = real_image.size(0)
    label = torch.full((b_size,), real_label, dtype=torch.float,
device=device)
    # Forward pass real batch through D
    output = netD(real_image, caption).view(-1)
    # Calculate loss on all-real batch
    errD_real = criterion(output, label)
    # Calculate gradients for D in backward pass
    errD_real.backward()
    D_x = output.mean().item()

    ## Train with all-wrong batch
    netD.zero_grad()
    # Format batch
    wrong_image = data[1].to(device)
    # Forward pass real batch through D
    output = netD(wrong_image, caption).view(-1)
    # Calculate loss on all-real batch
    errD_wrong = criterion(output, label)
    # Calculate gradients for D in backward pass
    errD_wrong.backward()
    D_x += output.mean().item()

    ## Train with all-fake batch
```

```
        # Generate batch of latent vectors
        noise = torch.randn(b_size, n_latent, 1, 1, device=device)
        # Generate fake image batch with G
        fake = netG(noise, caption)
        label.fill_(fake_label)
        # Classify all fake batch with D
        output = netD(fake.detach(), caption).view(-1)
        # Calculate D's loss on the all-fake batch
        errD_fake = criterion(output, label)
        # Calculate the gradients for this batch, accumulated (summed) with
previous gradients
        errD_fake.backward()
        D_G_z1 = output.mean().item()
        # Compute error of D as sum over the fake and the real batches
        errD = errD_real + errD_wrong + errD_fake

        # Update D
        optimizerD.step()

        ############################
        # (2) Update G network: maximize log(D(G(z)))
        ###########################
        netG.zero_grad()
        label.fill_(real_label)  # fake labels are real for generator cost
        # Since we just updated D, perform another forward pass of all-fake
batch through D
        output = netD(fake, caption).view(-1)
        # Calculate G's loss based on this output
        errG = criterion(output, label)
        # Calculate gradients for G
        errG.backward()
        D_G_z2 = output.mean().item()
        # Update G
        optimizerG.step()

        fid.update(real_image, real=True)
        fid.update(fake, real=False)
        tqdm_dataloader.set_postfix_str(
            f"Loss_D: {errD.item():.4f}, Loss_G: {errG.item():.4f}, D(x):
{D_x:.4f}, D(G(z)): {D_G_z1:.4f} / {D_G_z2:.4f}"
        )

        # Save Losses for plotting later
        G_losses.append(errG.item())
        D_losses.append(errD.item())

    fids.append(fid.compute().item())
    fid.reset()

    tqdm_dataloader.set_postfix_str(
        f"Loss_D: {errD.item():.4f}, Loss_G: {errG.item():.4f}, D(x):
{D_x:.4f}, D(G(z)): {D_G_z1:.4f} / {D_G_z2:.4f}, fid: {fids[-1]}"
    )

    # Check how the generator is doing by saving G's output on
fixed_noise
    with torch.no_grad():
        fake = netG(fixed_noise).detach().cpu()
    img_list.append(vutils.make_grid(fake, padding=2, normalize=True))

    torch.save(netG.state_dict(), f"model_saves/generator_E{epoch}.pth")
    torch.save(netD.state_dict(),
f"model_saves/discriminator_E{epoch}.pth")
    if fids[-1] < min([*fids[:-1],np.inf]):
        best_generator = f"model_saves/generator_E{epoch}.pth"
        best_discriminator = f"model_saves/discriminator_E{epoch}.pth"

[13]
print(fids)

plt.figure(figsize=(10,5))
plt.title("Generator and Discriminator Loss vs Iterations")
plt.plot(G_losses,label="Generator")
plt.plot(D_losses,label="Discriminator")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()


plt.figure(figsize=(10,5))
plt.title("FID Score vs Iterations")
plt.plot(fids)
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.show()

[14]
netG_best = Generator().to(device)
netG_best.load_state_dict(torch.load(best_generator))
netG_best.eval()

netD_best = Discriminator().to(device)
netD_best.load_state_dict(torch.load(best_discriminator))
netG_best.eval()


# Grab a batch of real images from the dataloader
real_batch = next(iter(dataloader))

# Plot the real images
plt.figure(figsize=(15,15))
plt.subplot(1,2,1)
plt.axis("off")
plt.title("Real Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64],
padding=2, normalize=True).cpu(),(1,2,0)))

# Plot the fake images from the last epoch
```

```
plt.subplot(1,2,2)
plt.axis("off")
plt.title("Fake Images")
plt.imshow(np.transpose(img_list[-1],(1,2,0)))
plt.show()

[15]
netG_current = Generator().to(device)
netG_current.load_state_dict(torch.load("model_saves/generator_E200.p
th"))
netG_current.eval()

netD_current = Discriminator().to(device)
netD_current.load_state_dict(torch.load("model_saves/discriminator_E2
00.pth"))
netG_current.eval()

[16]
caption = torch.zeros((1, 17), dtype=int, device=device)

noise = torch.randn(1, n_latent, 1, 1, device=device)
with torch.no_grad():
    fake_images = netG(noise, caption).detach().cpu()
plt.figure(figsize=(10, 10))
plt.imshow(np.transpose(vutils.make_grid(fake_images, nrow=3,
padding=5, normalize=True, pad_value=1),(1,2,0)))
plt.axis("off")
```

Appendix B: “gancls_model.py”

```
import tensorflow as tf
from Utils import ops



class GAN:
    """
    OPTIONS
    z_dim : Noise dimension 100
```

```
t_dim : Text feature dimension 256
image_size : Image Dimension 64
gf_dim : Number of conv in the first layer generator 64
df_dim : Number of conv in the first layer discriminator 64
gfc_dim : Dimension of gen untis for for fully connected layer 1024
caption_vector_length : Caption Vector Length 2400
batch_size : Batch Size 64
"""

def __init__(self, options):
    self.options = options

    self.g_bn0 = ops.batch_norm(name="g_bn0")
    self.g_bn1 = ops.batch_norm(name="g_bn1")
    self.g_bn2 = ops.batch_norm(name="g_bn2")
    self.g_bn3 = ops.batch_norm(name="g_bn3")

    self.d_bn1 = ops.batch_norm(name="d_bn1")
    self.d_bn2 = ops.batch_norm(name="d_bn2")
    self.d_bn3 = ops.batch_norm(name="d_bn3")
    self.d_bn4 = ops.batch_norm(name="d_bn4")

def build_model(self):
    img_size = self.options["image_size"]
    t_real_image = tf.placeholder(
        "float32",
        [self.options["batch_size"], img_size, img_size, 3],
        name="real_image",
    )
    t_wrong_image = tf.placeholder(
        "float32",
        [self.options["batch_size"], img_size, img_size, 3],
        name="wrong_image",
    )
    t_real_caption = tf.placeholder(
        "float32",
        [self.options["batch_size"],
self.options["caption_vector_length"]],
            name="real_caption_input",
        )
        t_z = tf.placeholder(
            "float32", [self.options["batch_size"], self.options["z_dim"]]
        )

        fake_image = self.generator(t_z, t_real_caption)

        disc_real_image, disc_real_image_logits = self.discriminator(
            t_real_image, t_real_caption
        )
        disc_wrong_image, disc_wrong_image_logits = self.discriminator(
            t_wrong_image, t_real_caption, reuse=True
        )
        disc_fake_image, disc_fake_image_logits = self.discriminator(
            fake_image, t_real_caption, reuse=True
        )

        g_loss = tf.reduce_mean(
            tf.nn.sigmoid_cross_entropy_with_logits(
                disc_fake_image_logits, tf.ones_like(disc_fake_image)
            )
        )

        d_loss1 = tf.reduce_mean(
            tf.nn.sigmoid_cross_entropy_with_logits(
                disc_real_image_logits, tf.ones_like(disc_real_image)
            )
        )
        d_loss2 = tf.reduce_mean(
            tf.nn.sigmoid_cross_entropy_with_logits(
                disc_wrong_image_logits, tf.zeros_like(disc_wrong_image)
            )
        )
        d_loss3 = tf.reduce_mean(
            tf.nn.sigmoid_cross_entropy_with_logits(
            disc_fake_image_logits, tf.zeros_like(disc_fake_image)
        )
    )

    d_loss = d_loss1 + d_loss2 + d_loss3

    t_vars = tf.trainable_variables()
    d_vars = [var for var in t_vars if "d_" in var.name]
    g_vars = [var for var in t_vars if "g_" in var.name]

    input_tensors = {
        "t_real_image": t_real_image,
        "t_wrong_image": t_wrong_image,
        "t_real_caption": t_real_caption,
        "t_z": t_z,
    }

    variables = {"d_vars": d_vars, "g_vars": g_vars}

    loss = {"g_loss": g_loss, "d_loss": d_loss}

    outputs = {"generator": fake_image}

    checks = {
        "d_loss1": d_loss1,
        "d_loss2": d_loss2,
        "d_loss3": d_loss3,
        "disc_real_image_logits": disc_real_image_logits,
        "disc_wrong_image_logits": disc_wrong_image,
        "disc_fake_image_logits": disc_fake_image_logits,
    }

    return input_tensors, variables, loss, outputs, checks

def build_generator(self):
    img_size = self.options["image_size"]
    t_real_caption = tf.placeholder(
            "float32",
            [self.options["batch_size"],
self.options["caption_vector_length"]],
            name="real_caption_input",
        )
        t_z = tf.placeholder(
            "float32", [self.options["batch_size"], self.options["z_dim"]]
        )
        fake_image = self.sampler(t_z, t_real_caption)

        input_tensors = {"t_real_caption": t_real_caption, "t_z": t_z}

        outputs = {"generator": fake_image}

        return input_tensors, outputs

    # Sample Images for a text embedding
    def sampler(self, t_z, t_text_embedding):
        tf.get_variable_scope().reuse_variables()

        s = self.options["image_size"]
        s2, s4, s8, s16 = int(s / 2), int(s / 4), int(s / 8), int(s / 16)

        reduced_text_embedding = ops.lrelu(
            ops.linear(t_text_embedding, self.options["t_dim"],
"g_embedding")
        )
        z_concat = tf.concat(1, [t_z, reduced_text_embedding])
        z_ = ops.linear(z_concat, self.options["gf_dim"] * 8 * s16 * s16,
"g_h0_lin")
        h0 = tf.reshape(z_, [-1, s16, s16, self.options["gf_dim"] * 8])
        h0 = tf.nn.relu(self.g_bn0(h0, train=False))

        h1 = ops.deconv2d(
            h0,
            [self.options["batch_size"], s8, s8, self.options["gf_dim"] * 4],
            name="g_h1",
        )
        h1 = tf.nn.relu(self.g_bn1(h1, train=False))

        h2 = ops.deconv2d(
            h1,
            [self.options["batch_size"], s4, s4, self.options["gf_dim"] * 2],
            name="g_h2",
        )
        h2 = tf.nn.relu(self.g_bn2(h2, train=False))

        h3 = ops.deconv2d(
            h2,
            [self.options["batch_size"], s2, s2, self.options["gf_dim"] * 1],
            name="g_h3",
        )
        h3 = tf.nn.relu(self.g_bn3(h3, train=False))

        h4 = ops.deconv2d(h3, [self.options["batch_size"], s, s, 3],
name="g_h4")

        return tf.tanh(h4) / 2.0 + 0.5

    # GENERATOR IMPLEMENTATION based on :
https://github.com/carpedm20/DCGAN-tensorflow/blob/master/model.p
y
    def generator(self, t_z, t_text_embedding):
        s = self.options["image_size"]
        s2, s4, s8, s16 = int(s / 2), int(s / 4), int(s / 8), int(s / 16)

        reduced_text_embedding = ops.lrelu(
            ops.linear(t_text_embedding, self.options["t_dim"],
"g_embedding")
        )
        z_concat = tf.concat(1, [t_z, reduced_text_embedding])
        z_ = ops.linear(z_concat, self.options["gf_dim"] * 8 * s16 * s16,
"g_h0_lin")
        h0 = tf.reshape(z_, [-1, s16, s16, self.options["gf_dim"] * 8])
        h0 = tf.nn.relu(self.g_bn0(h0))

        h1 = ops.deconv2d(
            h0,
            [self.options["batch_size"], s8, s8, self.options["gf_dim"] * 4],
            name="g_h1",
        )
        h1 = tf.nn.relu(self.g_bn1(h1))

        h2 = ops.deconv2d(
            h1,
            [self.options["batch_size"], s4, s4, self.options["gf_dim"] * 2],
            name="g_h2",
        )
        h2 = tf.nn.relu(self.g_bn2(h2))

        h3 = ops.deconv2d(
            h2,
            [self.options["batch_size"], s2, s2, self.options["gf_dim"] * 1],
            name="g_h3",
        )
        h3 = tf.nn.relu(self.g_bn3(h3))

        h4 = ops.deconv2d(h3, [self.options["batch_size"], s, s, 3],
name="g_h4")

        return tf.tanh(h4) / 2.0 + 0.5

    # DISCRIMINATOR IMPLEMENTATION based on :
https://github.com/carpedm20/DCGAN-tensorflow/blob/master/model.p
y
    def discriminator(self, image, t_text_embedding, reuse=False):
        if reuse:
            tf.get_variable_scope().reuse_variables()

        h0 = ops.lrelu(
            ops.conv2d(image, self.options["df_dim"], name="d_h0_conv")
        )  # 32
        h1 = ops.lrelu(
            self.d_bn1(ops.conv2d(h0, self.options["df_dim"] * 2,
name="d_h1_conv"))
        )  # 16
        h2 = ops.lrelu(
            self.d_bn2(ops.conv2d(h1, self.options["df_dim"] * 4,
name="d_h2_conv"))
        )  # 8
        h3 = ops.lrelu(
            self.d_bn3(ops.conv2d(h2, self.options["df_dim"] * 8,
name="d_h3_conv"))
        )  # 4

        # ADD TEXT EMBEDDING TO THE NETWORK
        reduced_text_embeddings = ops.lrelu(
            ops.linear(t_text_embedding, self.options["t_dim"],
"d_embedding")
        )
        reduced_text_embeddings =
tf.expand_dims(reduced_text_embeddings, 1)
        reduced_text_embeddings =
tf.expand_dims(reduced_text_embeddings, 2)
        tiled_embeddings = tf.tile(
            reduced_text_embeddings, [1, 4, 4, 1], name="tiled_embeddings"
        )

        h3_concat = tf.concat(3, [h3, tiled_embeddings],
name="h3_concat")
        h3_new = ops.lrelu(
            self.d_bn4(
                ops.conv2d(
                    h3_concat,
                    self.options["df_dim"] * 8,
                    1,
                    1,
                    1,
                    1,
                    name="d_h3_conv_new",
                )
            )
        )  # 4

        h4 = ops.linear(
            tf.reshape(h3_new, [self.options["batch_size"], -1]), 1, "d_h3_lin"
        )

        return tf.nn.sigmoid(h4), h4
```

Appendix C: "gancls_thought_vectors.py"

```
import os
from os.path import join, isfile
import re
import numpy as np
import pickle
import argparse
import skipthoughts
import h5py


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "--caption_file",
        type=str,
        default="Data/sample_captions.txt",
        help="caption file",
    )
    parser.add_argument("--data_dir", type=str, default="Data",
help="Data Directory")

    args = parser.parse_args()
    with open(args.caption_file) as f:
        captions = f.read().split("\n")

    captions = [cap for cap in captions if len(cap) > 0]
    print(captions)
    model = skipthoughts.load_model()
    caption_vectors = skipthoughts.encode(model, captions)

    if os.path.isfile(join(args.data_dir, "sample_caption_vectors.hdf5")):
        os.remove(join(args.data_dir, "sample_caption_vectors.hdf5"))
    h = h5py.File(join(args.data_dir, "sample_caption_vectors.hdf5"))
    h.create_dataset("vectors", data=caption_vectors)
    h.close()


if __name__ == "__main__":
    main()
```

Appendix D: "stackgan_models.py"

```
import torch
import torch.nn as nn
import torch.nn.parallel
from miscc.config import cfg
from torch.autograd import Variable


def conv3x3(in_planes, out_planes, stride=1):
    "3x3 convolution with padding"
    return nn.Conv2d(
        in_planes, out_planes, kernel_size=3, stride=stride, padding=1,
bias=False
    )


# Upsale the spatial size by a factor of 2
def upBlock(in_planes, out_planes):
    block = nn.Sequential(
        nn.Upsample(scale_factor=2, mode="nearest"),
        conv3x3(in_planes, out_planes),
        nn.BatchNorm2d(out_planes),
        nn.ReLU(True),
    )
    return block


class ResBlock(nn.Module):
    def __init__(self, channel_num):
        super(ResBlock, self).__init__()
        self.block = nn.Sequential(
            conv3x3(channel_num, channel_num),
            nn.BatchNorm2d(channel_num),
            nn.ReLU(True),
            conv3x3(channel_num, channel_num),
            nn.BatchNorm2d(channel_num),
        )
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        residual = x
        out = self.block(x)
        out += residual
        out = self.relu(out)
        return out


class CA_NET(nn.Module):
    # some code is modified from vae examples
    # (https://github.com/pytorch/examples/blob/master/vae/main.py)
    def __init__(self):
        super(CA_NET, self).__init__()
        self.t_dim = cfg.TEXT.DIMENSION
        self.c_dim = cfg.GAN.CONDITION_DIM
        self.fc = nn.Linear(self.t_dim, self.c_dim * 2, bias=True)
        self.relu = nn.ReLU()
    def encode(self, text_embedding):
        x = self.relu(self.fc(text_embedding))
        mu = x[:, : self.c_dim]
        logvar = x[:, self.c_dim :]
        return mu, logvar

    def reparametrize(self, mu, logvar):
        std = logvar.mul(0.5).exp_()
        if cfg.CUDA:
            eps = torch.cuda.FloatTensor(std.size()).normal_()
        else:
            eps = torch.FloatTensor(std.size()).normal_()
        eps = Variable(eps)
        return eps.mul(std).add_(mu)

    def forward(self, text_embedding):
        mu, logvar = self.encode(text_embedding)
        c_code = self.reparametrize(mu, logvar)
        return c_code, mu, logvar


class D_GET_LOGITS(nn.Module):
    def __init__(self, ndf, nef, bcondition=True):
        super(D_GET_LOGITS, self).__init__()
        self.df_dim = ndf
        self.ef_dim = nef
        self.bcondition = bcondition
        if bcondition:
            self.outlogits = nn.Sequential(
                conv3x3(ndf * 8 + nef, ndf * 8),
                nn.BatchNorm2d(ndf * 8),
                nn.LeakyReLU(0.2, inplace=True),
                nn.Conv2d(ndf * 8, 1, kernel_size=4, stride=4),
                nn.Sigmoid(),
            )
        else:
```

```
        self.outlogits = nn.Sequential(
            nn.Conv2d(ndf * 8, 1, kernel_size=4, stride=4), nn.Sigmoid()
        )

    def forward(self, h_code, c_code=None):
        # conditioning output
        if self.bcondition and c_code is not None:
            c_code = c_code.view(-1, self.ef_dim, 1, 1)
            c_code = c_code.repeat(1, 1, 4, 4)
            # state size (ngf+egf) x 4 x 4
            h_c_code = torch.cat((h_code, c_code), 1)
        else:
            h_c_code = h_code

        output = self.outlogits(h_c_code)
        return output.view(-1)


# ############# Networks for stageI GAN #############
class STAGE1_G(nn.Module):
    def __init__(self):
        super(STAGE1_G, self).__init__()
        self.gf_dim = cfg.GAN.GF_DIM * 8
        self.ef_dim = cfg.GAN.CONDITION_DIM
        self.z_dim = cfg.Z_DIM
        self.define_module()

    def define_module(self):
        ninput = self.z_dim + self.ef_dim
        ngf = self.gf_dim
        # TEXT.DIMENSION -> GAN.CONDITION_DIM
        self.ca_net = CA_NET()

        # -> ngf x 4 x 4
        self.fc = nn.Sequential(
            nn.Linear(ninput, ngf * 4 * 4, bias=False),
            nn.BatchNorm1d(ngf * 4 * 4),
            nn.ReLU(True),
        )

        # ngf x 4 x 4 -> ngf/2 x 8 x 8
        self.upsample1 = upBlock(ngf, ngf // 2)
        # -> ngf/4 x 16 x 16
        self.upsample2 = upBlock(ngf // 2, ngf // 4)
        # -> ngf/8 x 32 x 32
        self.upsample3 = upBlock(ngf // 4, ngf // 8)
        # -> ngf/16 x 64 x 64
        self.upsample4 = upBlock(ngf // 8, ngf // 16)
        # -> 3 x 64 x 64
        self.img = nn.Sequential(conv3x3(ngf // 16, 3), nn.Tanh())

    def forward(self, text_embedding, noise):
        c_code, mu, logvar = self.ca_net(text_embedding)
        z_c_code = torch.cat((noise, c_code), 1)
        h_code = self.fc(z_c_code)

        h_code = h_code.view(-1, self.gf_dim, 4, 4)
        h_code = self.upsample1(h_code)
        h_code = self.upsample2(h_code)
        h_code = self.upsample3(h_code)
        h_code = self.upsample4(h_code)
        # state size 3 x 64 x 64
        fake_img = self.img(h_code)
        return None, fake_img, mu, logvar


class STAGE1_D(nn.Module):
    def __init__(self):
        super(STAGE1_D, self).__init__()
        self.df_dim = cfg.GAN.DF_DIM
        self.ef_dim = cfg.GAN.CONDITION_DIM
        self.define_module()

    def define_module(self):
        ndf, nef = self.df_dim, self.ef_dim
        self.encode_img = nn.Sequential(
            nn.Conv2d(3, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf) x 32 x 32
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size (ndf*2) x 16 x 16
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size (ndf*4) x 8 x 8
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            # state size (ndf * 8) x 4 x 4)
            nn.LeakyReLU(0.2, inplace=True),
        )

        self.get_cond_logits = D_GET_LOGITS(ndf, nef)
        self.get_uncond_logits = None

    def forward(self, image):
        img_embedding = self.encode_img(image)

        return img_embedding


# ############# Networks for stageII GAN #############
class STAGE2_G(nn.Module):
    def __init__(self, STAGE1_G):
        super(STAGE2_G, self).__init__()
        self.gf_dim = cfg.GAN.GF_DIM
        self.ef_dim = cfg.GAN.CONDITION_DIM
        self.z_dim = cfg.Z_DIM
        self.STAGE1_G = STAGE1_G
        # fix parameters of stageI GAN
        for param in self.STAGE1_G.parameters():
            param.requires_grad = False
        self.define_module()

    def _make_layer(self, block, channel_num):
        layers = [block(channel_num) for _ in range(cfg.GAN.R_NUM)]
        return nn.Sequential(*layers)

    def define_module(self):
        ngf = self.gf_dim
        # TEXT.DIMENSION -> GAN.CONDITION_DIM
        self.ca_net = CA_NET()
        # --> 4ngf x 16 x 16
        self.encoder = nn.Sequential(
            conv3x3(3, ngf),
            nn.ReLU(True),
            nn.Conv2d(ngf, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            nn.Conv2d(ngf * 2, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
        )
        self.hr_joint = nn.Sequential(
            conv3x3(self.ef_dim + ngf * 4, ngf * 4),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
        )
        self.residual = self._make_layer(ResBlock, ngf * 4)
        # --> 2ngf x 32 x 32
        self.upsample1 = upBlock(ngf * 4, ngf * 2)
        # --> ngf x 64 x 64
        self.upsample2 = upBlock(ngf * 2, ngf)
        # --> ngf // 2 x 128 x 128
        self.upsample3 = upBlock(ngf, ngf // 2)
        # --> ngf // 4 x 256 x 256
        self.upsample4 = upBlock(ngf // 2, ngf // 4)
        # --> 3 x 256 x 256
        self.img = nn.Sequential(conv3x3(ngf // 4, 3), nn.Tanh())

    def forward(self, text_embedding, noise):
        _, stage1_img, _, _ = self.STAGE1_G(text_embedding, noise)
        stage1_img = stage1_img.detach()
        encoded_img = self.encoder(stage1_img)

        c_code, mu, logvar = self.ca_net(text_embedding)
        c_code = c_code.view(-1, self.ef_dim, 1, 1)
        c_code = c_code.repeat(1, 1, 16, 16)
        i_c_code = torch.cat([encoded_img, c_code], 1)
        h_code = self.hr_joint(i_c_code)
        h_code = self.residual(h_code)

        h_code = self.upsample1(h_code)
        h_code = self.upsample2(h_code)
        h_code = self.upsample3(h_code)
        h_code = self.upsample4(h_code)

        fake_img = self.img(h_code)
        return stage1_img, fake_img, mu, logvar


class STAGE2_D(nn.Module):
    def __init__(self):
        super(STAGE2_D, self).__init__()
        self.df_dim = cfg.GAN.DF_DIM
        self.ef_dim = cfg.GAN.CONDITION_DIM
        self.define_module()

    def define_module(self):
        ndf, nef = self.df_dim, self.ef_dim
        self.encode_img = nn.Sequential(
            nn.Conv2d(3, ndf, 4, 2, 1, bias=False),  # 128 * 128 * ndf
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),  # 64 * 64 * ndf * 2
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),  # 32 * 32 * ndf * 4
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),  # 16 * 16 * ndf * 8
            nn.Conv2d(ndf * 8, ndf * 16, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 16),
            nn.LeakyReLU(0.2, inplace=True),  # 8 * 8 * ndf * 16
            nn.Conv2d(ndf * 16, ndf * 32, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 32),
            nn.LeakyReLU(0.2, inplace=True),  # 4 * 4 * ndf * 32
            conv3x3(ndf * 32, ndf * 16),
            nn.BatchNorm2d(ndf * 16),
            nn.LeakyReLU(0.2, inplace=True),  # 4 * 4 * ndf * 16
            conv3x3(ndf * 16, ndf * 8),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),  # 4 * 4 * ndf * 8
        )

        self.get_cond_logits = D_GET_LOGITS(ndf, nef, bcondition=True)
        self.get_uncond_logits = D_GET_LOGITS(ndf, nef,
bcondition=False)

    def forward(self, image):
        img_embedding = self.encode_img(image)

        return img_embedding
```

Appendix E: “stackgan_train.py”

```
from __future__ import print_function
from six.moves import range
from PIL import Image
```

```
import torch.backends.cudnn as cudnn
import torch
import torch.nn as nn
from torch.autograd import Variable
import torch.optim as optim
import os
import time

import numpy as np
import torchfile

from miscc.config import cfg
from miscc.utils import mkdir_p
from miscc.utils import weights_init
from miscc.utils import save_img_results, save_model
from miscc.utils import KL_loss
from miscc.utils import compute_discriminator_loss,
compute_generator_loss

from tensorboard import summary
from tensorboard import FileWriter


class GANTrainer(object):
    def __init__(self, output_dir):
        if cfg.TRAIN.FLAG:
            self.model_dir = os.path.join(output_dir, "Model")
            self.image_dir = os.path.join(output_dir, "Image")
            self.log_dir = os.path.join(output_dir, "Log")
            mkdir_p(self.model_dir)
            mkdir_p(self.image_dir)
            mkdir_p(self.log_dir)
            self.summary_writer = FileWriter(self.log_dir)

        self.max_epoch = cfg.TRAIN.MAX_EPOCH
        self.snapshot_interval = cfg.TRAIN.SNAPSHOT_INTERVAL
        s_gpus = cfg.GPU_ID.split(",")
        self.gpus = [int(ix) for ix in s_gpus]
        self.num_gpus = len(self.gpus)
        self.batch_size = cfg.TRAIN.BATCH_SIZE * self.num_gpus
        torch.cuda.set_device(self.gpus[0])
        cudnn.benchmark = True

    # ############# For training stageI GAN #############
    def load_network_stageI(self):
        from model import STAGE1_G, STAGE1_D

        netG = STAGE1_G()
        netG.apply(weights_init)
        print(netG)
        netD = STAGE1_D()
        netD.apply(weights_init)
        print(netD)

        if cfg.NET_G != "":
            state_dict = torch.load(
                cfg.NET_G, map_location=lambda storage, loc: storage
            )
            netG.load_state_dict(state_dict)
            print("Load from: ", cfg.NET_G)
        if cfg.NET_D != "":
            state_dict = torch.load(
                cfg.NET_D, map_location=lambda storage, loc: storage
            )
            netD.load_state_dict(state_dict)
            print("Load from: ", cfg.NET_D)
        if cfg.CUDA:
            netG.cuda()
            netD.cuda()
        return netG, netD

    # ############# For training stageII GAN  #############
    def load_network_stageII(self):
```

```
from model import STAGE1_G, STAGE2_G, STAGE2_D

Stage1_G = STAGE1_G()
netG = STAGE2_G(Stage1_G)
netG.apply(weights_init)
print(netG)
if cfg.NET_G != "":
   state_dict = torch.load(
      cfg.NET_G, map_location=lambda storage, loc: storage
   )
   netG.load_state_dict(state_dict)
   print("Load from: ", cfg.NET_G)
elif cfg.STAGE1_G != "":
   state_dict = torch.load(
      cfg.STAGE1_G, map_location=lambda storage, loc: storage
   )
   netG.STAGE1_G.load_state_dict(state_dict)
   print("Load from: ", cfg.STAGE1_G)
else:
   print("Please give the Stage1_G path")
   return

netD = STAGE2_D()
netD.apply(weights_init)
if cfg.NET_D != "":
   state_dict = torch.load(
      cfg.NET_D, map_location=lambda storage, loc: storage
   )
   netD.load_state_dict(state_dict)
   print("Load from: ", cfg.NET_D)
print(netD)

if cfg.CUDA:
   netG.cuda()
   netD.cuda()
return netG, netD
```

```
def train(self, data_loader, stage=1):
    if stage == 1:
        netG, netD = self.load_network_stageI()
    else:
        netG, netD = self.load_network_stageII()

    nz = cfg.Z_DIM
    batch_size = self.batch_size
    noise = Variable(torch.FloatTensor(batch_size, nz))
    fixed_noise = Variable(
        torch.FloatTensor(batch_size, nz).normal_(0, 1), volatile=True
    )
    real_labels = Variable(torch.FloatTensor(batch_size).fill_(1))
    fake_labels = Variable(torch.FloatTensor(batch_size).fill_(0))
    if cfg.CUDA:
        noise, fixed_noise = noise.cuda(), fixed_noise.cuda()
        real_labels, fake_labels = real_labels.cuda(), fake_labels.cuda()

    generator_lr = cfg.TRAIN.GENERATOR_LR
    discriminator_lr = cfg.TRAIN.DISCRIMINATOR_LR
    lr_decay_step = cfg.TRAIN.LR_DECAY_EPOCH
    optimizerD = optim.Adam(
        netD.parameters(), lr=cfg.TRAIN.DISCRIMINATOR_LR,
betas=(0.5, 0.999)
    )
    netG_para = [p for p in netG.parameters() if p.requires_grad]
    optimizerG = optim.Adam(
        netG_para, lr=cfg.TRAIN.GENERATOR_LR, betas=(0.5, 0.999)
    )
    count = 0
    for epoch in range(self.max_epoch):
        start_t = time.time()
        if epoch % lr_decay_step == 0 and epoch > 0:
            generator_lr *= 0.5
            for param_group in optimizerG.param_groups:
                param_group["lr"] = generator_lr
            discriminator_lr *= 0.5
            for param_group in optimizerD.param_groups:
               param_group["lr"] = discriminator_lr

         for i, data in enumerate(data_loader, 0):

############################################################
            # (1) Prepare training data

############################################################
            real_img_cpu, txt_embedding = data
            real_imgs = Variable(real_img_cpu)
            txt_embedding = Variable(txt_embedding)
            if cfg.CUDA:
               real_imgs = real_imgs.cuda()
               txt_embedding = txt_embedding.cuda()


############################################################
            # (2) Generate fake images

############################################################
            noise.data.normal_(0, 1)
            inputs = (txt_embedding, noise)
            _, fake_imgs, mu, logvar = nn.parallel.data_parallel(
               netG, inputs, self.gpus
            )

            ##############################
            # (3) Update D network
            #############################
            netD.zero_grad()
            errD, errD_real, errD_wrong, errD_fake =
compute_discriminator_loss(
               netD, real_imgs, fake_imgs, real_labels, fake_labels, mu,
self.gpus
            )
            errD.backward()
            optimizerD.step()
            ################################
            # (2) Update G network
            ###############################
            netG.zero_grad()
            errG = compute_generator_loss(
               netD, fake_imgs, real_labels, mu, self.gpus
            )
            kl_loss = KL_loss(mu, logvar)
            errG_total = errG + kl_loss * cfg.TRAIN.COEFF.KL
            errG_total.backward()
            optimizerG.step()

            count = count + 1
            if i % 100 == 0:
               summary_D = summary.scalar("D_loss", errD.data[0])
               summary_D_r = summary.scalar("D_loss_real", errD_real)
               summary_D_w = summary.scalar("D_loss_wrong",
errD_wrong)
               summary_D_f = summary.scalar("D_loss_fake", errD_fake)
               summary_G = summary.scalar("G_loss", errG.data[0])
               summary_KL = summary.scalar("KL_loss", kl_loss.data[0])

               self.summary_writer.add_summary(summary_D, count)
               self.summary_writer.add_summary(summary_D_r, count)
               self.summary_writer.add_summary(summary_D_w, count)
               self.summary_writer.add_summary(summary_D_f, count)
               self.summary_writer.add_summary(summary_G, count)
               self.summary_writer.add_summary(summary_KL, count)

               # save the image result for each epoch
               inputs = (txt_embedding, fixed_noise)
               lr_fake, fake, _, _ = nn.parallel.data_parallel(
                  netG, inputs, self.gpus
               )
               save_img_results(real_img_cpu, fake, epoch,
self.image_dir)
```

```
                if lr_fake is not None:
                    save_img_results(None, lr_fake, epoch, self.image_dir)
            end_t = time.time()
            print(
                """[%d/%d][%d/%d] Loss_D: %.4f Loss_G: %.4f Loss_KL:
%.4f
                 Loss_real: %.4f Loss_wrong:%.4f Loss_fake %.4f
                 Total Time: %.2fsec
              """
                % (
                    epoch,
                    self.max_epoch,
                    i,
                    len(data_loader),
                    errD.data[0],
                    errG.data[0],
                    kl_loss.data[0],
                    errD_real,
                    errD_wrong,
                    errD_fake,
                    (end_t - start_t),
                )
            )
            if epoch % self.snapshot_interval == 0:
                save_model(netG, netD, epoch, self.model_dir)
        #
        save_model(netG, netD, self.max_epoch, self.model_dir)
        #
        self.summary_writer.close()

    def sample(self, datapath, stage=1):
        if stage == 1:
            netG, _ = self.load_network_stageI()
        else:
            netG, _ = self.load_network_stageII()
        netG.eval()
    # Load text embeddings generated from the encoder
    t_file = torchfile.load(datapath)
    captions_list = t_file.raw_txt
    embeddings = np.concatenate(t_file.fea_txt, axis=0)
    num_embeddings = len(captions_list)
    print("Successfully load sentences from: ", datapath)
    print("Total number of sentences:", num_embeddings)
    print("num_embeddings:", num_embeddings, embeddings.shape)
    # path to save generated samples
    save_dir = cfg.NET_G[: cfg.NET_G.find(".pth")]
    mkdir_p(save_dir)

    batch_size = np.minimum(num_embeddings, self.batch_size)
    nz = cfg.Z_DIM
    noise = Variable(torch.FloatTensor(batch_size, nz))
    if cfg.CUDA:
        noise = noise.cuda()
    count = 0
    while count < num_embeddings and count <= 3000:
        iend = count + batch_size
        if iend > num_embeddings:
            iend = num_embeddings
            count = num_embeddings - batch_size
        embeddings_batch = embeddings[count:iend]
        # captions_batch = captions_list[count:iend]
        txt_embedding = Variable(torch.FloatTensor(embeddings_batch))
        if cfg.CUDA:
            txt_embedding = txt_embedding.cuda()


#######################################################
        # (2) Generate fake images

######################################################
        noise.data.normal_(0, 1)
        inputs = (txt_embedding, noise)
        _, fake_imgs, mu, logvar = nn.parallel.data_parallel(
            netG, inputs, self.gpus
        )
        for i in range(batch_size):
            save_name = "%s/%d.png" % (save_dir, count + i)
            im = fake_imgs[i].data.cpu().numpy()
            im = (im + 1.0) * 127.5
            im = im.astype(np.uint8)
            # print('im', im.shape)
            im = np.transpose(im, (1, 2, 0))
            # print('im', im.shape)
            im = Image.fromarray(im)
            im.save(save_name)
        count += batch_size
```