Erkin Özlan - 21901705
Gökcan Kahraman - 21702271
Müşteba Anıl Önder-21604090
Group 12

EEE 443 Neural Networks Project Report

Abstract

Neural networks have become very popular in recent years, especially with the development of computer technology. With the recent developments, it created progress in many tasks like autonomous driving, emotion recognition, language translation and many others. For our task, neural networks were suitable, compared to the other machine learning methods. The aim of our project was to prepare a project using our neural network knowledge with the training and test data given to us. We could only use one dataset in the project. With the free use of the libraries, what we were asked to do in our project was to obtain images using texts. To achieve this goal, we implemented a GAN architecture where the generator generates the desired images from the given captions and the discriminator compares the original images and the desired images. We used pretrained glove models for text embedding which is inputted to the generator and the discriminator both. The feature extraction for the images is done inside the discriminator itself, since it is essentially a CNN architecture.

Introduction

Our goal in the project is to write a text2image algorithm and create a new image according to the desired text. We implemented text embedding by creating an embedding matrix according to our own dictionary by using the pre-trained ``glove`` vectors we received as text, downloaded from kaggle with the name ‘glove.6B.200d.txt’[1]. Essentially, our methodology was to obtain the desired image with a Stacked GAN model given the texts. Stacked GAN’s concatenate two GAN algorithms called as ‘stage 1’ and ‘stage 2’, where in the first stage, low resolution images are obtained with the stage 1 generator, and by feeding the generated images to stage 2, the resolution is increased and detailed real-looking images are obtained[2]. However, the process of training and coding both stages was cumbersome, hence we’ve implemented the stage 1 only. As a note, this is our only model that we managed to get some results. Different types of GAN’s like DcGAN, or other methods such as stable diffusion are also used for text2image applications which are based on deep learning [3].

The importance of this project is to create an artificial algorithm that thinks like humans and designs images that people want. We use visuals in many fields from marketing to technology and one of the cornerstones of these marketing strategies is texts. It will be much more effective to design a visual design using a text which will attract users and what they want to see according to their data. In addition, visual arts gain a different dimension with the use of algorithms in this way. Our algorithm, GANs, are used in this aspect. A brief overview of the whole process will be given below and will be detailed in the later sections.

We've downloaded the text data from the given urls with a function using the operating systems library to communicate with our device in the beginning. Before giving input to the generator function, we need textual descriptions for getting our text databases to the generator side. Then, we need to convert those textual descriptions to text embedding. Text embedding simply means a vector of a real number which we want to convert a sentence into a vector of real number of fixed length. If we want to evaluate the texts clearly on GAN architecture, we need to make this operation. Then, for the final step for generator input, we need to concatenate embedding with a random noise vector.

After the generator input is obtained, we need to obtain the generator output that would be inputted to the discriminator. Generally, the aim of the generator is generating fake images from random noise. But, for our function, we made our "generator" based on the text embeddings. Because we wanted to generate text2image, we needed to make a relative image of texts.. We used upsampling and convolutional layers for building the generator.

Finally, to complete our GAN stage, we need to use a discriminator. Before explaining how a discriminator works and how it trains our images, we need to explain what discriminators' inputs and outputs are. On the input side, we have two different types. We have images which are taken from either the dataset or from the generator and text which is related to images. The output of the discriminator aims to give zeros and ones because the discriminator is a training function of the generator. Its method of classifying outputs is that if our output is zero then our image is fake. Also, when our output is one, it means that our image is real. Since we need to look over around 80.000 images on datasets, we diminished the resolution of images to process the images more easily to 64*64*3 rgb format. During the coding of discriminator, we mainly used CNN classifier architecture which fits the problem well since we are classifying images.

All these steps are expressed in a more detailed fashion in the later section, but the methodology and planning process is shared here. Since we implemented stage 1 only, we expected the image quality would be low, for Stacked GAN architectures, images are turned to high quality at stage 2. However, we expected to receive related images with the captions.

Methods

In the methods section you see here, we will explain step by step the basic algorithms we implemented for our neural network project. For a better explanation of the methods section, we divided it into 3 main subsections, namely data processing, text embedding and the implemented gan model.

Data Processing

Erkin Özlan - 21901705
Gökcan Kahraman - 21702271
Müşteba Anıl Önder-21604090
Group 12

In this part of our report, we will explain how we make two different data groups in the database suitable for the project. Feature extraction will also be explained in this section.

train_url and test_url: As the name suggests, it contains the visual database of our project. There are 82783 different images in this database, which is converted to a numpy array later on in the algorithms. Although the sizes of these images are different from each other, there are various images such as photographs taken in different houses, sports competitions, nature images, animals and many more.

train_cap and test_caps: This subheading shows how many different captions the images are in the "url" arrays. This data, stored as an integer, is important for word_count. This is because word_count generates the string representation of the description elements.

train_imid and test_imid: This subheading shows the indexed URLs within the images. The presence of each indexed image in the train_url array is denoted by (x - 1). The symbol "x" here is the number of index URLs of the images. Because the minimum index in test_imid is zero, 1 is subtracted. Also, x is given for test captions similarly. The imid array also acts as a bridge between the captions and their URLs. This is why imid is used a lot in index algorithms.

word_count: As the name suggests, it shows how many different words are in the text database. There are 1004 different words in our text database. There are also four different words to indicate special cases:

“x_END_” : Used to finish the captions we have written.
“x_NULL_” : Used to extend the text spectrum with zeros for captions.
“x_START_” : Used to start the captions we have written.
“x_UNK_” : Used for unknown words such as special cases or just not included in a text database.

With these special cases and 1004 different words, the database required for our text2image algorithm is created. For this algorithm, there should be as many text databases as there are image databases. In this way, we train the images we will create in a more beautiful and precise way.

First of all, we need to get our image's appropriate extension. Therefore, we downloaded all of the pictures with the extension of “.jpg”. The reason why we did not use them based on their URL is since those URLs were not working because all of the URLs were built a while ago. To check which URLs are working correctly we used the response status algorithm and based on the value we got, we knew whether or not we were able to

download the images. Thanks to this method, we were able to download both the train images and test images. However, we thought about one of the problems that can be encountered with the database in a neural network: Our algorithm might not train itself well because some of the images were corrupted. Another significant issue with the test data was that two photos were unavaliable. Decoding the images and determining if they were genuinely JPEG images led to the deprecation of these images. We were able to recover around 85% of the part of the dataset by fixing these issues. Additionally, we checked a similar scenario for test images, and the outcome was around 80%.

We pulled the images from the database with a high success rate, but the sizes of the images were different from each other, which creates another problem. The reason why this concept is problematic is that when training our algorithm, it would look at different parts of the images when inferring different resolutions, which would be bad for training. So we reduced the resolution of our images to 64x64, both for better training and so that the running time within epochs is not too long. One other thing to mention is that like image sizes, caption lengths are also different from each other. We fixed the caption length to the maximum caption size which we found as 17, and zero pad the empty parts at the end to obtain the fixed length captions that would be inputted to the model.

Using the train_imid and test_imid data, we ‘ve combined the images and their several related captions together in a tensorflow object. We divided the training data into two parts with 15% separation to validation data, as this is the amount required from us. For this purpose, we randomly shuffled the training data and chose the first 15% of it. After obtaining the training and validation data, the pre-processing part was done.

Note: The pre-trained feature vectors for the images for both training and testing data are not used which halves the grades of the project. The feature extraction for comparing the images is done inside the discriminator, since it already includes convolutional layers and down samples the image to classify whether it is fake or real.

Text Embedding

In the data processing part we have prepared to train the GAN algorithm, we need text embedding to properly run the generator and discriminator materials in the GAN. We need to embed our input texts we have obtained with 1004 different word types that we have drawn from word_count. The reason the word embedding concept is so important is to make sure that groups of words that are semantically similar are represented with similar embedding vectors. This is critical in NLP applications. We chose to use the ‘glove’ vectors for this purpose, however the word2vec method is also used a lot. Glove is a method, in 2014 Pennington et al developed at Stanford, that measures the frequency of co-occurrence of bases of words and vectors. Firstly, the glove method creates a matrix which includes rows

for words and columns for context. Also, the element values of X vectors represent the frequency which words appear with different words in contexts. This matrix is factorized to a lower-dimensional form by reducing reconstruction loss since each row represents words as already mentioned above. The outcome is a learning model that could lead to a more accurate word embedding overall [4]. The glove text is downloaded from kaggle, as referenced in the introduction part.

The embedding matrix has size (1004,200) since our dictionary size is 1004 and we used 200d text embedding vectors. This matrix is then multiplied with our one-hotted captions and each word is represented with their corresponding 200 dimensional vector. Then, each sample caption became (17,200) of size. Before giving it to the input of our generator model, we flattened it to the size (1,3400) and our captions were ready to go.

Implemented Model: GAN

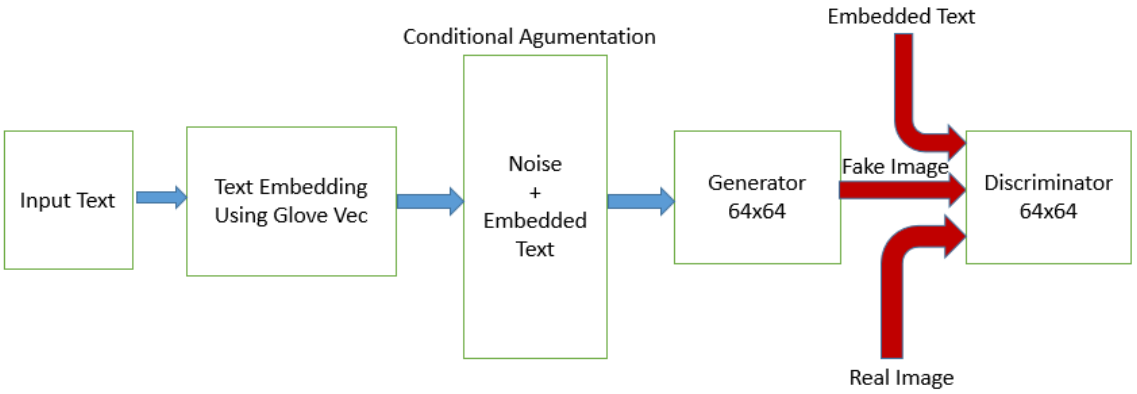

Figure 1: Gan Model Diagram

Our diagram of the whole GAN architecture can be seen above. It is essentially the stage 1 architecture of Stacked GAN. The architecture is done with similar ideas we've learned from the referenced source, which contains the Stacked GAN architecture we were inspired from[5]. We used a batch gradient descent algorithm for training the models, where the details will be shared below.

If we had run the GAN model directly with text embeddings, we would not have gotten the image we wanted. This is because the GAN concept requires noise to work. In fact, GAN algorithms that just do image generation don't have text embedding data but input just the noise to the generator. The process of concatenating the embedding text with the random noise is done with a conditioning augmentation network[6]. Also, we used a further text compressor later on, too.

GAN models consist of generators and discriminators. However, the stages made in GAN models differ according to the purpose of use. First, we need to understand how the generator works so that we can understand what kind of input it enters into the discriminator. By integrating input from the discriminator, the generator component of a GAN learns to produce fictitious data. It gains the ability to get the discriminator to label its output as 'real'. Our generator function takes the 3400 length embedding text and the noise as input from the conditional augmentation stage. Inside the generator, these two inputs are concatenated and fake images which have size 64x64 are generated. In this way, inputs that will be given to the discriminator are created [7].

The discriminator function in the GAN is used as a classifier. The fake image generated by the generator, the embedding text and the real image are inputted to the discriminator. According to these inputs, the discriminator makes a decision on whether the input image given its caption is real or fake. If the result is 0, our image is fake. If the result is 1, our image is real. In this way, we train the generator and as a result, we get our own images by working with other images with the GAN algorithm[8]. In our implementation, the discriminator is trained on 3 different text,image pairs. The desired target values for each are shared in the table below.

Input	Desired Output
real image, real caption	1
real image, fake caption	0
fake image, real caption	0

Here, fake images are created by the generator. Fake captions are created by us to make the discriminator learn to classify the output as true only when the image and caption matches.

To combine both models, an adversarial model is created which takes both models as inputs and used for training the generator network. Discriminator is trained separately, as mentioned above. Details of all the models with their losses, layers, optimizers and outputs will be shared below.

Discriminator model details: The discriminator network uses convolutional layers with filter size starting from 32 ranging to 512 with 4x4 kernels, stride of 2 and with zero padding

for extracting features from the inputted image. Batch normalization is used to stabilize and fasten the training process. 'Leaky Relu' activations are used at the end of convolutional layers. The input text is merged with the extracted image and after a fully connected layer with a single output (0 or 1), the classifier is created. The output is softmax, hence we assign probabilities to the images on whether they are real or fake. This is essentially a CNN classifier. The full model can be seen in the Appendix.
Loss function: Binary cross entropy
Optimizer: Adam, beta 1 = 0.5, beta 2 = 0.99

Generator model details: The discriminator model needs to generate 64*64*3 RGB images from the embedded text input, hence 2x2 upsampling and convolutional layers with 3x3 filters are present. Stride is 1, to avoid decreasing the size. Activation functions of upsample-convolution blocks are 'Relu' in hidden layers and hyperbolic tangent is used at the image output. Full model can be seen in the Appendix.
Loss function: Mean Squared Error
Optimizer: Adam, beta 1 = 0.5, beta 2 = 0.99

Adversarial model details: Adversarial model is just to combine the discriminator and generator, and is used to train the generator. Full model can be seen in the Appendix.
Loss function: KL divergence

Batch Gradient Descent: We used batch gradient descent with a batch size of 128. Learning rate is chosen as 0.001 for both the discriminator and the generator, with epoch size of 1000. A higher learning rate causes the algorithm to diverge. Batch size could be decreased, but training takes too long, hence we chose it this way.

Training time: Training took approximately 8-9 hours, where we stopped afterwards. GAN's architectures are really hard to train since 2 models are trained simultaneously. Our data is really large too. Also, the model is a big network which has parameters around 13 million.

Results

In this part of our report, we will share with you the results of the images obtained from the test data using our text2image GAN implementation. Also, we will give the loss values obtained for training and validation stages and the interpretation. Losses are given at 1000 (full) epochs.

discriminator loss for training set: 0.5750525478106283
discriminator validation loss: 0.8018298745155334

adversarial model validation loss: 0.001000416581518948

Here, the discriminator loss is binary cross entropy loss and the adversarial model's loss (indicating whether the generator was able to fool the discriminator, hence measuring performance of the generator in some sense) is the KL divergence loss. The interpretation of loss values for GAN architectures can change depending on the architecture and training process since both networks are trained together and loss depends not on the quality of the images generated given the caption, but whether one overcomes the other. However, the training error for the discriminator to be around 50% can be a good sign, since the generator is then able to fool the discriminator completely, it is just guessing which may mean that the training is over. Metrics like FID or IS could be used to capture the image qualities, but won't be effective on whether the image fits given the caption.

The extracted images can be a better tool to see the working quality of the model. Hence, some correct and false captions will be shared next with their given captions.

Generated Images with Test Captions

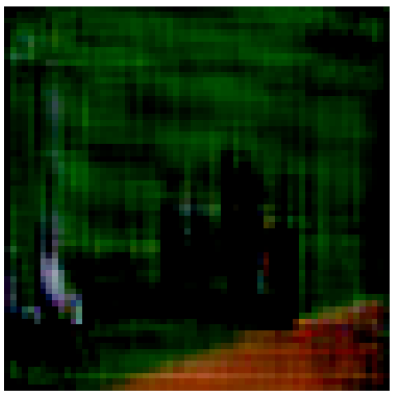

Figure 2 and 3: Generated image and actual image

First of all, in this image we have given, we see someone at a baseball game. The image you see on the right side is the image we obtained from the database, while the image on the left side is the image extracted by our algorithm. The text group we obtained while extracting the image on the left is: ["a man standing on a baseball field and playing baseball"]. As you can see from our text translation, we have provided the correct text for our image and we understand that the images are related to each other.

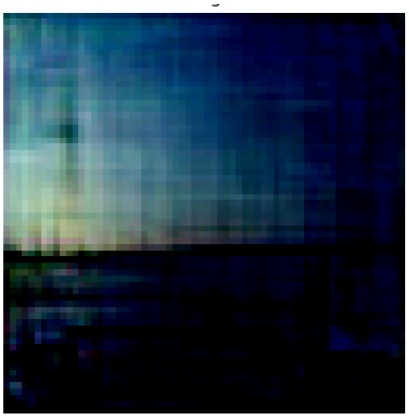

Figure 4 and 5: Generated image and actual image

In this part you see in our picture, just like in the previous example, the image you see on the left is the image made by our algorithm, while the image on the right is the image obtained from our data group. When we run this image, the text group we get is: ["a person standing on a sea around a X_UNK_"]. As you can see again from our text translation, we have provided the correct text for our image and we understand that the images are related to each other.

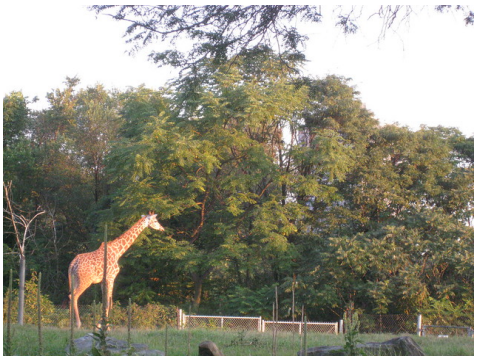

Figure 6 and 7: Generated image and actual image

In this last part, where we show the correct examples, again like the previous two examples, the image on the left is the one generated by our algorithm. The image on the right is again the image we obtained from our database. The text extracted as a text group for this image is: ["a giraffe standing on grass and eating leaves"]. As you can see again from our text translation, we have provided the correct text for our image and we understand that the images are related to each other.

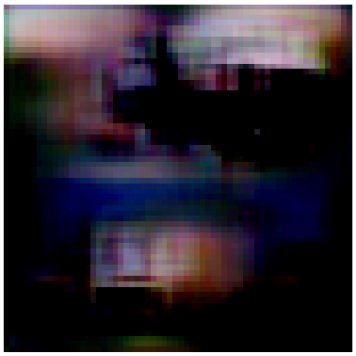

Figure 8 and 9: Generated image and actual image

Here is an example of a failed attempt. Also, the image doesn't look like it captures some other caption, the shapes don't make sense. The input caption for the image is: ["stove inside a X_UNK_ kitchen "]. As there are nice images generated, there are also bad ones.

Discussion

In our research, we read about many types of GANs and other text2image models like stable diffusion. After deciding on a GAN model we thought about which one to create. We've found that many people use the StackGAN concept more for text2image concepts. Because in the past Conditional GAN models were used for text2image files. However, after creating the stage 1 StackGan model (where the training lasted really long), we received many errors while trying to implement the stage 2 model. Hence we mainly focused on improving the stage 1 model.After we found the GAN model, we thought about what kind of concept we should have for better creation of embedded weight matrices. In these studies that we have carried out with the GAN model, we think that the Glove concept is ahead of other concepts. Thanks to the Glove algorithm, we have both global statistics and local statistics and we create a text embedding according to them. Thanks to this, GAN, which also works on this text model, worked with better quality. In this way, even if we used only one stage, we assigned a good input in conditional augmentation because we had better text embedding. We received relatively good results with this embedding type.

We aimed to handle the images and text in the database in a cleaner way because this way both our text embedding and our GAN algorithm would work better. We have completed our project in a good way by processing the remaining correct data through TensorFlow Keras library and the data that is of poor quality did not negatively affect our work. The images we obtained are not as good as the images obtained in modern times, but due to both computer performance and limited time, we tried to get the desired data in the project by reducing the image quality. This way, we managed to get images that do make sense given the caption.

Erkin Özlan - 21901705
Gökcan Kahraman - 21702271
Müşteba Anıl Önder-21604090
Group 12

References

[1]"glove.6B.200d.txt, [Online].https://www.kaggle.com/datasets/incorpes/glove6b200d (Accessed: Jan.10, 2022)

[2] "StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks", [Online] arXiv:1612.03242v2 **[cs.CV]**, (Accessed: Jan.10, 2022) ttps://doi.org/10.48550/arXiv.1612.03242

[3]"Text to image model", [Online].https://en.wikipedia.org/wiki/Text-to-image_model, (Accessed: Jan.10, 2022)

[4] "Introduction to Word Embeddings", [Online], https://medium.com/analytics-vidhya/introduction-to-word-embeddings-c2ba135dce2f, (Accessed: Jan.10, 2022)

[5]"StackGAN | Text to Image Generation with Stacked Generative Adversarial Networks", [Online].video, https://www.youtube.com/watch?v=jXpUo0r34nk, (Accessed: Jan.10, 2022)

[6] "StackGAN Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks",[Online],stackgan.pdf (rice.edu), (Accessed: Jan.10, 2022)

[7]"The Generator", [Online] https://developers.google.com/machine-learning/gan/generator?hl=en, (Accessed: Jan.10, 2022)

[8] "The Discriminator" , [Online], https://developers.google.com/machine-learning/gan/discriminator?hl=en, (Accessed: Jan.10, 2022)

Erkin Özlan - 21901705
Gökcan Kahraman - 21702271
Müşteba Anıl Önder-21604090
Group 12

Appendices

Discriminator model

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_83 (InputLayer)        [(None, 64, 64, 3)]       0         
conv2d_125 (Conv2D)          (None, 32, 32, 64)        3072      
leaky_re_lu_89 (LeakyReLU)   (None, 32, 32, 64)        0         
conv2d_126 (Conv2D)          (None, 16, 16, 128)       131072    
batch_normalization_100 (Batch (None, 16, 16, 128)     512       
Normalization)
leaky_re_lu_90 (LeakyReLU)   (None, 16, 16, 128)       0         
conv2d_127 (Conv2D)          (None, 8, 8, 256)         524288    
batch_normalization_101 (Batch (None, 8, 8, 256)       1024      
Normalization)
leaky_re_lu_91 (LeakyReLU)   (None, 8, 8, 256)         0         
conv2d_128 (Conv2D)          (None, 4, 4, 512)         2097152   
batch_normalization_102 (Batch (None, 4, 4, 512)       2048      
Normalization)
leaky_re_lu_92 (LeakyReLU)   (None, 4, 4, 512)         0         
input_84 (InputLayer)        [(None, 4, 4, 128)]       0         
concatenate_25 (Concatenate) (None, 4, 4, 640)         0         
conv2d_129 (Conv2D)          (None, 4, 4, 512)         328192    
batch_normalization_103 (Batch (None, 4, 4, 512)       2048      
Normalization)
leaky_re_lu_93 (LeakyReLU)   (None, 4, 4, 512)         0         
```

flatten_13 (Flatten) (None, 8192) 0 ['leaky_re_lu_93[0][0]']

dense_54 (Dense) (None, 1) 8193 ['flatten_13[0][0]']

activation_25 (Activation) (None, 1) 0 ['dense_54[0][0]']

==========
Total params: 3,097,601
Trainable params: 3,094,785
Non-trainable params: 2,816
_____

Generator model

_____
Layer (type) Output Shape Param # Connected to
==========

input_85 (InputLayer) [(None, 3400)] 0 []

dense_55 (Dense) (None, 256) 870656 ['input_85[0][0]']

leaky_re_lu_94 (LeakyReLU) (None, 256) 0 ['dense_55[0][0]']

lambda_12 (Lambda) (None, 128) 0 ['leaky_re_lu_94[0][0]']

input_86 (InputLayer) [(None, 100)] 0 []

concatenate_26 (Concatenate) (None, 228) 0 ['lambda_12[0][0]',
'input_86[0][0]']

dense_56 (Dense) (None, 16384) 3735552 ['concatenate_26[0][0]']

re_lu_65 (ReLU) (None, 16384) 0 ['dense_56[0][0]']

reshape_12 (Reshape) (None, 4, 4, 1024) 0 ['re_lu_65[0][0]']

up_sampling2d_48 (UpSampling2D (None, 8, 8, 1024) 0 ['reshape_12[0][0]']
)

conv2d_130 (Conv2D) (None, 8, 8, 512) 4718592 ['up_sampling2d_48[0][0]']

batch_normalization_104 (Batch (None, 8, 8, 512) 2048 ['conv2d_130[0][0]']
Normalization)

```
re_lu_66 (ReLU)              (None, 8, 8, 512)    0          ['batch_normalization_104[0][0]']

up_sampling2d_49 (UpSampling2D  (None, 16, 16, 512)  0         ['re_lu_66[0][0]']
)

conv2d_131 (Conv2D)          (None, 16, 16, 256)  1179648     ['up_sampling2d_49[0][0]']

batch_normalization_105 (Batch  (None, 16, 16, 256)  1024       ['conv2d_131[0][0]']
Normalization)

re_lu_67 (ReLU)              (None, 16, 16, 256)  0          ['batch_normalization_105[0][0]']

up_sampling2d_50 (UpSampling2D  (None, 32, 32, 256)  0         ['re_lu_67[0][0]']
)

conv2d_132 (Conv2D)          (None, 32, 32, 128)  294912      ['up_sampling2d_50[0][0]']

batch_normalization_106 (Batch  (None, 32, 32, 128)  512        ['conv2d_132[0][0]']
Normalization)

re_lu_68 (ReLU)              (None, 32, 32, 128)  0          ['batch_normalization_106[0][0]']

up_sampling2d_51 (UpSampling2D  (None, 64, 64, 128)  0         ['re_lu_68[0][0]']
)

conv2d_133 (Conv2D)          (None, 64, 64, 64)   73728       ['up_sampling2d_51[0][0]']

batch_normalization_107 (Batch  (None, 64, 64, 64)  256         ['conv2d_133[0][0]']
Normalization)

re_lu_69 (ReLU)              (None, 64, 64, 64)   0          ['batch_normalization_107[0][0]']

conv2d_134 (Conv2D)          (None, 64, 64, 3)    1728        ['re_lu_69[0][0]']

activation_26 (Activation)   (None, 64, 64, 3)    0          ['conv2d_134[0][0]']

==================================================================================
==========================
Total params: 10,878,656
Trainable params: 10,876,736
Non-trainable params: 1,920
__________________________________________________________________________________
_____________
```

Adversarial Model

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   Connected to
=================================================================
input_78 (InputLayer)        [(None, 3400)]            0         []

input_79 (InputLayer)        [(None, 100)]             0         []

model_42 (Functional)        [(None, 64, 64, 3),       10878656  ['input_78[0][0]',
                              (None, 256)]                        'input_79[0][0]']

input_80 (InputLayer)        [(None, 4, 4, 128)]       0         []

model_41 (Functional)        (None, 1)                 3097601   ['model_42[0][0]',
                                                                  'input_80[0][0]']

=================================================================
Total params: 13,976,257
Trainable params: 10,876,736
Non-trainable params: 3,099,521
_________________________________________________________________
```

Erkin Özlan - 21901705
Gökcan Kahraman - 21702271
Müşteba Anıl Önder-21604090
Group 12

Code

```
# Batchsizes, epoch number, training data number are all reduced for displaying purposes
# Importing the required modules
from tqdm.notebook import tqdm
import time
import os
import json
import requests
from struct import unpack

import h5py
import pickle
import pandas as pd
import numpy as np
import cv2
import matplotlib.pyplot as plt
import tensorflow as tf
import random

import PIL
from PIL import Image
import keras
from keras import Input, Model
from keras import backend as K
from keras.callbacks import TensorBoard
from keras.layers import Dense, LeakyReLU, BatchNormalization, ReLU, Reshape,
UpSampling2D, Conv2D, Activation, \
    concatenate, Flatten, Lambda, Concatenate, add, Input, LSTM, GRU, RNN, Embedding,
Multiply, TimeDistributed, Bidirectional, RepeatVector, Activation, Flatten, Dropout,
BatchNormalization
from keras.optimizers import Adam
from matplotlib import pyplot as plt
from keras_preprocessing.sequence import pad_sequences
from keras.utils import to_categorical
from keras.models import Model, Sequential

# path names can be changed for downloading the data somewhere else
datTrainFileName = "eee443_project_dataset_train.h5"
```

```
imTrainDirect = "train_images/"
feaTrainDirect = "train_tupled_data"

datTestFileName = "eee443_project_dataset_test.h5"
imTestDirect = "test_images/"
feaTestDirect = "test_tupled_data"
# Functions to obtain the data

def imgDown(data, namPath):

# In this function, we downloaded the images from the given url

    tim = time.time()
    i = 0

    if not os.path.exists(namPath):
        os.makedirs(namPath)

    for url in data:

        url = url.decode()
        name = url.split("/")[-1].strip()
        path = os.path.join(namPath, name)

        if not os.path.exists(path):

            hed = {'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/74.0.3729.169 Safari/537.36'}
            resp = requests.get(url, stream=True)

            # check if data is obttained successfully
            if resp.status_code == 200:
                with open(path, 'wb') as outfile:
                    outfile.write(resp.content)

        # prints affirmation at each 1000 iterations
        if i % 1000 == 1:

            timRes = time.time() - tim
            timPerIter = timRes/i
            print("Total passed minute is: {:.2f} . There is {:.2f} seconds per iteration. Our iteration
is: {}".format(timRes/60, timPerIter, i))

        i += 1

# Functions for data processing
```

```
def StringFromCaptArry(captArry):

# This function is used for prediction and demonstration

    captList = []

    capt = ""

    if(captArry.ndim == 1):
        captArry = np.expand_dims(captArry, axis=0)

    for caps in captArry:

        for NamWord in caps:

            if (NamWord == 'x_NULL_') or (NamWord == 'x_START_') or (NamWord ==
'x_END_'):
                continue

            capt += NamWord + " "


        captList.append(capt.strip())
        capt = ""


    return captList


def captGet(namList, imid, cap, name):

    indexx = namList.index(name) + imid.min()

    return cap[np.where(imid == indexx)]

def PreProcCreation(directImg, shuffle=False):

    def fileMethod(path):
        img = tf.io.read_file(path)
        img = tf.image.decode_jpeg(img, channels=3)
        img = tf.image.resize(img, (64, 64))
        return img

    def nameProcess(path):
        name = path.numpy().decode().split("\\")[-1]
```

```
        return name

    def proc(path):
        name = tf.py_function(nameProcess, [path], tf.string)
        img = tf.py_function(fileMethod, [path], tf.float32)
        return (img, name)

    datFile = tf.data.Dataset.list_files(str(directImg) + "*.jpg", shuffle=shuffle)

    return datFile.map(lambda x: proc(x))


def create_features(namFile, ims, namList, imid, cap, process_size = 250):


    length = 0

    with open(namFile, "wb") as outfile:

        for data in tqdm(ims.batch(process_size)):

            image = data[0]
            name = data[1].numpy()

            feature = image
            for i in range(feature.shape[0]):

                feat = feature[i]
                namm = name[i].decode()
                cpt = captGet(namList, imid, cap, namm)

                tp = (feat, cpt, namm)
                pickle.dump(tp, outfile)

                length += 1

    outfile.close()
    return length


def installPickle(namFile):

    with open(namFile, "rb") as f:
        while True:
            try:
                yield pickle.load(f)
```

```
            except EOFError:
                break



def dataGet(directFeat, url=None, imid=None, cap=None, directImg=None):


# Main function that combines the captions and corresponding images

    len = -1

    if not os.path.isfile(directFeat):

        if not directImg:
            raise Exception("No image directory given. Enter image directory for feature
extraction.")

        listName = [u.split("/")[-1].strip() for u in np.char.decode(url).tolist()]
        images = PreProcCreation(directImg)
        len = create_features(directFeat, images, listName, imid, cap)

    dset = tf.data.Dataset.from_generator(installPickle, args=[directFeat],
output_types=(np.float32,np.int32, tf.string))

    if len == -1:
        len = dset.reduce(0, lambda x, _: x + 1).numpy()

    return dset, len

def GetEmbed(w2ix):
# This function creates the embedding matrix with our vocabulary


    # Load Glove vectors
    pathGlove = 'glove.6B.200d.txt'

    indxEmbed = {} # empty dictionary
    fil = open(pathGlove, encoding="utf-8")

    for line in fil:
        val = line.split()
```

```
        word = val[0]
        coefficients = np.asarray(val[1:], dtype='float32')
        # if (word == 'startseq' or word == 'unk' ):
        #   print(word)

        indxEmbed[word] = coefficients
    fil.close()
    print('Our total vector of word is: %s' % len(indxEmbed))


    dimEmbed = 200
    totVocab = 1004

    # Get 200-dim dense vector for each of the 10000 words in out vocabulary
    mtxEmbed = np.zeros((totVocab, dimEmbed))

    for word, i in w2ix.items():

        if (word == 'x_UNK_'):
          word = 'unk'

        vectEmbed = indxEmbed.get(word)
        if  vectEmbed is None:
          print('There is no existing word for capturing')

        if vectEmbed is not None:
            # Words not found in the embedding index will be all zeros
            mtxEmbed[i] = vectEmbed

    return mtxEmbed

def HotOne(X, size=1004):

    X = X -1
    onehott = np.zeros((X.shape[0], 0))
    temp = np.zeros((X.shape[0], size))
    tempp = np.arange(X.shape[0])
    temp[tempp, X[:,0]] = 1
    onehott = np.hstack((onehott, temp))

    return onehott

# Get the train data
fill = h5py.File(datTrainFileName, 'r')

for key in list(fill.keys()):
```

```
    print(key, ":", fill[key][()].shape)

captTrain = fill["train_cap"][()]
imidTrain = fill["train_imid"][()]
urlTrain = fill["train_url"][()]
codeWord = fill["word_code"][()]


dif = pd.DataFrame(codeWord)
dif = dif.sort_values(0, axis=1)
words = np.asarray(dif.columns)

w2ixx = {}
for i in range(len(words)):
  word = words[i]
  w2ixx[word] = i

# To download the images, this code needs to be run but lasts long
# If the data is already downloaded, path name can be changed accordingly.


# save_ims(train_url,TRAIN_IMAGES_DIRECTORY )

# This part is used for combining the images and their related captions using
# the train_imid data, also the image resolutions are fixed to 64x64

datTrain, datTrainLen = dataGet( feaTrainDirect, urlTrain, imidTrain, captTrain,
imTrainDirect)
print( "Total data obtaining proportion is: {} / {} ".format(datTrainLen, len(urlTrain)) )

# Similar with the training data, test data is combined and resolutions are fixed
fill = h5py.File(datTestFileName, "r")

captTest = fill["test_caps"][()]
imidTest = fill["test_imid"][()]
urlTest = fill["test_url"][()]


datTest, datTestLength = dataGet(feaTestDirect, urlTest, imidTest, captTest, imTestDirect)

print( "The proportion of data obtaining is: {} / {} ".format(datTestLength, len(urlTest),  ))


for d in datTrain.shuffle(500).take(1):
    feats = d[0]
    namImg= d[2].numpy().decode()
```

```
    capt = d[1].numpy()


    imgg = cv2.imread(imTrainDirect + namImg)
    imgg = cv2.cvtColor(imgg, cv2.COLOR_BGR2RGB)
    plt.imshow(imgg)
    plt.show()
    captt = StringFromCaptArry(words[capt])

    print(captt[0])

    print(feats.shape, capt.shape)

for d in datTrain.shuffle(1000).take(1):
    feats = d[0].numpy()
    namImg= d[2].numpy().decode()
    capt = d[1].numpy()
xxx = feats
plt.imshow(xxx.astype('uint8'))

# Creating the embedding matrix from glove vectors
matrixEmbed = GetEmbed(w2ixx)

# Functions for the GAN architecture


def lossKL(corY, predY):
    u = predY[:, :128]
    sigLogarithm = predY[:, :128]
    totLoss = -sigLogarithm + .5 * (-1 + K.exp(2. * sigLogarithm) + K.square(u))
    totLoss = K.mean(totLoss)
    return totLoss


def LossCustGen(corY, predY):

    return K.binary_crossentropy(corY, predY)

def ImgSave(img, path):
    #rgb image saving
    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
    ax.imshow(img)
    ax.axis("off")
    ax.set_title("Image")
```

```
    plt.savefig(path)
    plt.close()

def cGen(x):
    u = x[:, :128]
    sigmaLogaritmm = x[:, 128:]
    derivStandart = K.exp(sigmaLogaritmm)
    epsilon = K.random_normal(shape=K.constant((u.shape[1],), dtype='int32'))
    crr = derivStandart * epsilon + u
    return crr

def caModelBuild():
    # Cond
    totLayerInp = Input(shape=(3400,))
    x = Dense(256)(totLayerInp)
    x = LeakyReLU(alpha=0.2)(x)
    mod = Model(inputs=[totLayerInp], outputs=[x])
    return mod

def CompresBuildEmbedModel():
    # Compressing the embedding matrix

    totLayerInp = Input(shape=(3400,))
    x = Dense(128)(totLayerInp)
    x = ReLU()(x)

    mod = Model(inputs=[totLayerInp], outputs=[x])
    return mod

def stage1BuildGen():

    totLayerInp = Input(shape=(3400,))
    dns = Dense(256)(totLayerInp)
    sigmaLogU = LeakyReLU(alpha=0.2)(dns)

    crr = Lambda(cGen)(sigmaLogU)

    totLayerInp2 = Input(shape=(100,))

    InpGen = Concatenate(axis=1)([crr, totLayerInp2])

    dns = Dense(128 * 8 * 4 * 4, use_bias=False)(InpGen)
    dns = ReLU()(dns)

    dns = Reshape((4, 4, 128 * 8), input_shape=(128 * 8 * 4 * 4,))(dns)
```

```
    dns = UpSampling2D(size=(2, 2))(dns)
    dns = Conv2D(512, kernel_size=3, padding="same", strides=1, use_bias=False)(dns)
    dns = BatchNormalization()(dns)
    dns = ReLU()(dns)

    dns = UpSampling2D(size=(2, 2))(dns)
    dns = Conv2D(256, kernel_size=3, padding="same", strides=1, use_bias=False)(dns)
    dns = BatchNormalization()(dns)
    dns = ReLU()(dns)

    dns = UpSampling2D(size=(2, 2))(dns)
    dns = Conv2D(128, kernel_size=3, padding="same", strides=1, use_bias=False)(dns)
    dns = BatchNormalization()(dns)
    dns = ReLU()(dns)

    dns = UpSampling2D(size=(2, 2))(dns)
    dns = Conv2D(64, kernel_size=3, padding="same", strides=1, use_bias=False)(dns)
    dns = BatchNormalization()(dns)
    dns = ReLU()(dns)

    dns = Conv2D(3, kernel_size=3, padding="same", strides=1, use_bias=False)(dns)
    dns = Activation(activation='tanh')(dns)

    wrkStage1 = Model(inputs=[totLayerInp, totLayerInp2], outputs=[dns,sigmaLogU])

    return wrkStage1

def stage1BuildDisc():

    totLayerInp = Input(shape=(64, 64, 3))

    dns = Conv2D(64, (4, 4),
            padding='same', strides=2,
            input_shape=(64, 64, 3), use_bias=False)(totLayerInp)
    dns = LeakyReLU(alpha=0.2)(dns)

    dns = Conv2D(128, (4, 4), padding='same', strides=2, use_bias=False)(dns)
    dns = BatchNormalization()(dns)
    dns = LeakyReLU(alpha=0.2)(dns)

    dns = Conv2D(256, (4, 4), padding='same', strides=2, use_bias=False)(dns)
    dns = BatchNormalization()(dns)
    dns = LeakyReLU(alpha=0.2)(dns)

    dns = Conv2D(512, (4, 4), padding='same', strides=2, use_bias=False)(dns)
    dns = BatchNormalization()(dns)
```

```
    dns = LeakyReLU(alpha=0.2)(dns)

    totLayerInp2 = Input(shape=(4, 4, 128))

    InpComb = concatenate([dns, totLayerInp2])

    dns2 = Conv2D(64 * 8, kernel_size=1,
                  padding="same", strides=1)(InpComb)
    dns2 = BatchNormalization()(dns2)
    dns2 = LeakyReLU(alpha=0.2)(dns2)
    dns2 = Flatten()(dns2)
    dns2 = Dense(1)(dns2)
    dns2 = Activation('sigmoid')(dns2)

    modDisStage1 = Model(inputs=[totLayerInp, totLayerInp2], outputs=[dns2])
    return modDisStage1

def advModelBuild(ModGen, modDis):
    totLayerInp = Input(shape=(3400,))
    totLayerInp2 = Input(shape=(100,))
    totLayerInp3 = Input(shape=(4, 4, 128))

    dns, sigmaLogU = ModGen([totLayerInp, totLayerInp2])

    modDis.trainable = False
    corr = modDis([dns, totLayerInp3])

    modd = Model(inputs=[totLayerInp, totLayerInp2, totLayerInp3], outputs=[corr,
sigmaLogU])
    return modd

lenData = 30 #train_data_length
trData = datTrain.take(lenData)

lenValue = round(lenData * 0.15)
trValue = lenData - lenValue

datVal = trData.take(lenValue)
datTr = trData.skip(lenValue)

# Captions and Images are received from train and val data

str = []
strr = []
for data in datTr:
```

```
        feat = data[0].numpy()
        cp = data[1].numpy()

        str.append(feat)
        strr.append(cp[0])

sttr = []
sttrr = []
for data in datVal:

        featt = data[0].numpy()
        cpp = data[1].numpy()

        sttr.append(featt)
        sttrr.append(cpp[0])

# Here, text embedding is done using the embedding matrix and it is flattened.

relEmbed = []
for i in range(trValue):
    emb = np.array(strr[i:i+1])
    em = np.transpose(emb)
    emm = HotOne(em)
    emmm = np.matmul(emm,matrixEmbed)
    relEmbed.append(emmm)

bssEm = np.array(relEmbed)
bssEm = bssEm.reshape((trValue,3400))
bssEm.shape

datNum1 = lenValue
relEmb1 = []
for i in range(lenValue):
    emb1 = np.array(sttrr[i:i+1])
    em1 = np.transpose(emb1)
    emm1 = HotOne(em1)
    emmm1 = np.matmul(emm1,matrixEmbed)
    relEmb1.append(emmm1)

bssEm1 = np.array(relEmb1)
bssEm1 = bssEm1.reshape((lenValue,3400))

bssEm1.shape,bssEm.shape

trAr = np.array(str[0:trValue])
trEmbed = bssEm
```

```
valAr = np.array(sttr[0:trValue])
valEmbed = bssEm1


trAr.shape,valAr.shape

# The training of the model

if __name__ == '__main__':

    totImg = 64
    totBatch = 4
    dimensionZ = 100
    LRgenST1 = 0.001
    LRdistST1 = 0.001
    LRdsST1 = 600
    epochs = 1000
    totCond = 128


    # Optimizers that we use for generator and discriminator
    Optdis = Adam(lr=LRdistST1, beta_1=0.5, beta_2=0.999)
    Optgen = Adam(lr=LRgenST1, beta_1=0.5, beta_2=0.999)


    # Networks implemented and build
    ModCA = caModelBuild()
    ModCA.compile(loss="binary_crossentropy", optimizer="adam")

    disST1 = stage1BuildDisc()
    disST1.compile(loss='binary_crossentropy', optimizer=Optdis)

    genST1 = stage1BuildGen()
    genST1.compile(loss="mse", optimizer=Optgen)

    modEmbedComp = CompresBuildEmbedModel()
    modEmbedComp.compile(loss="binary_crossentropy", optimizer="adam")

    modAdver = advModelBuild(ModGen=genST1, modDis=disST1)
    modAdver.compile(loss=['binary_crossentropy', lossKL], loss_weights=[1, 2.0],
                     optimizer=Optgen, metrics=None)

    # Fake and Real Labels

    labCor = np.ones((totBatch, 1), dtype=float) * 0.9
```

```
    labWro = np.zeros((totBatch, 1), dtype=float)

    for epoch in range(epochs):
        print("Please wait...")
        print("Total Epoch:", epoch)
        print("Total Batches:", int(trAr.shape[0] / totBatch))

        LossGen = []
        LossDis = []

        # Here is batch gradient descend
        totBatchh = int(trAr.shape[0] / totBatch)
        for index in range(totBatchh):
            print("Total Batch:{}".format(index+1))


            # Batch data and noise,
            noiseZ = np.random.normal(0, 1, size=(totBatch, dimensionZ))
            batchImg = trAr[index * totBatch:(index + 1) * totBatch]
            batchEmbed = trEmbed[index * totBatch:(index + 1) * totBatch]
            batchImg = (batchImg - 127.5) / 127.5

            # Forward pass through the generator network
            WroImg, _ = genST1.predict([batchEmbed, noiseZ], verbose=3)

            # Compressing the text embedding further
            embedComp = modEmbedComp.predict_on_batch(batchEmbed)
            embedComp = np.reshape(embedComp, (-1, 1, 1, totCond))
            embedComp = np.tile(embedComp, (1, 4, 4, 1))

            RealLossDis = disST1.train_on_batch([batchImg, embedComp],
                                                np.reshape(labCor, (totBatch, 1)))
            FakeLossDis = disST1.train_on_batch([WroImg, embedComp],
                                                np.reshape(labWro, (totBatch, 1)))
            WrongLossDis = disST1.train_on_batch([batchImg[:(totBatch - 1)], embedComp[1:]],
                                                 np.reshape(labWro[1:], (totBatch-1, 1)))

            LossD = 0.5 * np.add(RealLossDis, 0.5 * np.add(WrongLossDis, FakeLossDis))

            # Training of the generator network

            LossG = modAdver.train_on_batch([batchEmbed, noiseZ,
embedComp],[K.ones((totBatch, 1)) * 0.9, K.ones((totBatch, 256)) * 0.9])
            LossDis.append(LossD)
            LossGen.append(LossG)
```

```
        print("Loss of D:{}".format(LossD))
        print("Loss of G:{}".format(LossG))

        if epoch % 1 == 0:

            # Predict on Validation see the loss
            NoiseZZ = np.random.normal(0, 1, size=(totBatch, dimensionZ))
            batchEmbed = valEmbed[0:totBatch]
            WroImg, _ = genST1.predict_on_batch([batchEmbed, NoiseZZ])
            lossVal =  modAdver.evaluate([batchEmbed, noiseZ,
embedComp],[K.ones((totBatch, 1)) * 0.9, K.ones((totBatch, 256)) * 0.9])
            print("Validation Loss:{}".format(lossVal))

            # Save images
            for i, img in enumerate(WroImg[:10]):
                ImgSave(img, "Printed Results/gen_{}_{}.png".format(epoch, i))

#           Save the model in each epoch
#     stage1_gen.save_weights("stage1_gen.h5")
#     stage1_dis.save_weights("stage1_dis.h5")

# summary of generator network
genST1.summary()

lenTest = 3000 #test_data_length
datTest = datTest.take(lenTest)

xRowColll = []
xRowwColll = []
for data in datTest:

            feattt = data[0].numpy()
            cpsss = data[1].numpy()

            xRowColll.append(feattt)
            xRowwColll.append(cpsss[0])

realEmbbb = []
for i in range(lenTest):
    embeddd = np.array(xRowwColll[i:i+1])
    emRowColll = np.transpose(embeddd)
    emRowwColll = HotOne(emRowColll)
    emRowwwColll = np.matmul(emRowwColll,matrixEmbed)
    realEmbbb.append(emRowwwColll)

bassEmbeddd = np.array(realEmbbb)
```

```
bassEmbeddd = bassEmbeddd.reshape((lenTest,3400))
bassEmbeddd.shape

testX = np.array(str[0:lenTest])
testEmbeds = bssEm

testX.shape,testEmbeds.shape

btcEmbed1 = testEmbeds[0:totBatch]
imgWro = genST1.predict_on_batch([btcEmbed1, NoiseZZ])

cpExamp = np.array(strr)
cpp = StringFromCaptArry(words[strr[0]])
print(cpp)
# needs to be trained, output won't give a good result
imggg = imgWro[0]
plt.imshow(imggg[0])
```