EEE 443 - Neural Networks

27 May 2022

Final Project

Text To Image

Osman Buğra Aydın | 21704100

Emre Orta | 21703335

Abdullah Arda Aşçı | 21702748

Ali Necat Karakuloğlu | 22101543

Github: github.com/ArdaAsci/Caption2Image

Drive: Preprocessed Data

Abstract

In this project, various types of neural network architectures are used and combined to create a system that generates images according to the given text input. To achieve this functionality, the data that is provided by the instructors is analyzed and preprocessed. In the preprocessing stage, images and their corresponding captions are extracted from the dataset to be converted into image and text encodings. We have been inspired by an ablated diffusion model(ADB) to generate image encodings that will be decoded into another image which is conditioned by the text input. We designed a 6-layered Transformer to obtain the images by decoding text inputs. Our model has U-Net-like architecture which includes many different types of layers such as convolutional and up-convolutional. For the decoding part, we have used a model which has upsampling layers like U-Net to generate encoded images. We have trained our model with the preprocessed data and monitored each epoch not to have an overfitting problem by checking the results of the loss function.

1. Introduction

This paper describes a framework that has multiple neural networks cooperating to perform new images from the given captions as text input. This is a highly debated topic and there are lots of recent works about this functionality which is known as Text2Image. Besides, Google recently offered a new state-of-the-art Imagen that generates high-resolution images from the given captions[1]. It creates these photorealistic images with the help of a diffusion model as we have inspired in

our project. Diffusion models perform better for synthesizing photorealistic images compared to the other algorithms.

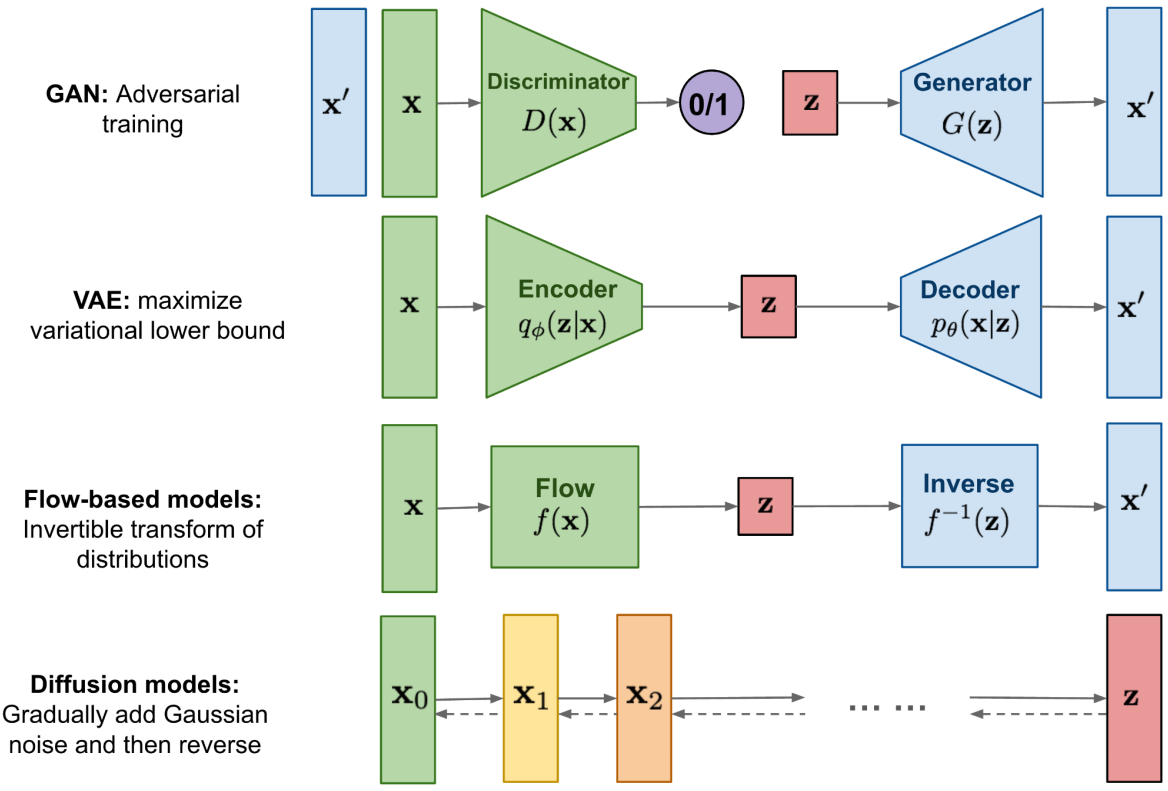

Figure 1. Comparison of various generative networks

Generative adversarial networks are dominantly used for image generation and are superior to the other types of networks in terms of Fréchet inception distance(FID)[3]. It is a metric that represents the quality of the generated image[4]. On the other hand, it is hard to train, error-prone during the training, and lacks diversity in the image synthesis. On the contrary, diffusion models are relatively easier to train and it offers a broad scope of diversity in image generation. Furthermore, it allows us to condition image synthesis based on the given text input. These were the main reasons why we have determined to construct a model inspired

by an ablated diffusion model to convert text encodings, that we obtain from the captions to the image encodings.

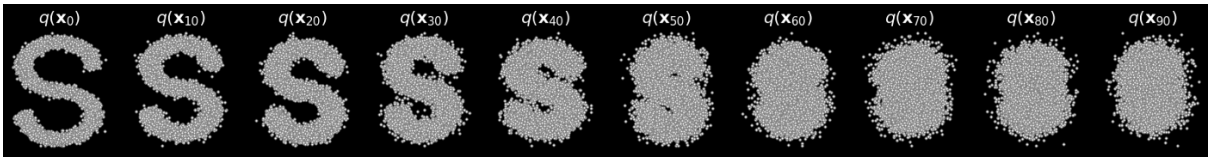

Figure 2. Illustration of forward propagation in diffusion models

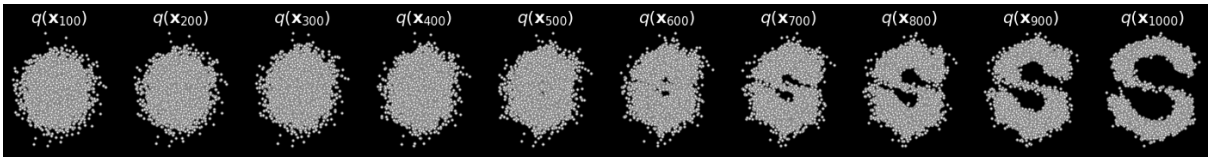

Figure 3. Illustration of forward propagation in diffusion models

The main idea of diffusion models is simply adding noise to the input of the network and recovering it by removing the noise in a probabilistic manner. The number of steps for adding and removing the noise has to be the same as can be seen in Figure 1. Adding noise to the inputs of the network will result in the systematic decay of information[5]. As a matter of fact, systematic decay enables us to recover or reconstruct noisy images. It is a similar approach to how VAE achieves generating new images but the diffusion model aims to learn noise distributions rather than data distribution. Figures 2 and 3 show the effects of these processes on data at a high level.

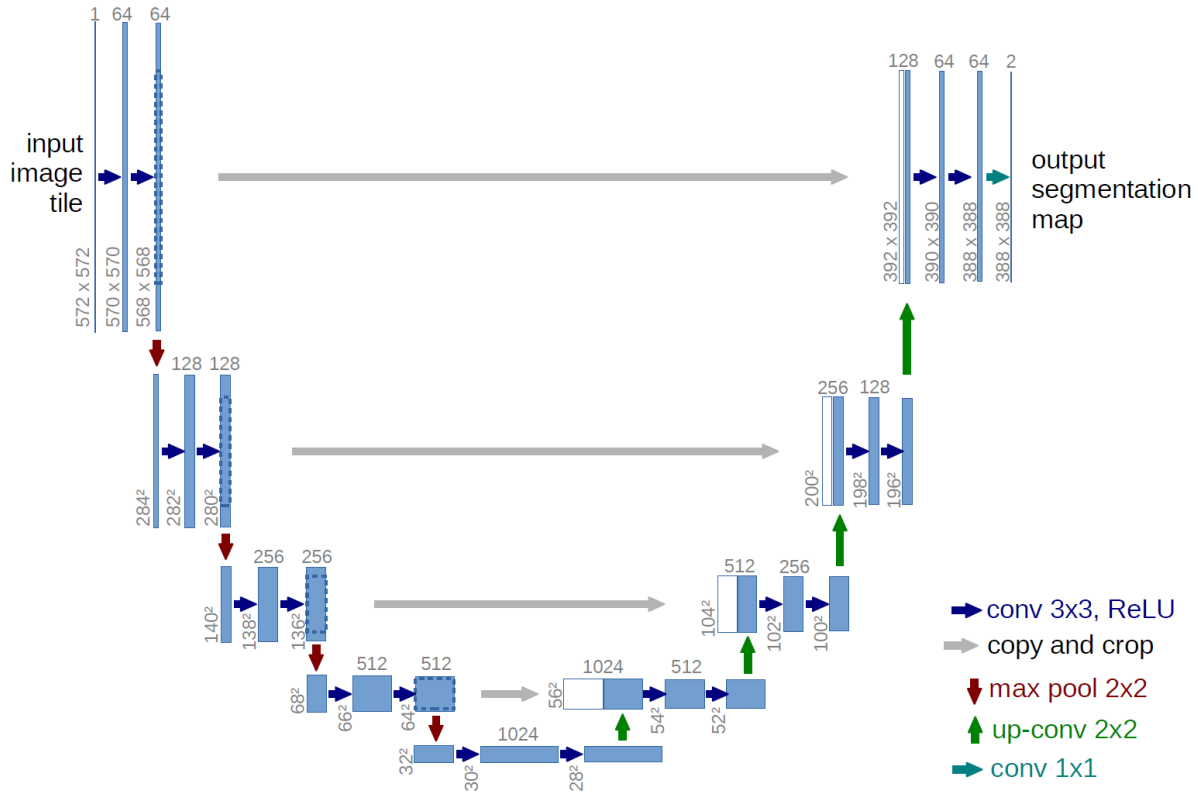

Figure 4. The Network Architecture of U-Net

The architecture of the diffusion models differs from the other generative networks like GAN, VAE, and flow-based models. Diffusion models generally have similar architecture or inherit U-Net-like networks with some modifications. Figure 4 illustrates a U-Net model that uses different neural networks to obtain the required output. In our case, the input will be the text encoding of a specific caption and the output will be the image encoding which encodes conditionally generated images by the captions. With the image encoding, we will use a U-Net-like decoder that transforms our image encoding into a real image. Unlike the general usage of the CNNs which is to find solutions to the classification problems, localization is the main content of the output[6]. As it offers powerful localization which enables us to create

different images but it has two drawbacks which are redundancy and using large patches. Since different patches need to be executed one by one and overlapping problems of the patches, redundancy will occur. Also, the larger the patches, the more details disappear due to the downsampling done by the red arrow(max pool) in Figure 4. It will reduce the overall accuracy of the model. It was also one of the considerations in our project as we tried different sizes of patches to find the optimal one for increasing accuracy and decreasing redundancy.

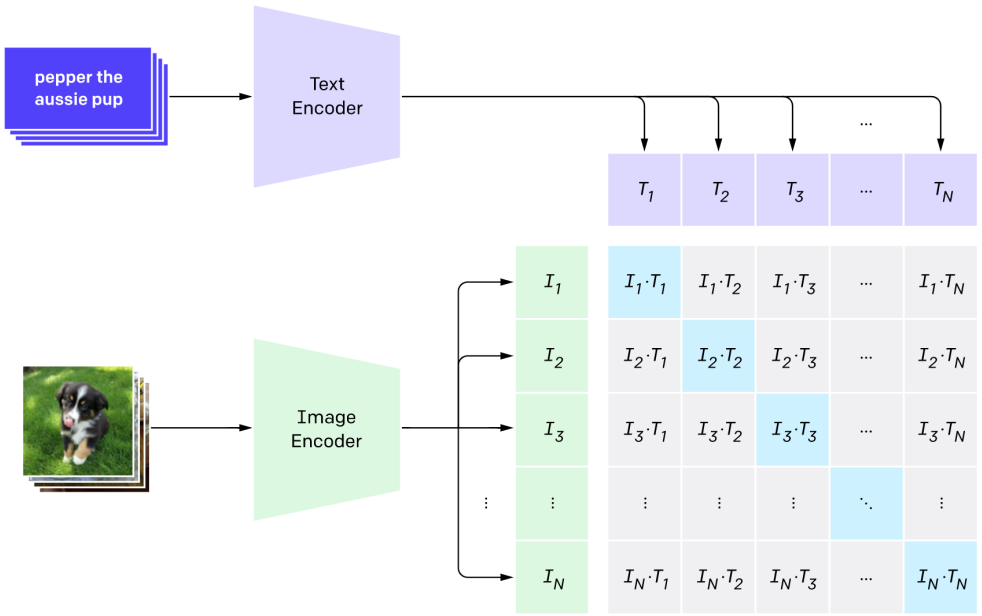

Figure 5. Illustration of Clip's Architecture

As there is much research and work on Text2Image, we have investigated some projects like DALL-E, GLIDE, CLIP, VQGAN, and Imagen. Before starting our project, we have detailly examined these projects to understand how they implemented their framework. Since we have determined that the parts inspired by diffusion models will be used in this project, we have eliminated the VQGAN because it is another implementation of the GAN model. The main structure of CLIP includes three parts such as encoding both images and text, creating a classifier

from text, and creating a joint representation matrix of the encodings[7]. We did not use a similar approach with this paper but we use the idea of encoding images and text to use for our neural network architecture. Also, CLIP uses one-shot prediction by inspiring from the representation matrix formed by the multiplication of text and image encodings as in Figure 5. GLIDE stands for Guided Language to Image Diffusion for Generation and Editing. It has a framework that has three main components which are a text transformer, an ADB that converts the input of the transformer into 64x64 images, and an upsampling model to produce a high-quality version of previous images of 256x256. We analyzed this paper to decide the structure of our project. However, we decided to use an extra image encoder to train our image decoder which will be used to generate new images.

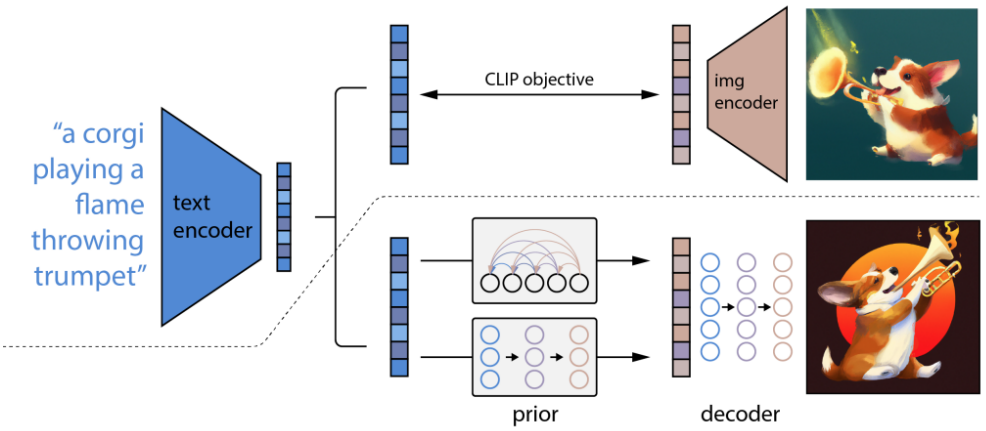

Figure 6. High Level view of DALL-E 2 project

The DALL-E 2 project has offered a more complex structure compared to the others. It uses text and image encoders of CLIP[9]. Text encodings are fed into a prior step which can be done by two different models such as autoregressive or diffusion prior[9]. The output will be the encoding of the newly generated image as in Figure 6. Finally, the decoding part will be done by a diffusion decoder. This structure helped

us to create our framework. We prefer a transformer model rather than an autoregressive since it is more complex and gives better results.

2. Methods

There are three main parts of this project that interact with each other to produce new images from given captions. Firstly, we have a preprocessing step in which we reduce unnecessary parts and convert them to a state to be usable by our structure. The captions and images are transformed into encodings to be used in prior and decoder stages. The prior stage is another model that transforms text encoding into image encoding. Lastly, the decoder stage creates the final newly generated image. We have used PyTorch to implement our framework.

A.) Preprocess

This section is divided into three parts which are introducing the dataset, cleaning the data, and creating the encodings.

Dataset was the first material that we analyzed in this project. It is a widely used dataset named COCO that is categorized to help segmentation, object detection, and caption generation neural networks to train[10]. The keys of the data are train_cap, train_imid, train_ims, train_url, word_code, test_caps, test_imid, test_ims, and test_url. To start, train_cap([400135,17]) consists of the captions for each image in the form of digits which represents indices of the vocabulary, train_imid([400135, 1]) stands for the index table that corresponds to each caption for every image, train_url([82783, 1]) contains the URLs of each image that we need to download manually in Flickr, word_code([1004, 1]) is the dictionary that converts words into indexes, and train_ims has previously trained vector for each image. We

did not use train_ims since it halves the points that we can get from the project. The usefulness of all kinds explained above is the same with test data. The shapes of the test data cap, imid, and URL are respectively [195954, 17], [195954, 1], [40504, 1].

Cleaning the data is an essential part of our project to give better results. The code that we provide for this section is the “data_generator.ipynb” file. It can be inspected for further understanding. To summarize the steps, we loaded and calculated tokenized captions to save them into the Torch by checking the according to files. Since there are many images, we realized that 15 percent of the image URLs are broken. Therefore, we omit invalid URLs to clean the data. Also, 10 percent of the training data is used as validation data.

Creating encoding is the crucial step for our framework to train and generate new images. To create encodings, we have used CLIP vision transformers named ViT-B/32. The main function of ViT-B/32 is to map text or image to 512-length sequences that represent the features extracted from a particular caption/image. Besides, using CLIP enabled us to overcome a possible problem which is caused by the different sizes of the images in the dataset. As a result, we had the same dimension for all of the encodings, even if we have different-sized images. As a matter of fact, even popular frameworks like DALL-E and GLIDE use the encoding function of the CLIP[11].

B.) Transformer Prior

In this part, we have offered a transformer decoder model which takes text encoding as input and outputs an image encoding that represents a newly generated image. We have constructed a complex model with 6 layers.

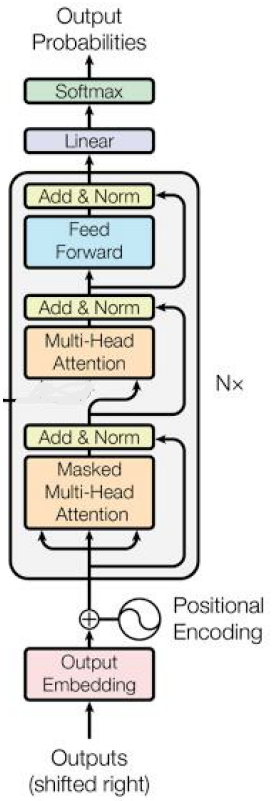

Figure 7. High level view of Transformer Model

Transformers include a recurrent neural network(RNN) to produce results based on previous outcomes. RNN is advantageous to use for our project to detect and reveal common patterns in a series of data[12]. It enables us to understand the relations between captions and images over time.

$$H_t = \phi_t(X_t W_{xh} + H_{t-1} W_{hh} + b_h) \quad (1)$$

It is the mathematical formula of how each block calculates the information that will be conveyed to the next block where t and h refer to time and hidden respectively. To examine, Ht is equal to the output of a sigmoidal activation function that includes the sum of multiplication of input at a time t and input to hidden weights, multiplication of the previous Ht-1 and hidden to hidden weights, and a bias. By training our images, we relate the captions with the images as we consider the relations of former

captions. We use fixed-sized vectors to feed the network and get a sequence of the same size as the output. We have utilized attention blocks to simply investigate the relations of words in a caption. We used the softmax activation function to scale and normalize the weights of each word with respect to 1. We have investigated to find a better alternative to classical softmax function like Taylor softmax but it did not respond to our needs[13]. The equation of the softmax function is given below.

$$sm(z)_i = e^{z_i} / \sum_{j=1}^{K} e^{z_j} \text{ for i = 1…K and z = } (z_1 \dots z_k) \epsilon R^K \quad (2)$$

Attention block is a popular implementation to reduce the time complexity of convolution blocks by taking the dot product of kernel feature maps so that the big O complexity reduces from O(n2) to O(n)[14]. Another advantage of the attention block is each block is distinguished from another block. As we can see from Figure 7, we have 2 attention blocks in our model. Since they are separate, we can utilize parallelization which will result in efficient computations[15]. The actual reason that we used attention blocks is to calculate image encodings with respect to previous outcomes of the network for a specific input without considering its current value. Thus, we have found a more robust way to come up with accurate results. After the attention blocks, to get the output in the form that we wanted, which is a vector of 512, we have constructed a feed-forward network(FFN). As a loss function, we have used mean squared error(MSE) which takes the average of squares of calculated errors between expected and actual value[16].

$$E[(X - \widehat{X})^2] = E[(X - g(Y))^2] \quad (3)$$

We have decided to make use of ADAM optimizer since it gives better results compared to the alternatives with the MSE loss function that we use[17].

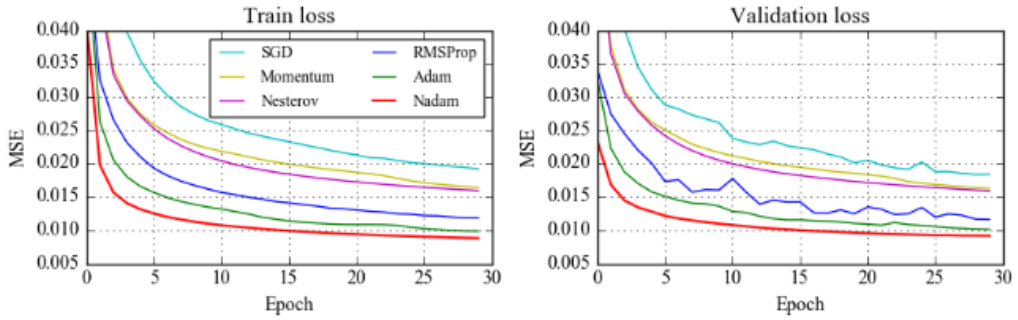

Figure 8. Comparison of Optimizers with MSE

ADAM is a stochastic optimization method that enables us to use various parameters for moments[18]. Therefore, it is advantageous with respect to the classical implementation of gradient descent, momentum, etc. To sufficiently train this model, we set some parameters. We took 25 epochs to train. In each epoch, we have planned the batch size as 2048, and the learning rate is 1e-4. It took 20 and 4 minutes in CPU and CUDA respectively. We stopped propagation when validation loss started to increase. The oscillation happens when the learning rate is insufficiently smaller and misses the minimum point.

C.) Decoder

In this part, we have constructed a model which we inspired by diffusion models in the DALLE-2 project. Unlike the DALLE-2, we did not use the denoising technique to decode images, we created our model which is 5 layers with various types. Before diving into the details of our model, the general logic of the right part of the U-Net model shown in Figure 4 is similar to our model. The left part of the U-Net is a type of convolutional encoder that creates a latent vector. This vector is

produced to be converted into an image by decoders. Since we already have image encodings which are the outputs of the transformer model that we offered in the previous section, we have developed a U-Net-like decoder model which is inspired by the decoding side of general U-Net. In the first four layers, we have a combination of Conv2DTranspose, batch normalization, and relu as activation functions. Lastly, we have Conv2D and tanh as activation. The first four layers' duty is to up-sample the feature map so that we can decode the image encodings without losing the features of the images generally. The function of the last layer is mapping feature vectors to their corresponding classes in a certain number. For training, we have used the image encoding that is created by CLIP in the preprocess section. We took vectors of 512 as input and produced outputs of 64x64 images. We have used MSE to compare the difference between original and generated images with respect to their pixels. ADAM is selected for the optimizer algorithm. We have trained our model for 250 epochs in which batch size is 128 and the learning rate is 2 * 1e-4. The training process was held in a GPU which is RTX 3070 and took 125 minutes. To prevent the overfitting problem, we stopped our process when the loss function started not to reduce significantly. After training, we used our model to generate images that came from prior encoding.

3. Results

A. Prior+Decoder Model

In this section, some sample network outputs and graphical representation of the network performances are presented.

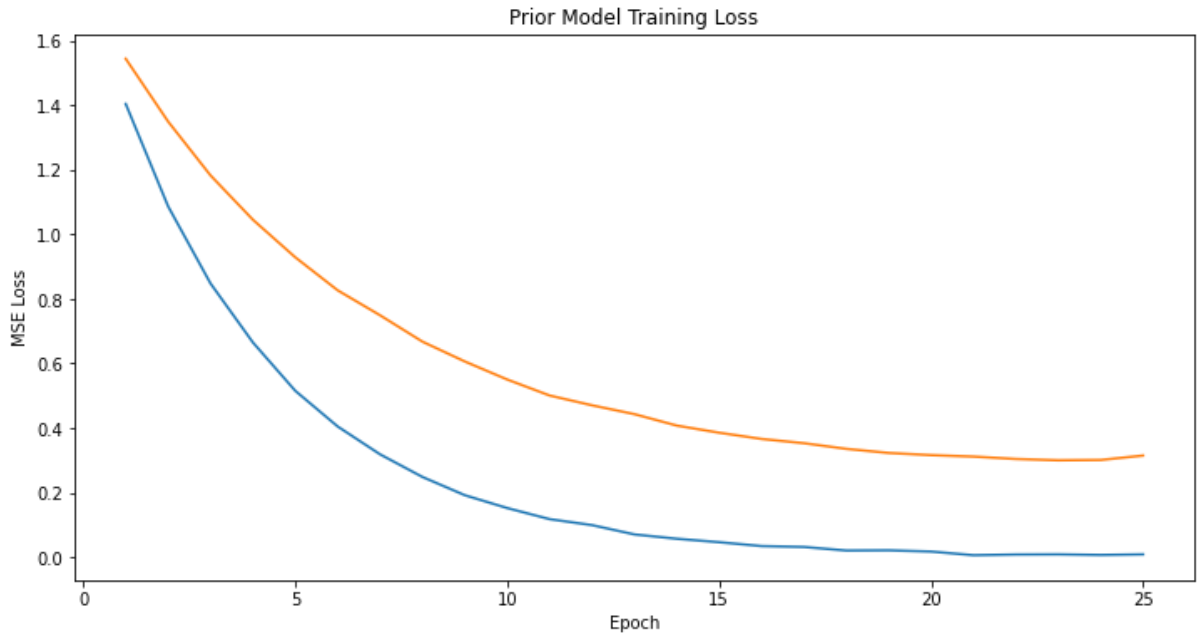

Figure 9. MSE losses of Prior Transformer Network for training and validation

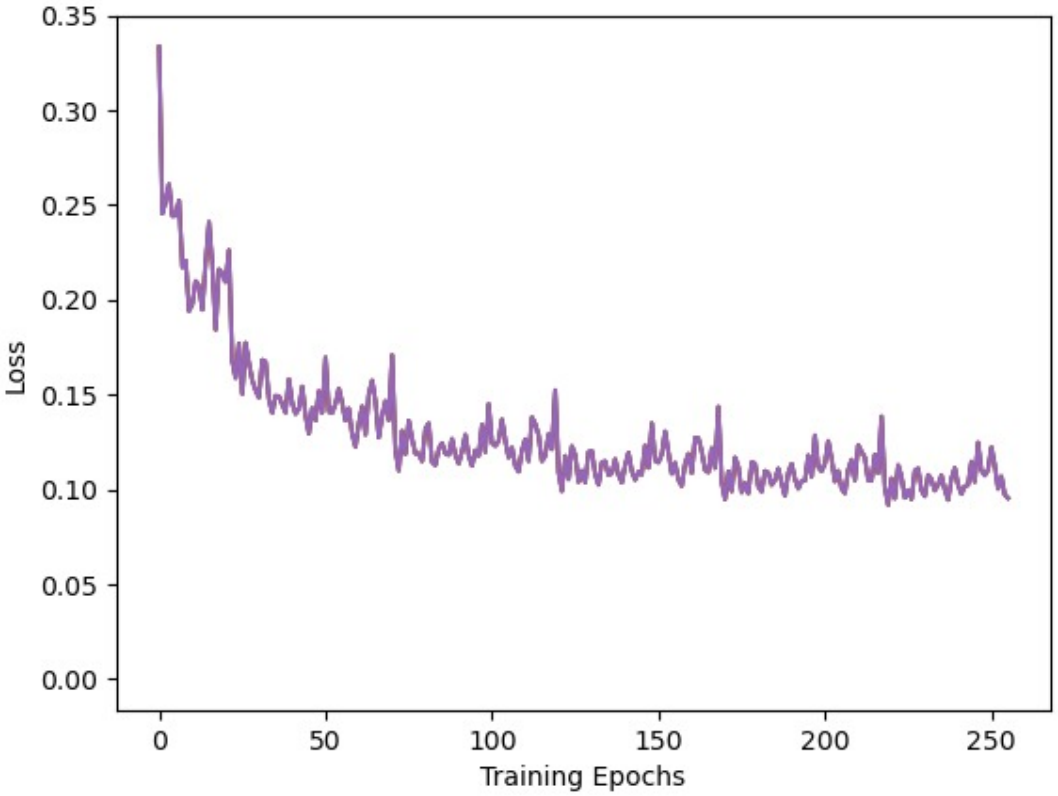

Figure 10. MSE loss of the Decoder Network

Figure 11. The figure shows the output of the decoder network for a sample image. The network is fed with the embeddings of the caption and it outputs the corresponding image prediction as in the first image of Figure 11.

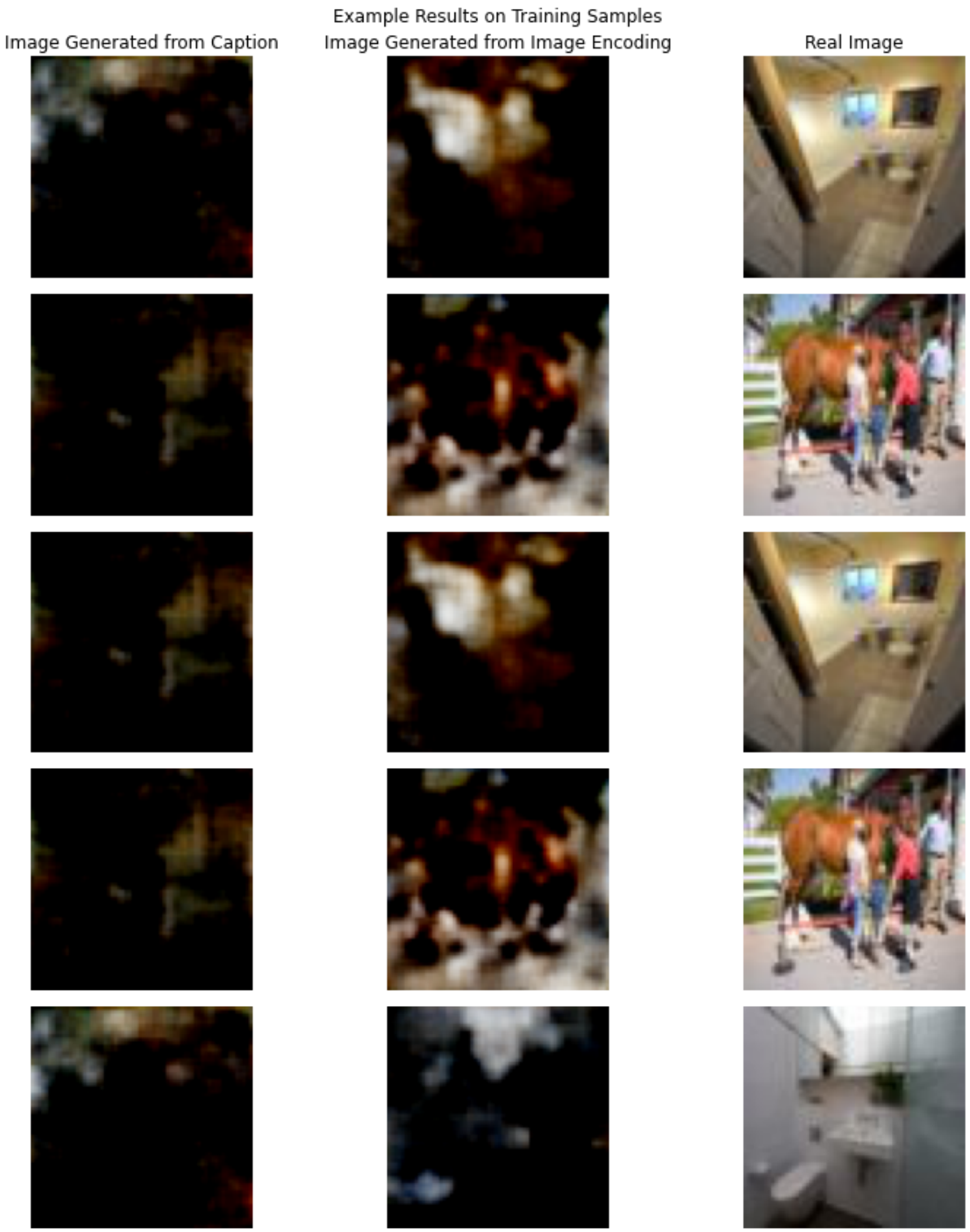

Figure 12. Generated images on training samples

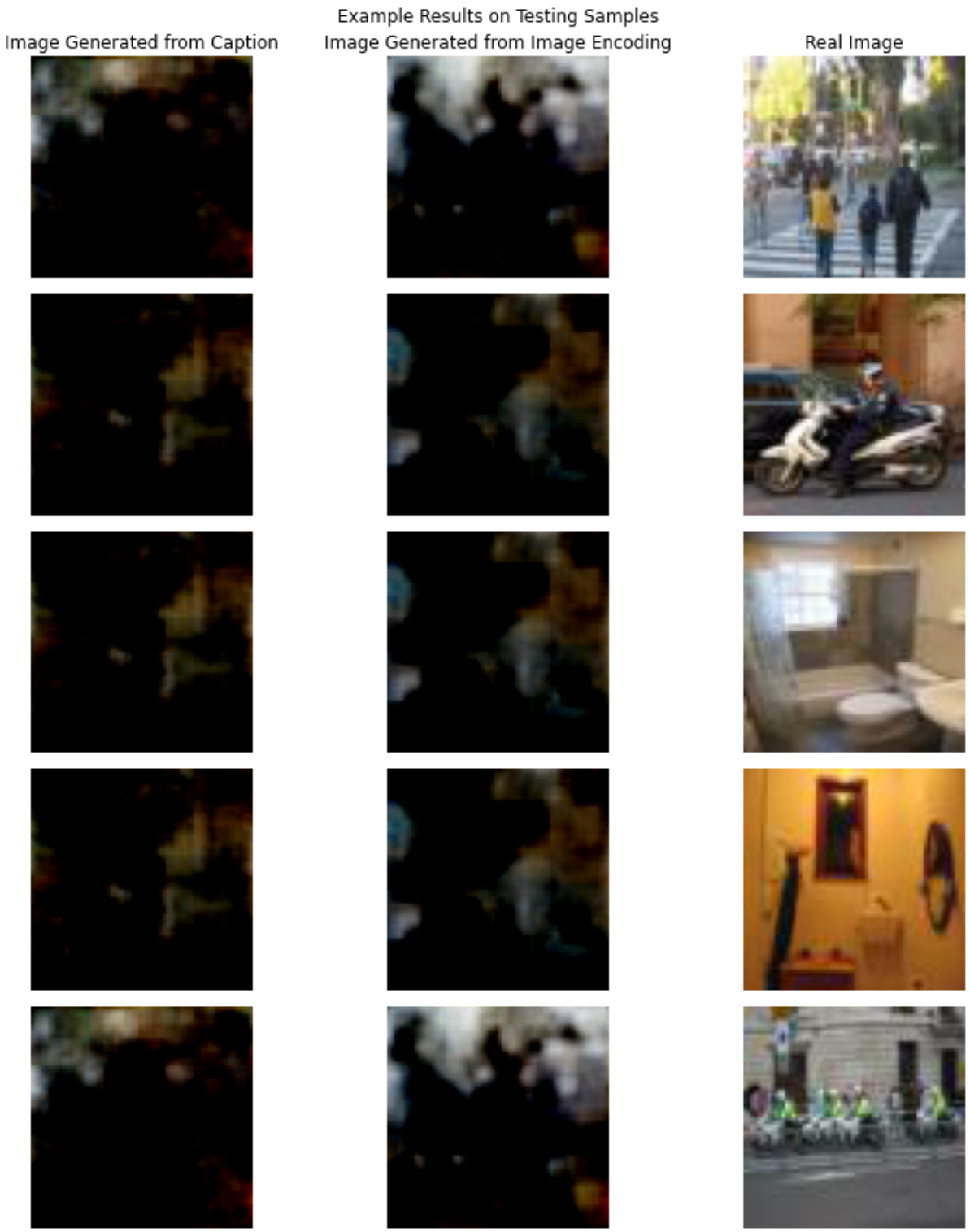

Figure 13. Generated images on test samples

The accuracy of the generated images on test samples is low however, the images generated from the training samples resemble the original images. The horse in the original image can be distinguished in the image that is generated from image encoding.

B. Decoder Only Model

To experiment with a different state-of-the-art architecture, we omit the prior part of our framework in which we transform text encodings to newly generated image encodings. To use a different architecture, we have directly used the decoder for text encodings. They are converted to the images that we see in Figure. The inspiration for this idea comes from the biggest difference between the DALLE-2 and GLIDE project. DALLE-2 uses a prior for its text embeddings whereas GLIDE directly transforms them into images. If images are investigated, we can conclude that generated images might resemble their original but they are different pictures even if we ignore the noise.

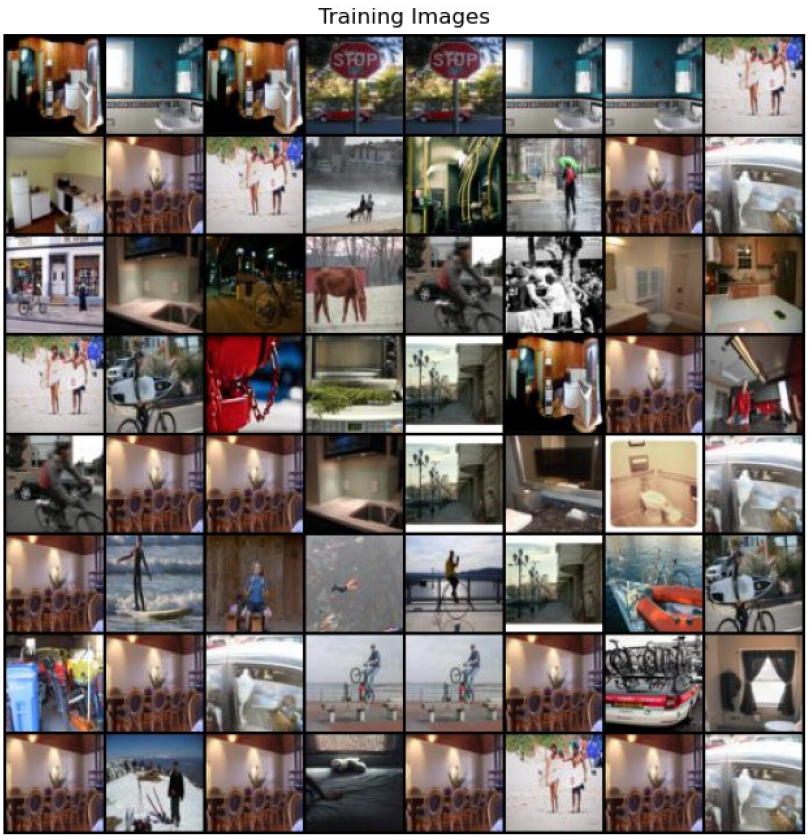

Figure 14. Sample Images Given to the Decoder Only Model

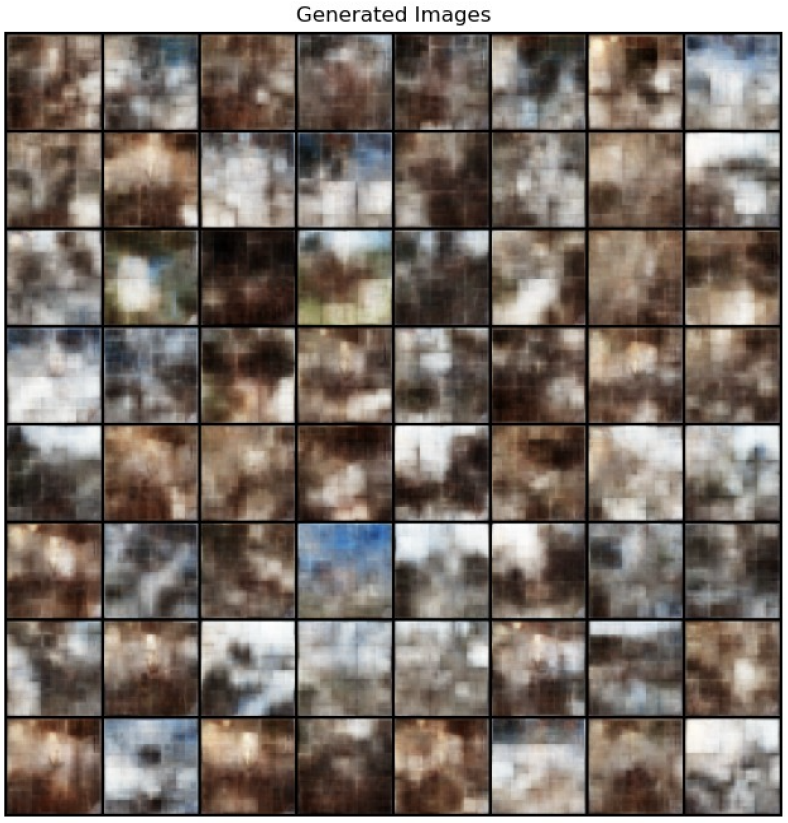

Figure 15. Generated Images From Decoder Only Model

4. Discussion

In our project, we implemented a modified DALLE program which is used for creating images for textual descriptions. We tried to modify and create a novel approach to the DALL-E system according to our understanding of the text-to-image generation problem. We first implemented the text and image encoding parts that are shown in Figure 6 by using the neural network architecture called CLIP. Then, the prior and decoder networks were designed. According to the literature review that we performed, diffusion or GAN models are used for the image generator part of the models. We used a Transformer Model that is presented in Figure 7, and the decoder model that is discussed under the decoder subsection of the methods. Overall, we used a transformer-based model in the prior network without diffusion and a U-Net-like model for the decoder network again without diffusion. We think that the learning performance of the models would have been better with the diffusion models since their working principle is on corrupting the training samples, then applying a reverse corruption process and obtaining the pure samples. We constructed a model and verified the performances of the encoder, prior, and decoder networks separately as can also be seen in Figures 9, 10, and 11 in the results section. We strongly believe that the reason why the overall network did not work as accurately as the models in the literature is the lack of diffusion models in the prior and decoder networks, especially in the prior. As many projects are using ADAM as an optimizer, it has some drawbacks. For instance, Wilson[19] stated that the diversity of the tasks makes ADAM generalize worse than classical stochastic gradient descent with static momentum. Also, a more complex loss function may result in better performance since it may prevent overfitting or underfitting by adjusting the regularization term.

5. References

[1] C. Saharia, W. Chan, S. Saxena, L. Li, J. Whang, E. Denton, S. K. S. Ghasemipour, B. K. Ayan, S. S. Mahdavi, R. G. Lopes, T. Salimans, J. Ho, D. J. Fleet, and M. Norouzi, “Photorealistic text-to-image diffusion models with Deep Language understanding,” *arXiv.org*, 23-May-2022. [Online]. Available: https://doi.org/10.48550/arXiv.2205.11487. [Accessed: 26-May-2022].

[2] P. Dhariwal, A. Nichol, “Diffusion Models Beat GANs on Image Synthesis”, *arXiv.org*, 23-May-2022. [Online]. Available: https://doi.org/10.48550/arXiv.2105.05233. [Accessed: 26-May-2022].

[3] 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*, 12-Jan-2018. [Online]. Available: https://arxiv.org/abs/1706.08500. [Accessed: 26-May-2022].

[4] “Fréchet inception distance,” *Wikipedia*, 02-Apr-2022. [Online]. Available: https://en.wikipedia.org/wiki/Fr%C3%A9chet_inception_distance. [Accessed: 26-May-2022].

[5] P. D. J. Rafid Siddiqui, “Diffusion models made easy,” *Medium*, 17-May-2022. [Online]. Available: https://towardsdatascience.com/diffusion-models-made-easy-8414298ce4da. [Accessed: 26-May-2022].

[6] O. Ronneberger, P. Fischer, and T. Brox, “U-Net: Convolutional Networks for Biomedical Image Segmentation,” *arXiv.org*, 18-May-2015. [Online]. Available: https://arxiv.org/abs/1505.04597. [Accessed: 26-May-2022].

[7] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, and I. Sutskever, “Learning transferable visual models from Natural Language Supervision,” *arXiv.org*, 26-Feb-2021. [Online]. Available: https://arxiv.org/abs/2103.00020. [Accessed: 26-May-2022].

[8] A. Nichol, P. Dhariwal, A. Ramesh, P. Shyam, P. Mishkin, B. McGrew, I. Sutskever, and M. Chen, “Glide: Towards photorealistic image generation and editing with text-guided Diffusion Models,” *arXiv.org*, 08-Mar-2022. [Online]. Available: https://arxiv.org/abs/2112.10741. [Accessed: 26-May-2022].

[9] A. Ramesh, P. Dhariwal, A. Nichol, C. Chu, and M. Chen, “Hierarchical text-conditional image generation with clip latents,” *arXiv.org*, 13-Apr-2022. [Online]. Available: https://arxiv.org/abs/2204.06125. [Accessed: 26-May-2022].

[10] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár, “Microsoft Coco: Common Objects in Context,” *arXiv.org*, 21-Feb-2015. [Online]. Available: https://arxiv.org/abs/1405.0312. [Accessed: 27-May-2022].

[11] J. Skelton, “Generating and editing photorealistic images from text-prompts using OpenAI's glide,” *Paperspace Blog*, 12-Apr-2022. [Online]. Available: https://blog.paperspace.com/glide-image-generation/. [Accessed: 27-May-2022].

[12] R. M. Schmidt, “Recurrent neural networks (rnns): A gentle introduction and overview,” *arXiv.org*, 23-Nov-2019. [Online]. Available: https://arxiv.org/abs/1912.05911. [Accessed: 27-May-2022].

[13] K. Banerjee, V. P. C, R. R. Gupta, K. Vyas, A. H, and B. Mishra, “Exploring alternatives to Softmax function,” *arXiv.org*, 23-Nov-2020. [Online]. Available: https://arxiv.org/abs/2011.11538#:~:text=Softmax%20function%20is%20widely%20used,is%20often%20questioned%20in%20literature. [Accessed: 27-May-2022].

[14] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret, “Transformers are RNNS: Fast autoregressive transformers with linear attention,” *arXiv.org*, 31-Aug-2020. [Online]. Available: https://arxiv.org/abs/2006.16236. [Accessed: 27-May-2022].

[15] U. Ankit, “Transformer neural network: Step-by-step breakdown of the beast.,” *Medium*, 15-Sep-2021. [Online]. Available: https://towardsdatascience.com/transformer-neural-network-step-by-step-breakdown-of-the-beast-b3e096dc857f. [Accessed: 27-May-2022].

[16] *Mean squared error (MSE)*. [Online]. Available: https://www.probabilitycourse.com/chapter9/9_1_5_mean_squared_error_MSE.php. [Accessed: 27-May-2022].

[17] V. Bushaev, “Adam - latest trends in deep learning optimization.,” *Medium*, 24-Oct-2018. [Online]. Available: https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c. [Accessed: 27-May-2022].

[18] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv.org*, 30-Jan-2017. [Online]. Available: https://arxiv.org/abs/1412.6980. [Accessed: 27-May-2022].

[19] A. C. Wilson, R. Roelofs, M. Stern, N. Srebro, and B. Recht, “The marginal value of adaptive gradient methods in machine learning,” *arXiv.org*, 22-May-2018. [Online]. Available: https://arxiv.org/abs/1705.08292. [Accessed: 27-May-2022].

Appendix

encode_train_images.py

```
import io
import torch
import requests
from PIL import Image
import numpy as np
import h5py
import clip

if __name__ == "__main__":
        with h5py.File("data/eee443_project_dataset_train.h5", "r") as f:
      print("Keys: %s" % f.keys())
        train_url = np.array(f["train_url"])
        train_health = np.load("data/healty_train_urls.npy")
        device = "cpu"
        model, preprocessor = clip.load("ViT-B/32", device="cpu")
        IMS_SIZE = train_url.shape[0]

   train_image_features = torch.empty((IMS_SIZE,512), device=device)
        with torch.no_grad():
        for i in range(IMS_SIZE):
        if not train_health[i]:
            train_image_features[i] = torch.zeros(512, device=device)
            continue
        r = requests.get(train_url[i])
        image = Image.open(io.BytesIO(r.content))
        image = preprocessor(image).unsqueeze(0).to(device)
         train_image_features[i] = model.encode_image(image).float()
         print(f"Encoded {i}/{IMS_SIZE} images", end="\r")
   torch.save(train_image_features, "data/train_image_features.pt")
```

encode_test_images.py

```
import io
import torch
import requests
from PIL import Image
import numpy as np
import h5py
import clip

if __name__ == "__main__":
        with h5py.File("data/eee443_project_dataset_test.h5", "r") as f:
```

```
        print("Keys: %s" % f.keys())
        test_url = np.array(f["test_url"])
        test_health = np.load("data/healty_test_urls.npy")
        device = "cpu"
        model, preprocessor = clip.load("ViT-B/32", device="cpu")
        IMS_SIZE = test_url.shape[0]

        test_image_features = torch.empty((IMS_SIZE,512), device=device)
        with torch.no_grad():
        for i in range(IMS_SIZE):
        if not test_health[i]:
            test_image_features[i] = torch.zeros(512, device=device)
                continue
        r = requests.get(test_url[i])
        image = Image.open(io.BytesIO(r.content))
        image = preprocessor(image).unsqueeze(0).to(device)
         test_image_features[i] = model.encode_image(image).float()
        print(f"Encoded {i}/{IMS_SIZE} images", end="\r")
    torch.save(test_image_features, "data/test_image_features.pt")
```

image_loader.py

```
import io
from typing import List
import requests
from PIL import Image
import numpy as np
import h5py

def load_image(url: bytes | List[bytes]) -> List[Image.Image]:
        if isinstance(url, bytes):
        url = [url]
        images = []
        for u in url:
        r = requests.get(u)
        if r.status_code == 200:
        img = Image.open(io.BytesIO(r.content))
        else:
        img = None
       images.append(r.status_code, img)
        return images


def check_urls(urls: np.ndarray) -> np.ndarray:
        """
    Check if the urls are valid
        :param urls:
        :return:
        """
```

```
        healthy_urls = np.zeros(urls.shape, dtype=bool)
        for idx, url in enumerate(urls):
        r = requests.get(url)
        if r.status_code == 200:
        healthy_urls[idx] = True
      print(f"{idx}/{urls.shape[0]}", end="\r")
        return healthy_urls


if __name__ == "__main__":
        with h5py.File("data/eee443_project_dataset_train.h5", "r") as f:
        print("Keys: %s" % f.keys())
          train_url = np.array(f["train_url"])
        with h5py.File("data/eee443_project_dataset_test.h5", "r") as f:
        print("Keys: %s" % f.keys())
        test_url = np.array(f["test_url"])

        healty_train_urls = check_urls(train_url)
        healty_test_urls = check_urls(test_url)
        with open("data/healty_train_urls.npy", "wb") as f:
        np.save(f, healty_train_urls)
        with open("data/healty_test_urls.npy", "wb") as f:
        np.save(f, healty_test_urls)
```

data_generator.py

```
# %% [markdown]
# # NN Project Data Generator

# %% [markdown]
# ### Imports

# %%
import h5py
import torch
import numpy as np
import matplotlib.pyplot as plt
import clip
from PIL import Image
import requests
from importlib import reload
import io
# Import our custom modules
import image_loader
reload(image_loader)

# %% [markdown]
# ### Read Data From H5 Files
```

```
# %%
with h5py.File("data/eee443_project_dataset_train.h5", "r") as f:
        print("Keys: %s" % f.keys())
        train_cap = np.array(f["train_cap"])
        train_imid = np.array(f["train_imid"])
        #train_ims = np.array(f["train_ims"])
        train_url = np.array(f["train_url"])
        word_code = np.array(f["word_code"])
words = np.array(word_code.dtype.names)
word_indices = np.array(list(word_code[0]), dtype=np.int32)
with h5py.File("data/eee443_project_dataset_test.h5", "r") as f:
        print("Keys: %s" % f.keys())
        test_cap = np.array(f["test_caps"])
        test_imid = np.array(f["test_imid"])
        #test_ims = np.array(f["test_ims"])
        test_url = np.array(f["test_url"])
train_N = train_cap.shape[0]
test_N = test_cap.shape[0]
# %% [markdown]
# ### Load CLIP

# %%
device = "cuda" if torch.cuda.is_available() else "cpu"
model, preprocessor = clip.load("ViT-B/32", device=device)

# %% [markdown]
# ### Load or Calculate Tokenized Captions

# %%
try:
   tokenized_train_captions = torch.load("data/tokenized_train_captions.pt",
map_location=device)
except FileNotFoundError:
        ends = np.where(train_cap == 2)[1]
        all_caption =  [""] * train_N
        for i in range(len(train_cap)):
        cap_int = train_cap[i,1:ends[i]]
        cap_int = [cap for cap in cap_int if cap not in [0,1,2,3]]
        cap = " ".join(words[cap_int])
        all_caption[i] = cap
   tokenized_train_captions = clip.tokenize(all_caption).to(device)
   torch.save(tokenized_train_captions, "data/tokenized_train_captions.pt")
try:
   tokenized_test_captions = torch.load("data/tokenized_test_captions.pt",
map_location=device)
except FileNotFoundError:
        ends = np.where(test_cap == 2)[1]
        all_caption =  [""] * test_N
        for i in range(len(test_cap)):
```

```
        cap_int = test_cap[i,1:ends[i]]
        cap_int = [cap for cap in cap_int if cap not in [0,1,2,3]]
        cap = " ".join(words[cap_int])
        all_caption[i] = cap
    tokenized_test_captions = clip.tokenize(all_caption).to(device)
    torch.save(tokenized_test_captions, "data/tokenized_test_captions.pt")

# %% [markdown]
# ### Load or Calculate Text Features

# %%
try:
    encoded_train_captions = torch.load("data/encoded_train_captions.pt",
map_location=device)
except FileNotFoundError:
    encoded_train_captions = torch.empty((train_N,512), device=device)
        TEXT_ENCODE_BATCH = 1000
        with torch.no_grad():
        for i in range(train_N//100):

encoded_train_captions[i*TEXT_ENCODE_BATCH:(i+1)*TEXT_ENCODE_BATCH]
=
model.encode_text(tokenized_train_captions[i*TEXT_ENCODE_BATCH:(i+1)*TEXT
_ENCODE_BATCH]).float()
         print(f"Encoded {i*TEXT_ENCODE_BATCH} captions", end="\r")
    torch.save(encoded_train_captions, "data/encoded_train_captions.pt")
try:
    encoded_test_captions = torch.load("data/encoded_test_captions.pt",
map_location=device)
except FileNotFoundError:
    encoded_test_captions = torch.empty((test_N,512), device=device)
        TEXT_ENCODE_BATCH = 1000
        with torch.no_grad():
        for i in range(test_N//100):

encoded_test_captions[i*TEXT_ENCODE_BATCH:(i+1)*TEXT_ENCODE_BATCH] =
model.encode_text(tokenized_test_captions[i*TEXT_ENCODE_BATCH:(i+1)*TEXT_
ENCODE_BATCH]).float()
        print(f"Encoded {i*TEXT_ENCODE_BATCH} captions", end="\r")
    torch.save(encoded_test_captions, "data/encoded_test_captions.pt")

# %% [markdown]
# ### Remove Tokenized Captions (only needed for caption encoding)

# %%
del tokenized_train_captions, tokenized_test_captions

# %% [markdown]
# ### Load Image Features
```

```
# %%
try:
    test_image_features = torch.load("data/test_image_features.pt")
except FileNotFoundError:
        print("Test Image Fatures Missing")
try:
    train_image_features = torch.load("data/train_image_features.pt")
except FileNotFoundError:
        print("Train Image Fatures Missing")

# %% [markdown]
# ### Load URL Health Masks

# %%
try:
        healty_test_urls = np.load("data/healty_test_urls.npy")
except FileNotFoundError:
        print("Healty Test URLs Missing")
try:
        healty_train_urls = np.load("data/healty_train_urls.npy")
except FileNotFoundError:
        print("Healty Train URLs Missing")


# %% [markdown]
# ### Remove Missing Images from ALL Datasets and Train-Validation Split

# %%
try:
        test_X = np.load("data/test_X.npy")
        test_Y = np.load("data/test_Y.npy")
        train_X = np.load("data/train_X.npy")
        train_Y = np.load("data/train_Y.npy")
        validation_X = np.load("data/validation_X.npy")
        validation_Y = np.load("data/validation_Y.npy")
        encoded_train_X = torch.load("data/encoded_train_X.pt",
map_location=device)
        encoded_test_X = torch.load("data/encoded_test_X.pt",
map_location=device)
    encoded_validation_X = torch.load("data/encoded_validation_X.pt",
map_location=device)
except FileNotFoundError:
        print("Data Missing")
        validation_split = 0.1
    missing_train_url_indices = np.where(healty_train_urls == False)[0]
    missing_test_url_indices = np.where(healty_test_urls == False)[0]
    train_missing_data_mask = np.zeros(train_N, dtype=bool)
    test_missing_data_mask = np.zeros(test_N, dtype=bool)
        for missing_url in missing_train_url_indices:
        train_missing_data_mask[train_imid == missing_url] = True
```

```
        for missing_url in missing_test_url_indices:
    test_missing_data_mask[test_imid == missing_url] = True
        clean_train_cap = train_cap[~train_missing_data_mask]
        clean_train_imid = train_imid[~train_missing_data_mask]
        clean_test_cap = test_cap[~test_missing_data_mask]
        clean_test_imid = test_imid[~test_missing_data_mask]
clean_encoded_train_cap = encoded_train_captions[~train_missing_data_mask]
clean_encoded_test_cap = encoded_test_captions[~test_missing_data_mask]
        clean_test_N = clean_test_cap.shape[0]
        clean_train_N = clean_train_cap.shape[0]
        val_N = int(validation_split * clean_train_N)
        validation_indices = np.random.choice(clean_train_N, val_N, replace=False)
        train_indices = np.setdiff1d(np.arange(clean_train_N), validation_indices)
        train_X = clean_train_cap[train_indices]
        train_Y = clean_train_imid[train_indices]
        validation_X = clean_train_cap[validation_indices]
        validation_Y = clean_train_imid[validation_indices]
        test_X = clean_test_cap
        test_Y = clean_test_imid
        encoded_train_X = clean_encoded_train_cap[train_indices]
encoded_validation_X = clean_encoded_train_cap[validation_indices]
        encoded_test_X = clean_encoded_test_cap
np.save("data/train_X.npy", train_X)
np.save("data/train_Y.npy", train_Y)
np.save("data/validation_X.npy", validation_X)
np.save("data/validation_Y.npy", validation_Y)
np.save("data/test_X.npy", test_X)
np.save("data/test_Y.npy", test_Y)
torch.save(encoded_train_X, "data/encoded_train_X.pt")
torch.save(encoded_validation_X, "data/encoded_validation_X.pt")
torch.save(encoded_test_X, "data/encoded_test_X.pt")
        del clean_train_cap, clean_train_imid, clean_test_cap, clean_test_imid
        del train_missing_data_mask, test_missing_data_mask
        del missing_train_url_indices, missing_test_url_indices
        del train_indices, validation_indices
```

prior_trainer_and_tester.py

```
# %% [markdown]
# # Implementation of The Caption2Image

# %%
import h5py
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
```

```
device = "cuda" if torch.cuda.is_available() else "cpu"
#device = "cpu"
print(f"Using {device} device")

# %%
encoded_train_X = torch.load("data/encoded_train_X.pt", map_location=device)
train_image_features = torch.load("data/train_image_features.pt",
map_location=device)
train_Y_indices = np.load("data/train_Y.npy") - 1
encoded_vadidation_X = torch.load("data/encoded_validation_X.pt",
map_location="cpu")[:1000].to(device)
validation_Y_indices = np.load("data/validation_Y.npy")[:1000] - 1

# %%
transformer_decoder_layer = nn.TransformerDecoderLayer(d_model=512, nhead=8,
batch_first=True, activation="relu", device=device)
prior = torch.nn.TransformerDecoder(transformer_decoder_layer,
num_layers=6).to(device)
MAX_EPOCH = 75
BATCH_SIZE = 2048
LR = 1e-4
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(prior.parameters(), lr=LR)
max_i = list(range(0, encoded_train_X.shape[0], BATCH_SIZE))[-1]
DIFF_SIZE= 10

# %%
for epoch in range(MAX_EPOCH):
        prior.train()
        for i in range(0, encoded_train_X.shape[0], BATCH_SIZE):
        if i == max_i:
        break
        encoded_train_X_batch = encoded_train_X[i:i+BATCH_SIZE][:,None,:]
        train_image_features_batch =
train_image_features[train_Y_indices[i:i+BATCH_SIZE]][:,None,:]
        optimizer.zero_grad()
        out1 = prior(encoded_train_X_batch, train_image_features_batch)
        loss: torch.Tensor = criterion(out1, train_image_features_batch)
        loss.backward()
        optimizer.step()
        print(f"Epoch {epoch}/{MAX_EPOCH} Batch
{i//BATCH_SIZE}/{max_i//BATCH_SIZE} -- Loss: {loss.item():.4f}", end="\r")
        prior.eval()
        with torch.no_grad():
        out = prior(encoded_vadidation_X[:,None,:],
train_image_features[validation_Y_indices][:,None,:])
        loss: torch.Tensor = criterion(out,
train_image_features[validation_Y_indices][:,None,:])
        print(f"End of Epoch {epoch}/{MAX_EPOCH} -- Validation Loss:
{loss.item():.6f}")
```

```
# %%
#torch.save(prior.state_dict(), "prior_backup2")

# %%
nz = 512
ngf = 64
nc = 3
ngpu = 1

class Generator(nn.Module):
        def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
        # input is Z, going into a convolution
        nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0, bias=False),
        nn.BatchNorm2d(ngf * 8),
        nn.ReLU(True),
        # state size. (ngf*8) x 4 x 4
        nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
        nn.BatchNorm2d(ngf * 4),
        nn.ReLU(True),
        # state size. (ngf*4) x 8 x 8
        nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=False),
        nn.BatchNorm2d(ngf * 2),
         nn.ReLU(True),
        # state size. (ngf*2) x 16 x 16
        nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
        nn.BatchNorm2d(ngf),
        nn.ReLU(True),
        # state size. (ngf) x 32 x 32
        nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
        nn.Tanh()
        # state size. (nc) x 64 x 64
        )

        def forward(self, input):
        return self.main(input)

decoder = Generator(ngpu)
decoder.load_state_dict(torch.load("decoder_model"))
docoder = decoder.to(device)
decoder.eval()

# %%

with h5py.File("data/eee443_project_dataset_train.h5", "r") as f:
```

```
        print("Keys: %s" % f.keys())
        train_cap = np.array(f["train_cap"])
        train_imid = np.array(f["train_imid"])
        #train_ims = np.array(f["train_ims"])
        train_url = np.array(f["train_url"])
        word_code = np.array(f["word_code"])
words = np.array(word_code.dtype.names)
word_indices = np.array(list(word_code[0]), dtype=np.int32)
with h5py.File("data/eee443_project_dataset_test.h5", "r") as f:
        print("Keys: %s" % f.keys())
        test_cap = np.array(f["test_caps"])
        test_imid = np.array(f["test_imid"])
        #test_ims = np.array(f["test_ims"])
        test_url = np.array(f["test_url"])
transformer_decoder_layer = nn.TransformerDecoderLayer(d_model=512, nhead=8,
batch_first=True, activation="relu", device=device)
prior = torch.nn.TransformerDecoder(transformer_decoder_layer,
num_layers=6).to(device)
print(prior.load_state_dict(torch.load("prior_backup2")))
train_cap.shape, train_imid.shape, train_url.shape, test_cap.shape, test_imid.shape,
test_url.shape, words.shape

# %%
train_visual_map = {"1": "53315.jpg", "2": "21549.jpg", "3": "53315.jpg", "4":
"21549.jpg", "5": "43078.jpg"}
test_visual_map = {"1": "30770.jpg", "2": "26743.jpg", "3": "27731.jpg", "4":
"8263.jpg", "5": "17549.jpg"}
test_imid[:8], train_imid[:5]

# %%
encoded_test_caption1 = torch.load("data/encoded_test_captions.pt")[:5].to("cpu")
test_image_indices = np.load("data/test_Y.npy")[:5]
test_image_features =
torch.load("data/test_image_features.pt")[test_image_indices].to("cpu")
test_imid[1], test_url[26743], encoded_test_caption1[None,None].shape
#test_out = prior(encoded_test_caption1[None,None],
torch.zeros_like(test_image_features[None,None], device=device) )
test_out = torch.zeros((5,512))
prior.eval()
with torch.no_grad():
        for i in range(5):
        test_out[i,:] = prior(encoded_test_caption1[i,None,:].to(device),
test_image_features[i,None,:].to(device)).to("cpu")
test_out = torch.reshape(test_out, (5,512,1,1))

# %%
test_image_features[:,:, None, None].shape, test_out.shape
test_visual_map[str(i)], i

# %%
```

```
with torch.no_grad():
        images_from_text = decoder(test_out.to(device))
        images_from_encodings = decoder(test_image_features[:,:, None,
None].to(device))
N = images_from_text.shape[0]
plt.figure(figsize=(12,12))
plt.suptitle("Example Results on Testing Samples")
for i in range(N):
        plt.subplot(N,3,i*3+1)
        plt.imshow(images_from_text[i].permute(1,2,0).cpu().detach().numpy())
        if i == 0:
        plt.title("Image Generated from Caption")
        plt.axis("off")
        plt.subplot(N,3,i*3+2)
   plt.imshow(images_from_encodings[i].permute(1,2,0).cpu().detach().numpy())
        if i == 0:
        plt.title("Image Generated from Image Encoding")
        plt.axis("off")
        plt.subplot(N,3,i*3+3)
        real_img = Image.open("display_images/test/"+test_visual_map[str(i+1)])
        plt.imshow(real_img)
        if i == 0:
        plt.title("Real Image")
        plt.axis("off")
        plt.tight_layout()
plt.savefig("results/test_results.png")
plt.show()

# %%
with torch.no_grad():
        train_image_features1 = train_image_features[train_imid[:5]]
        train_caption_encodings = encoded_train_X[:5]
        train_out = torch.zeros((5,512))
        for i in range(5):
        train_out[i,:] = prior(train_caption_encodings[i,None,:].to(device),
train_image_features1[i,None,:].to(device))
        images_from_text_train = decoder(test_out.to(device))
        images_from_encodings_train = decoder(train_image_features1[:,:,
None,None])
N = images_from_text_train.shape[0]
plt.figure(figsize=(12,12))
plt.suptitle("Example Results on Training Samples")
for i in range(N):
        plt.subplot(N,3,i*3+1)
        plt.imshow(images_from_text_train[i].permute(1,2,0).cpu().detach().numpy())
        if i == 0:
        plt.title("Image Generated from Caption")
        plt.axis("off")
        plt.subplot(N,3,i*3+2)
```

```
plt.imshow(images_from_encodings_train[i].permute(1,2,0).cpu().detach().numpy())
        if i == 0:
        plt.title("Image Generated from Image Encoding")
        plt.axis("off")
        plt.subplot(N,3,i*3+3)
        real_img = Image.open("display_images/train/"+train_visual_map[str(i+1)])
        plt.imshow(real_img)
        if i == 0:
        plt.title("Real Image")
        plt.axis("off")
        plt.tight_layout()
plt.savefig("results/train_results.png")
plt.show()
```

decoder_model.py

```
from __future__ import print_function
#%matplotlib inline
import argparse
import os
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
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

# Set random seed for reproducibility
manualSeed = 999
#manualSeed = random.randint(1, 10000) # use if you want new results
print("Random Seed: ", manualSeed)
random.seed(manualSeed)
torch.manual_seed(manualSeed)



# Root directory for dataset
dataroot = "/scratch/ali/proj/proj/train_imgs"
```

```
# Number of workers for dataloader
workers = 2

# Batch size during training
batch_size = 128

# Spatial size of training images. All images will be resized to this
#   size using a transformer.
image_size = 64

# Number of channels in the training images. For color images this is 3
nc = 3

# Size of z latent vector (i.e. size of generator input)
nz = 512

# Size of feature maps in generator
ngf = 64


# Number of training epochs
num_epochs = 500

# Learning rate for optimizers
lr = 0.0002

# Beta1 hyperparam for Adam optimizers
beta1 = 0.5

# Number of GPUs available. Use 0 for CPU mode.
ngpu = 1



train_img_ids = np.load("/scratch/ali/proj/proj/data/train_img_ids.npy")
# Create the dataset
dataset = dset.ImageFolder(root=dataroot,
                           transform=transforms.Compose([
                               transforms.Resize(image_size),
                               transforms.CenterCrop(image_size),
                                   transforms.ToTensor(),
                               transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                                ]))
# Create the dataloader
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                         shuffle=False, num_workers=0)

# Decide which device we want to run on
device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) else "cpu")
```

#Dataset Preprocessing

```
train_img_features = torch.load("/scratch/ali/proj/proj/data/train_image_features.pt",
map_location=device)


tensor = torch.ones((), device=device)
train_x = tensor.new_empty((len(train_img_ids),512,1,1), device=device)
train_index = []

for i, id_ in enumerate(train_img_ids):

        raw_id = dataloader.dataset.samples[i][0]
        id = int(raw_id.translate({ord(letter):  None for letter in
'/scratch/ali/proj/proj/train_imgs/train_images/.jpg'}))
        train_index.append(id)
        train_x[i] = torch.reshape(train_img_features[id], (512,1,1))
```

```
# 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)))
plt.savefig("/scratch/ali/proj/proj/some_train_imgs.png")
```

```
# custom weights initialization called on netG and netD
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)
```

```
# Generator Code

class Generator(nn.Module):
        def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
        # input is Z, going into a convolution
        nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0, bias=False),
        nn.BatchNorm2d(ngf * 8),
        nn.ReLU(True),
        # state size. (ngf*8) x 4 x 4
         nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
        nn.BatchNorm2d(ngf * 4),
        nn.ReLU(True),
        # state size. (ngf*4) x 8 x 8
        nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=False),
        nn.BatchNorm2d(ngf * 2),
        nn.ReLU(True),
        # state size. (ngf*2) x 16 x 16
        nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
        nn.BatchNorm2d(ngf),
        nn.ReLU(True),
        # state size. (ngf) x 32 x 32
        nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
        nn.Tanh()
        # state size. (nc) x 64 x 64
        )

        def forward(self, input):
        return self.main(input)



# Create the generator
netG = Generator(ngpu).to(device)

# Handle multi-gpu if desired
if (device.type == 'cuda') and (ngpu > 1):
        netG = nn.DataParallel(netG, list(range(ngpu)))

# Apply the weights_init function to randomly initialize all weights
#  to mean=0, stdev=0.02.
netG.apply(weights_init)

# Print the model
print(netG)
```

```
criterion = nn.MSELoss()


optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))


# Lists to keep track of progress
img_list = []
G_losses = []
iters = 0




print("Starting Training Loop...")
# For each epoch
for epoch in range(num_epochs):
        # For each batch in the dataloader
        print("Epoch:", epoch)
        for i, data in enumerate(dataloader, 0):

        reconstructed = netG(train_x[i*batch_size:(i+1)*batch_size])
        data_gpu = data[0].to(device)

        loss = criterion(reconstructed, data_gpu)
        optimizerG.zero_grad()
        loss.backward()
        optimizerG.step()

        if i % 50 == 0:
        # Save Losses for plotting later
        loss_np = loss.data.cpu().numpy()
         G_losses.append(loss_np)
        print("Step: ",i)
        print("Loss: ",loss)
       iters += 1




######################################################################
###
# Results
# -------
#
# Finally, lets check out how we did. Here, we will look at three
# different results. First, we will see how D and G's losses changed
```

```
# during training. Second, we will visualize G’s output on the fixed_noise
# batch for every epoch. And third, we will look at a batch of real data
# next to a batch of fake data from G.
#
# **Loss versus training iteration**
#
# Below is a plot of D & G’s losses versus training iterations.
#

"""
plt.figure(figsize=(10,5))
plt.title("Generator and Discriminator Loss During Training")
plt.plot(G_losses.cpu(),label="G")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.savefig("/scratch/ali/proj/proj/loss_plot.png")
"""


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)))
plt.savefig("/scratch/ali/proj/proj/some_train_imgs.png")



reconstructed = netG(train_x[:128])


plt.figure(figsize=(8,8))
plt.axis("off")
plt.title("Generated Images")
plt.imshow(np.transpose(vutils.make_grid(reconstructed.to(device)[:64], padding=2,
normalize=True).cpu(),(1,2,0)))
plt.savefig("/scratch/ali/proj/proj/some_generated_images.png")




torch.save(netG.state_dict(), "/scratch/ali/proj/proj/decoder_model")
#To load the model
netG.load_state_dict(torch.load("/scratch/ali/proj/proj/decoder_model"))
```

decoder_from_text_train.py

```
from __future__ import print_function
#%matplotlib inline
import argparse
import os
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
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
import h5py


# Set random seed for reproducibility
manualSeed = 999
#manualSeed = random.randint(1, 10000) # use if you want new results
print("Random Seed: ", manualSeed)
random.seed(manualSeed)
torch.manual_seed(manualSeed)



# Root directory for dataset
dataroot = "/scratch/ali/proj/proj/train_imgs"

# Number of workers for dataloader
workers = 2

# Batch size during training
batch_size = 128

# Spatial size of training images. All images will be resized to this
#   size using a transformer.
image_size = 64

# Number of channels in the training images. For color images this is 3
```

```
nc = 3

# Size of z latent vector (i.e. size of generator input)
nz = 512

# Size of feature maps in generator
ngf = 64


# Number of training epochs
num_epochs = 250

# Learning rate for optimizers
lr = 0.0002

# Beta1 hyperparam for Adam optimizers
beta1 = 0.5

# Number of GPUs available. Use 0 for CPU mode.
ngpu = 1

with h5py.File("data/eee443_project_dataset_train.h5", "r") as f:
        print("Keys: %s" % f.keys())
        train_cap = np.array(f["train_cap"])
        train_imid = np.array(f["train_imid"])
        #train_ims = np.array(f["train_ims"])
        train_url = np.array(f["train_url"])
        word_code = np.array(f["word_code"])
words = np.array(word_code.dtype.names)
word_indices = np.array(list(word_code[0]), dtype=np.int32)




train_Y_indices = np.load("data/train_Y.npy") - 1

train_img_ids = np.load("/scratch/ali/proj/proj/data/train_img_ids.npy")
# Create the dataset
dataset = dset.ImageFolder(root=dataroot,
                        transform=transforms.Compose([
                            transforms.Resize(image_size),
                            transforms.CenterCrop(image_size),
                            transforms.ToTensor(),
                            transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                        ]))
# Create the dataloader
dataloader = torch.utils.data.DataLoader(dataset, batch_size=len(train_img_ids),
                                 shuffle=False, num_workers=0)

dataloader2 = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
```

```
                                    shuffle=False, num_workers=0)

all_images = next(iter(dataloader))
all_images = all_images[0]

# Decide which device we want to run on
device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) else "cpu")
#Dataset Preprocessing
train_img_features = torch.load("/scratch/ali/proj/proj/data/encoded_train_X.pt",
map_location=device)
step_size = int(len(train_img_features)/batch_size)
tensor = torch.ones((), device=device)
train_x = tensor.new_empty((len(train_img_features),512,1,1), device=device)
train_index = []
train_index_inv = np.zeros((max(train_img_ids)+1), dtype=np.int64)

for i, id_ in enumerate(train_img_features):
        train_x[i] = torch.reshape(train_img_features[i], (512,1,1))

for i, id_ in enumerate(train_img_ids):
        raw_id = dataloader.dataset.samples[i][0]
        id = int(raw_id.translate({ord(letter):  None for letter in
'/scratch/ali/proj/proj/train_imgs/train_images/.jpg'}))
   train_index.append(id)
   train_index_inv[id] = i

"""
# Plot some training images
plt.figure(figsize=(8,8))
plt.axis("off")
plt.title("Training Images")
plt.imshow(np.transpose(vutils.make_grid(all_images.to(device)[:64], padding=2,
normalize=True).cpu(),(1,2,0)))
plt.savefig("/scratch/ali/proj/proj/some_train_imgs.png")
"""

# custom weights initialization called on netG and netD
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)
# Generator Code
class Generator(nn.Module):
        def __init__(self, ngpu):
      super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
```

```
            # input is Z, going into a convolution
             nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0, bias=False),
             nn.BatchNorm2d(ngf * 8),
             nn.ReLU(True),
            # state size. (ngf*8) x 4 x 4
             nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
             nn.BatchNorm2d(ngf * 4),
             nn.ReLU(True),
            # state size. (ngf*4) x 8 x 8
               nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=False),
             nn.BatchNorm2d(ngf * 2),
             nn.ReLU(True),
            # state size. (ngf*2) x 16 x 16
             nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
             nn.ReLU(True),
            # state size. (ngf) x 32 x 32
             nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. (nc) x 64 x 64
            )

            def forward(self, input):
            return self.main(input)



# Create the generator
netG = Generator(ngpu).to(device)

# Handle multi-gpu if desired
if (device.type == 'cuda') and (ngpu > 1):
            netG = nn.DataParallel(netG, list(range(ngpu)))

# Apply the weights_init function to randomly initialize all weights
#  to mean=0, stdev=0.02.
netG.apply(weights_init)

# Print the model
print(netG)
criterion = nn.MSELoss()
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))


# Lists to keep track of progress
img_list = []
G_losses = []
iters = 0

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

```
# For each epoch
for epoch in range(num_epochs):
        # For each batch in the dataloader
    print("Epoch:", epoch)
        for i in range(step_size):
        reconstructed = netG(train_x[i*batch_size:(i+1)*batch_size])
        data_gpu =
all_images[train_index_inv[train_Y_indices[i*batch_size:(i+1)*batch_size]].tolist()].to(
device)

          loss = criterion(reconstructed, data_gpu)
       optimizerG.zero_grad()
          loss.backward()
       optimizerG.step()

          if i % 50 == 0:
          # Save Losses for plotting later
          loss_np = loss.data.cpu().numpy()
           G_losses.append(loss_np)
           print("Step: ",i)
           print("Loss: ",loss)
          iters += 1


loss_arr = np.array(G_losses)
np.save("scartch/ali/proj/proj/data/loss_of_decoder.npy", loss_arr)

reconstructed = netG(train_x[:128])
plt.figure(figsize=(8,8))
plt.axis("off")
plt.title("Generated Images")
plt.imshow(np.transpose(vutils.make_grid(reconstructed.to(device)[:64], padding=2,
normalize=True).cpu(),(1,2,0)))
plt.savefig("/scratch/ali/proj/proj/some_generated_images.png")

plt.figure(figsize=(8,8))
plt.axis("off")
plt.title("Training Images")
plt.imshow(np.transpose(vutils.make_grid(all_images[train_index_inv[train_Y_indice
s[:64]].tolist()].to(device), padding=2, normalize=True).cpu(),(1,2,0)))
plt.savefig("/scratch/ali/proj/proj/some_train_imgs_text_emb.png")


torch.save(netG.state_dict(), "/scratch/ali/proj/proj/decoder_model_text")
#To load the model
netG.load_state_dict(torch.load("/scratch/ali/proj/proj/decoder_model_text"))
```

decoder_from_text.py

```
from __future__ import print_function
#%matplotlib inline
import argparse
import os
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
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
import h5py


# Set random seed for reproducibility
manualSeed = 999
#manualSeed = random.randint(1, 10000) # use if you want new results
print("Random Seed: ", manualSeed)
random.seed(manualSeed)
torch.manual_seed(manualSeed)
# Root directory for dataset
dataroot = "/scratch/ali/proj/proj/train_imgs"

# Number of workers for dataloader
workers = 2

# Batch size during training
batch_size = 128

# Spatial size of training images. All images will be resized to this
#   size using a transformer.
image_size = 64

# Number of channels in the training images. For color images this is 3
nc = 3

# Size of z latent vector (i.e. size of generator input)
```

```
nz = 512

# Size of feature maps in generator
ngf = 64


# Number of training epochs
num_epochs = 250

# Learning rate for optimizers
lr = 0.0002

# Beta1 hyperparam for Adam optimizers
beta1 = 0.5

# Number of GPUs available. Use 0 for CPU mode.
ngpu = 1

with h5py.File("data/eee443_project_dataset_train.h5", "r") as f:
        print("Keys: %s" % f.keys())
        train_cap = np.array(f["train_cap"])
        train_imid = np.array(f["train_imid"])
        #train_ims = np.array(f["train_ims"])
        train_url = np.array(f["train_url"])
        word_code = np.array(f["word_code"])
words = np.array(word_code.dtype.names)
word_indices = np.array(list(word_code[0]), dtype=np.int32)

train_Y_indices = np.load("data/train_Y.npy") - 1

train_img_ids = np.load("/scratch/ali/proj/proj/data/train_img_ids.npy")
# Create the dataset
dataset = dset.ImageFolder(root=dataroot,
                    transform=transforms.Compose([
                        transforms.Resize(image_size),
                        transforms.CenterCrop(image_size),
                        transforms.ToTensor(),
                        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                      ]))
# Create the dataloader
dataloader = torch.utils.data.DataLoader(dataset, batch_size=len(train_img_ids),
                                         shuffle=False, num_workers=0)

dataloader2 = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                         shuffle=False, num_workers=0)

all_images = next(iter(dataloader))
all_images = all_images[0]

# Decide which device we want to run on
```

```
device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) else "cpu")

#Dataset Preprocessing


train_img_features = torch.load("/scratch/ali/proj/proj/data/encoded_train_X.pt",
map_location=device)
step_size = int(len(train_img_features)/batch_size)


tensor = torch.ones((), device=device)
train_x = tensor.new_empty((len(train_img_features),512,1,1), device=device)
train_index = []
train_index_inv = np.zeros((max(train_img_ids)+1), dtype=np.int64)

for i, id_ in enumerate(train_img_features):
        train_x[i] = torch.reshape(train_img_features[i], (512,1,1))

for i, id_ in enumerate(train_img_ids):

        raw_id = dataloader.dataset.samples[i][0]
        id = int(raw_id.translate({ord(letter):  None for letter in
'/scratch/ali/proj/proj/train_imgs/train_images/.jpg'}))
   train_index.append(id)
   train_index_inv[id] = i

"""
# Plot some training images
plt.figure(figsize=(8,8))
plt.axis("off")
plt.title("Training Images")
plt.imshow(np.transpose(vutils.make_grid(all_images.to(device)[:64], padding=2,
normalize=True).cpu(),(1,2,0)))
plt.savefig("/scratch/ali/proj/proj/some_train_imgs.png")
"""


# custom weights initialization called on netG and netD
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)

# Generator Code

class Generator(nn.Module):
        def __init__(self, ngpu):
      super(Generator, self).__init__()
```

```
        self.ngpu = ngpu
        self.main = nn.Sequential(
        # input is Z, going into a convolution
         nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0, bias=False),
         nn.BatchNorm2d(ngf * 8),
         nn.ReLU(True),
        # state size. (ngf*8) x 4 x 4
         nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
         nn.BatchNorm2d(ngf * 4),
         nn.ReLU(True),
        # state size. (ngf*4) x 8 x 8
         nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=False),
         nn.BatchNorm2d(ngf * 2),
         nn.ReLU(True),
        # state size. (ngf*2) x 16 x 16
         nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
         nn.BatchNorm2d(ngf),
         nn.ReLU(True),
        # state size. (ngf) x 32 x 32
         nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
        nn.Tanh()
        # state size. (nc) x 64 x 64
        )

        def forward(self, input):
        return self.main(input)

# Create the generator
netG = Generator(ngpu).to(device)

# Handle multi-gpu if desired
if (device.type == 'cuda') and (ngpu > 1):
        netG = nn.DataParallel(netG, list(range(ngpu)))

# Apply the weights_init function to randomly initialize all weights
#  to mean=0, stdev=0.02.
netG.apply(weights_init)
# Print the model
print(netG)
criterion = nn.MSELoss()
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))

# Lists to keep track of progress
img_list = []
G_losses = []
iters = 0

#To load the trained model
netG.load_state_dict(torch.load("/scratch/ali/proj/proj/decoder_model_text"))
netG.eval()
```

```
#Plot loss curve
loss_train = np.load("data/loss_of_decoder.npy")

loss_curve = []
for i in range(len(loss_train)):
        if i%int((step_size/50)) == 0:
      loss_curve.append(loss_train[i])


plt.plot(loss_curve)
plt.xlabel("Training Epochs")
plt.ylabel("Loss")
plt.savefig("/scratch/ali/proj/proj/train_loss.png")
#Generate images
reconstructed = netG(train_x[:128])

plt.figure(figsize=(8,8))
plt.axis("off")
plt.title("Generated Images")
plt.imshow(np.transpose(vutils.make_grid(reconstructed.to(device)[:64], padding=2,
normalize=True).cpu(),(1,2,0)))
plt.savefig("/scratch/ali/proj/proj/some_generated_images.png")

plt.figure(figsize=(8,8))
plt.axis("off")
plt.title("Training Images")
plt.imshow(np.transpose(vutils.make_grid(all_images[train_index_inv[train_Y_indice
s[:64]].tolist()].to(device), padding=2, normalize=True).cpu(),(1,2,0)))
plt.savefig("/scratch/ali/proj/proj/some_train_imgs_text_emb.png")
```