Term Project: Image Generation Based on Description

Akshay Chitale, Siyuan Yu, and Haoran Lou

CS 6375.001 Spring 2019

Dr. Anjum Chida

5th May 2019

Introduction

One of the most exciting applications of machine learning is text to image conversion, which could be used for a wide variety of applications such as generating the portrait of a criminal based on a description, automatically producing images for stories, and illustrating people's dreams. If such a complicated task could be automated instead of relying on human talent, we could have thumbnail images, for example, for just about anything.

Success has been achieved in the field of text-to-image generation using generative adversarial networks, or GANs. This means that there are two networks: a generator, which produces images based off of text descriptions; and a discriminator, which determines whether an image is real or generated. Some more complicated structures were used in other works, such as a StackGAN [1] or a an AttnGAN [2], which have seen success even for complicated sets such as the COCO dataset, which has hundreds of thousands of captioned images to used for training [3].

Our basic approach more closely resembles the simpler structure of the network from Reed et. al., which has one generator and a discriminator [4]. Both networks are convolutional neural networks, or CNNs, where the generator upscales to an image output and the discriminator has a classification output. We used the COCO dataset [3] and ran our code in the Google Colab environment in order to use GPU processing and to use Keras and Tensorflow [5].

The network did not perform as desired. While some problems were identified and resolved, such as a lack of normalization and the inability to overfit on a small set, large issues such as dying units in the network and similar outputs for different inputs remain. We were also limited by time and compute power, as the COCO dataset is very large.

Problem Definition and Algorithm

Given a vocabulary list V of size 37226 and a set of (image, caption) tuples T for training, our generator is designed to learn which image to generate for a given caption, and the discriminator to tell apart fake images from real ones from the dataset. Specifically, the caption is split into words and stripped of any punctuation to give an array of words up to 250 words long. The words are identified by their position in the vocabulary list.

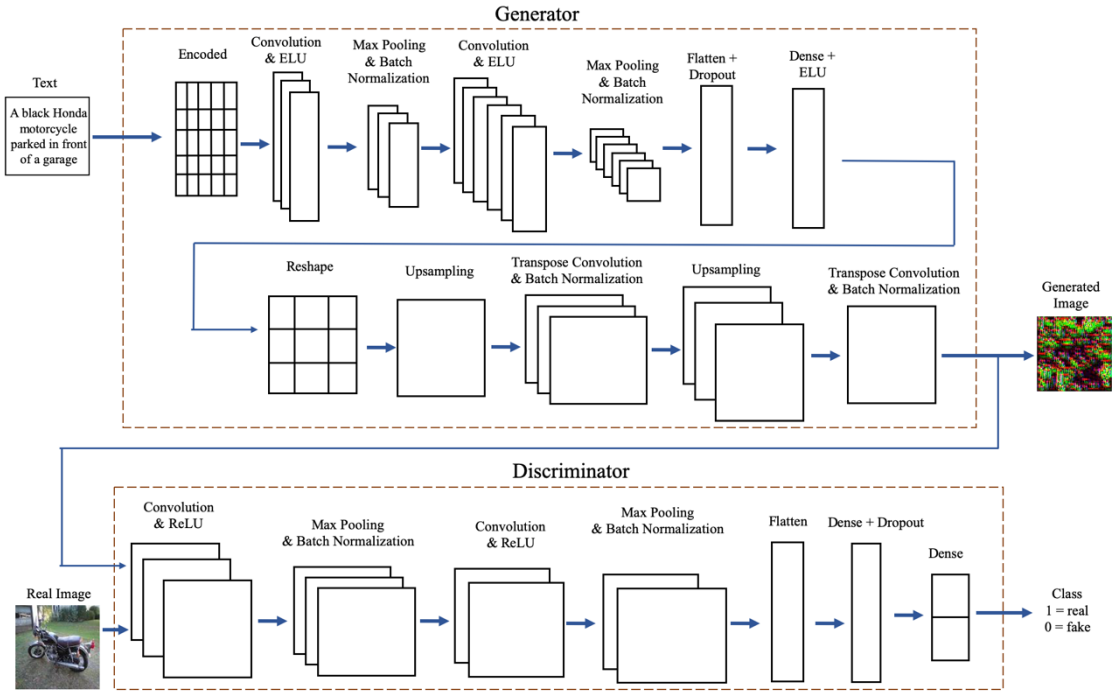
This string of indices is given to the generator as input, which begins with an embedding layer in order to map the words to a continuous space. Then, similar to Reed et. Al. [4], the generator is a deep convolutional network. It has two convolution layers, one dense layer, and two transpose convolutional layers. For efficiency, each convolutional layer is followed by max pooling and batch normalization; each dense layer is preceded by a dropout layer; and each transpose convolutional layer is preceded by an upsampling layer and followed by batch normalization. The output is an RGB (three channel) 128 x 128 pixel image. The loss for the generator was binary cross entropy. The full generator summary is available in Appendix A.

Again like Reed et al. [4], the discriminator is also a deep convolutional network. It takes 128 x 128 pixel RGB images as input. It has two convolutional layers and two dense layers. Again, for efficiency, each convolutional layer is followed by max pooling and batch normalization, and the second dense layer is preceded by a dropout layer. The loss for the discriminator was categorical cross entropy. The full discriminator summary is available in Appendix B.

Our GAN consists only of the generator and the discriminator. The full generator summary is available in Appendix C. For testing example inputs, one simply gives the generator an input string in the

appropriate format and receives the image output based on its prediction. Our entire network structure is shown in Figure 1 below.

Figure 1 (below): Our text-to-image GAN



The training algorithm, similar but modified from Reed et. al., is as follows:

1. **Inputs:** The number of epochs E, the batch size B
2. Get the training list captions T_C and images T_I, the vocabulary set V, and the validation set captions for examples T_{VC}
3. Initialize the generator G and the discriminator D
4. For every epoch e from 1 to E:
    a. For every batch b from 0 to $\lceil |T|/B \rceil$:
        i. Let $b_C = T_C, b_I = T_I$
        ii. Get generator output $b_{gI} = G(b_c)$
        iii. Let the training list for the discriminator be $b_{dI} = b_I + b_{gI}$, where the $+$ symbol is concatenation
        iv. Let the correct classes for the discriminator be $b_{dC} = [1] * B + [0] * B$, where the $*$ symbol means a list of that length and the $+$ symbol is concatenation
        v. Train the discriminator, $D \leftarrow train(D(b_{dI}, b_{dC}))$
        vi. Let the correct classes for the whole GAN be $b_{gC} = [1] * B$, where the $*$ symbol means a list of that length
        vii. With discriminator training disabled, train the GAN, $(G, D) \leftarrow train\left((G, D)(b_c, b_{gC})\right)$
    b. For every caption c in T_{VC}:
        i. Get generator output $i = G(c)$
        ii. Save image i

Experimental Process

Our networks were written in Python3 and use Keras with a Tensorflow backend so that they can run in the Google Colab environment, which is a free GPU platform available for machine learning projects [5]. The networks were trained using the COCO dataset, which, due to its large size of over 590,000 captions, had to be trimmed down from as low as 0.08% its full size to as high as 10% its full size for some runs.

Our method to find better network parameters was to solve issues as they arose, so that we can arrive at a network that overfits on a small set, and then train it on far more data so that it can be more general. Several of the network layers had to be modified, added, or removed over the course of this project to solve some problems faced in training the network.

One of the earlier problems faced was due to a lack of normalization. As shown in Figures 2 and 3 below, the network often learned to always output the maximum value of white or the minimum value of black. This sort of settling to an extreme value can happen if the weights are not normalized, so batch normalization layers were added to address this issue.

Figure 2 (below): Learning the maximum value

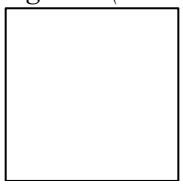
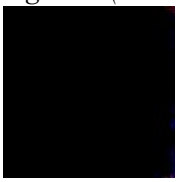


Figure 3 (below): Learning the minimum value



Another issue encountered was that there was little to no change in the generator output, as shown below in Figures 4 and 5 from a run on 10% of the COCO set. By examining the loss of the discriminator, it was observed that the discriminator was actually unable to decrease its loss with each passing epoch, as shown in Figure 5. This was solved by increasing the learning rate so that the network could change between epochs and by redesigning the discriminator to its current design, where it has two convolutional layers instead of the previous four and also has a dropout layer. This means that there are fewer trainable parameters for the discriminator, and it is less likely to get stuck at the minimum loss it was stuck at before. This is what was observed in later runs.

Figure 4 (below): Image at epoch 1

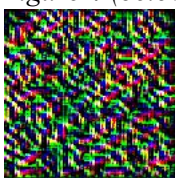


Figure 5 (below): Image at epoch 20

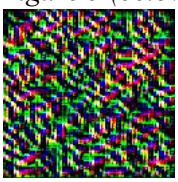
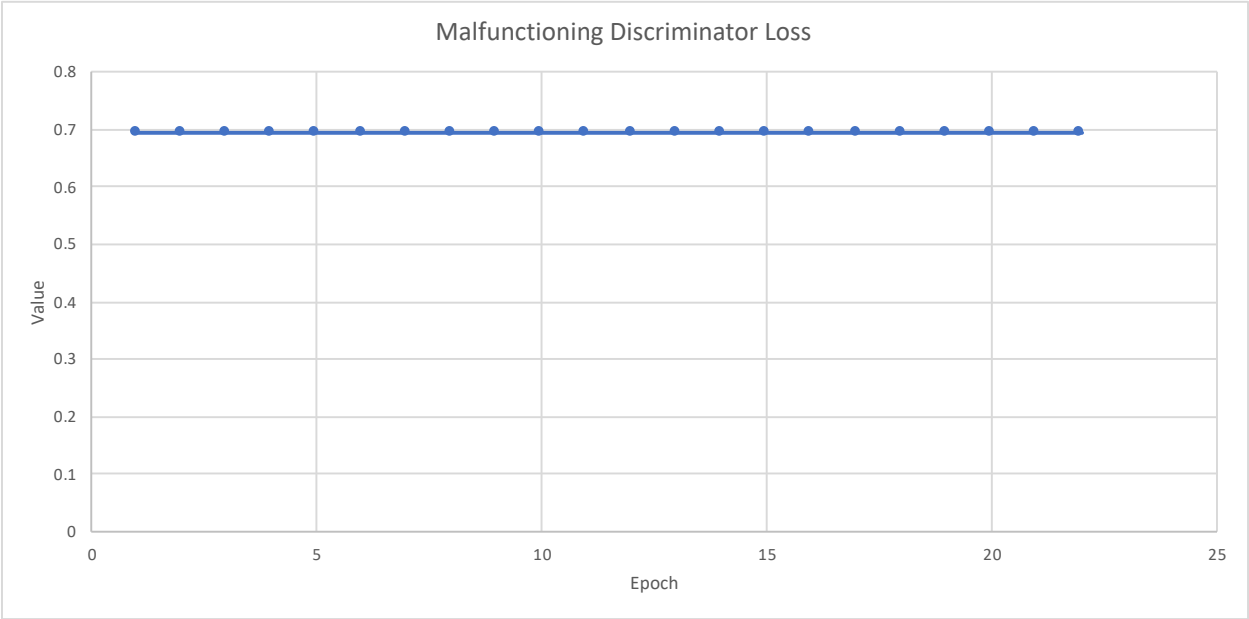


Figure 6 (below): Loss of a malfunctioning discriminator



After roughly twelve different iterations of the network design, including the ones highlighted here, the final network design for this project was reached. This design was chosen for a longer training due to its ability to minimize its loss function and overfit on a small dataset, as shown in Figure 7 for a run on 0.08% of the COCO dataset. The loss also decreases, but not as well, for the entire GAN while training the generator, as shown in Figure 8 for the same small test run. This small run also shows that the model does change over time, as shown in Figures 9 and 10.

Figure 7 (below): Discriminator with decreasing loss

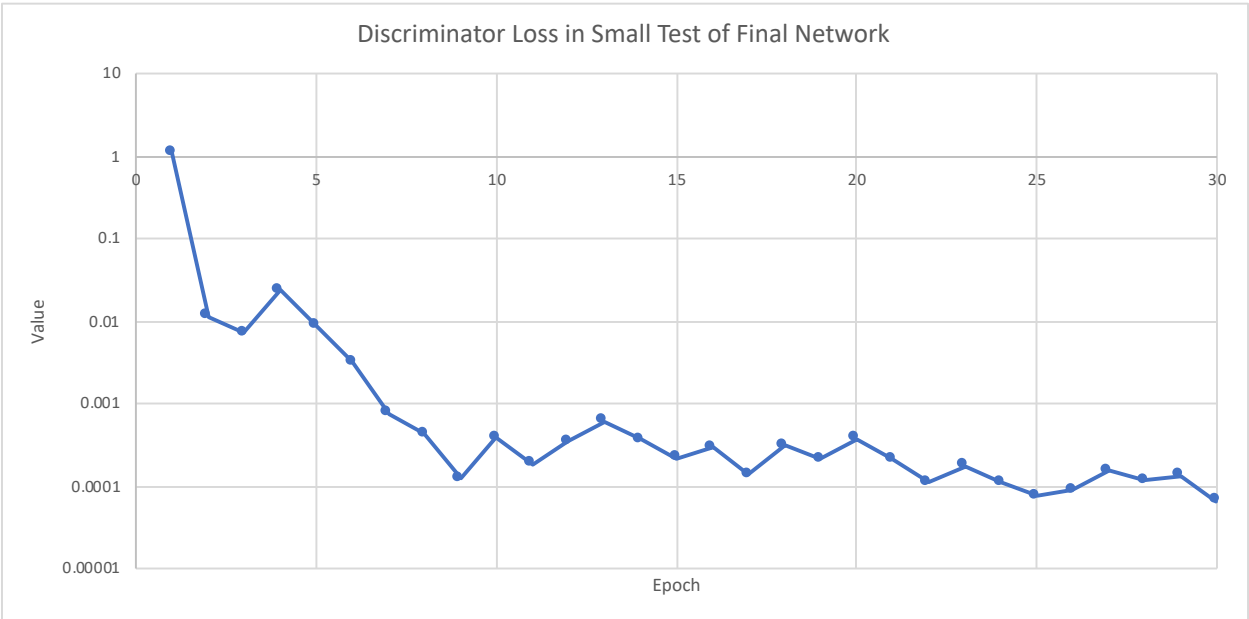


Figure 8 (below): GAN with decreasing loss

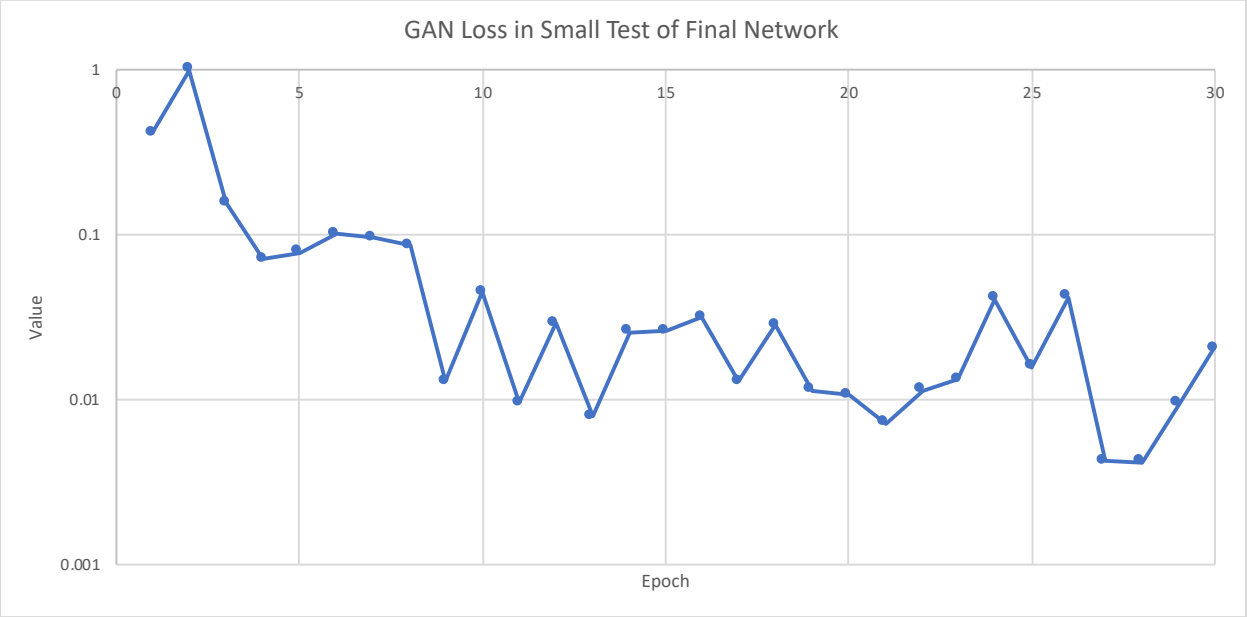


Figure 9 (below): Image at epoch 1

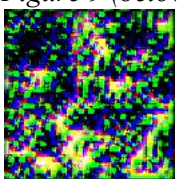
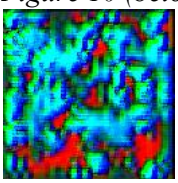


Figure 10 (below): Image at epoch 20



Method and Results

The final training of the network was done with the model described in the Problem Definition section on 1% of the COCO dataset, which is almost 6000 images, for 20 epochs. The reason for the dataset size and number of epochs constraints was largely due to the execution time needed with the compute power available – to run the whole set for 50 epochs, for example, would take several days, for example, before which Google Colab would likely disconnect the runtime instance and discard all of the data.

Below are the plots of the discriminator and GAN loss by epoch in Figures 11 and 12, respectively. Also below in Figure 13 are examples of a test image for the caption “A black Honda motorcycle parked in front of a garage.” from the COCO validation set.

Figure 11 (below): Discriminator loss in final network

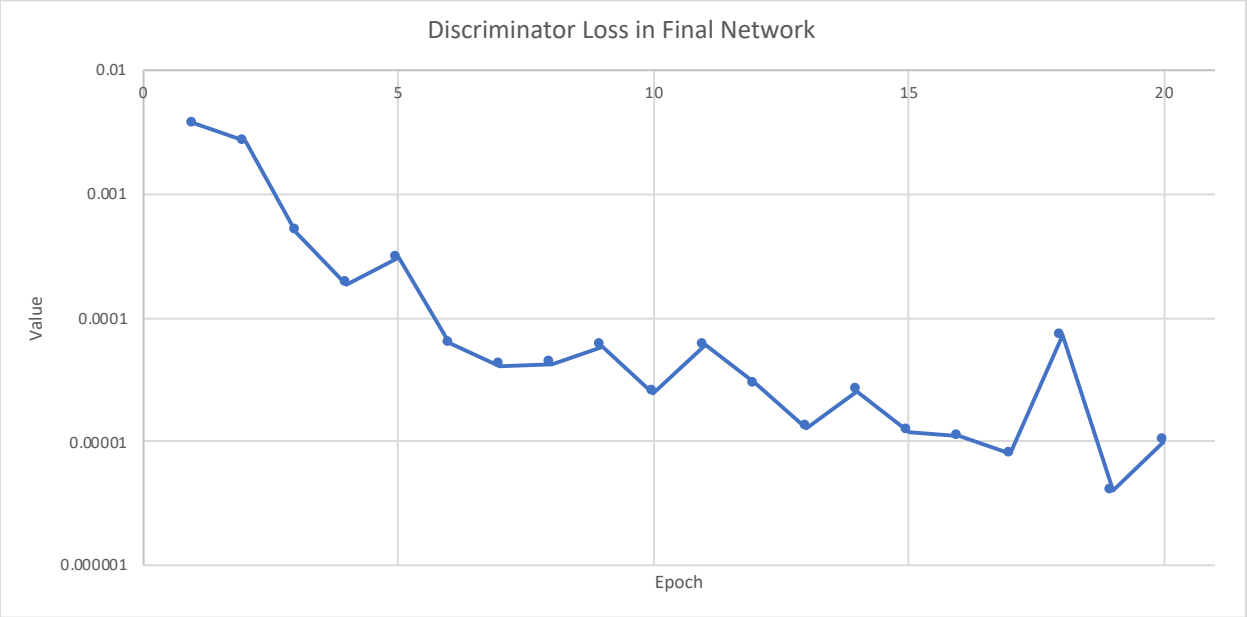


Figure 12 (below): GAN loss in final network

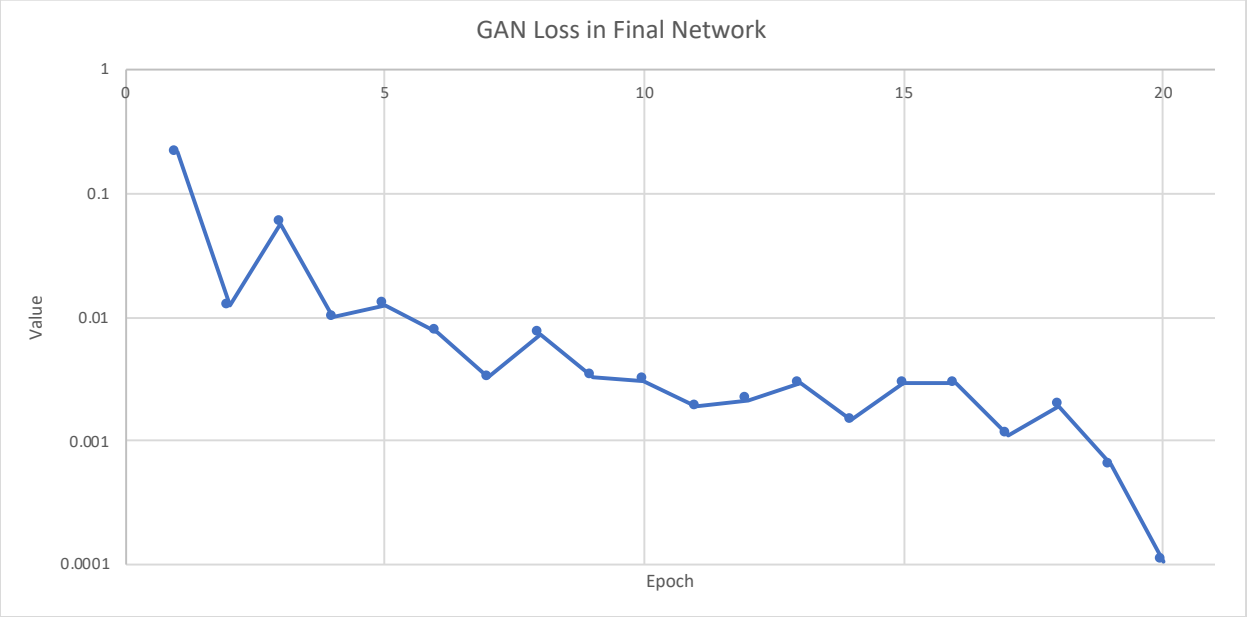
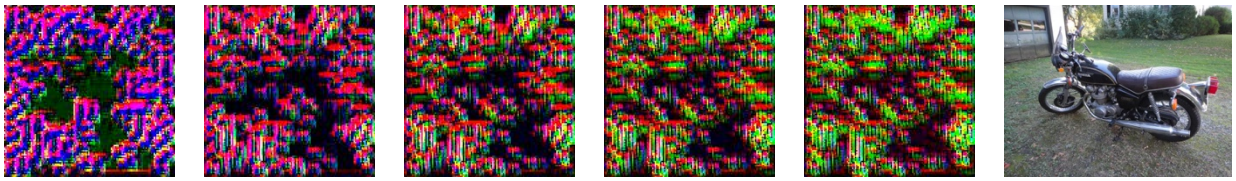


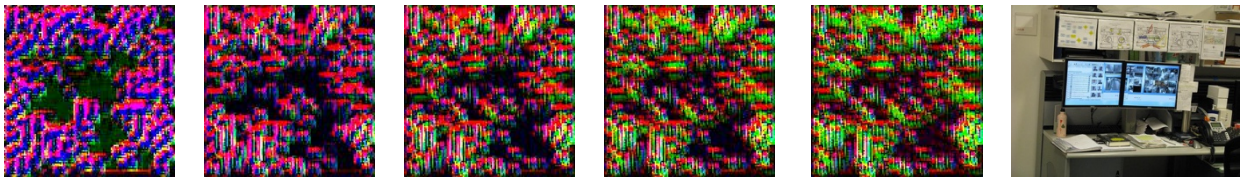
Figure 13 (below): Test results from the final network at epoch 1, 5, 10, 15, and 20 for the phrase "A black Honda motorcycle parked in front of a garage." The last image is the original picture [3].



Discussion

While some of the issues that plague deep CNNs, such as slow learning or a lack of normalization, were overcome, there were several problems that our network was not able to solve. One of these is the issue of vanishing gradients, in which the activation layers in the network die out. Our attempted solution was to replace rectified linear unit activations, or ReLUs, with exponential linear units, or ELUs. The difference is that, while a ReLU is zero for all inputs less than zero, meaning that the gradient is zero, an ELU has a value of $e^x - 1$ for $x < 0$, meaning it has a nonzero gradient of e^x. However, this did not solve the issue, as many of the inputs result in very similar outputs. For example, Figure 14 below shows the output at several epochs for the phrase "An office cubicle with multiple computers in it".

Figure 14 (below): Test results from the final network at epoch 1, 5, 10, 15, and 20 for the phrase "An office cubicle with multiple computers in it." The last image is the original picture [3].



The fundamental problem here could be with the text encoding, since the generator network has to learn a lot of parameters in the very first layer (See Appendix A). If the text encoding is faulty, then it may be that the units in the encoding layer are dying, and that results in a network where the inputs have a very small effect on the output. We did try initializing the encoding layer with a normal distribution with little success.

Related Work

There are a handful of projects that have been done with the same goal of text to image generation through the use of a GAN. The use of a StackGAN by Zhang et. al. is interesting because the structure is actually remarkably similar to the method we pursued [1]. The discriminator and generator are both convolutional, and they both perform downsampling, with the generator upsampling to generate the image [1]. The difference is that there are two of each – hence the term StackGAN; the GANs are stacked on each other to produce higher resolution (256 x 256) results that are more accurate [1].

Another interesting structure to solve the same problem is the AttnGAN, or attentional generative adversarial network [2]. This model uses attention features derived from the text in addition to the word data in order to generate images [2]. It has a lot of fine-tuning capabilities due to features being provided at levels further into the network [2]. This structure has seen successful results even on complex sets such as the COCO dataset [2].

As referred to several times in this report, the GAN from Reed et. al. is the most similar structurally to our network. The main difference is that the network from Reed et. al. adds noise to the input in order to allow the resulting images to be different for a given string of text [3]. This led to results that were remarkable even on varied datasets such as COCO [3].

The other works discussed here achieved much better results than our project did. A potential explanation can be found in a paper from the University of Illinois at Chicago explores mode collapse, a situation in which the generator learns a particular input that is able to fool the discriminator every time [6]. This would be able to explain the results we had in this project, as if the result the generator seems to

always output is sufficient for the discriminator to be fooled, there is no reason that the generator would change. Text to image synthesis can therefore be an unstable process, with many traps that the network could fall into [6].

Future Work

In order to solve the vanishing gradient problem where the input does not depend on the output anymore, we can look at more successful projects for potential modifications. To modify the existing network, we could explore adding noise to the input of the generator as Reed et. al. did [3]. Otherwise, the generator may have to be redesigned in stacks such as the StackGAN [1] or with additional features supplied such as the AttnGAN[2]. Finally, a part of the issue may just be the amount of training performed – by training on the whole COCO dataset for more epochs, a better result may be achieved.

Conclusion

In this project we designed a generative adversarial network (GAN) for text to image generation. Our GAN has two convolutional neural networks. One is the generative network, which tries to generate an image to fool the discriminator, and the discriminator, which tries to determine if an image is real or generated.

Though we were able to resolve some issues during training, such as non-normalized inputs, we were ultimately unable to overcome mode collapse that causes the generator to always output the same single image it knows will trick the discriminator. Future work would be to resolve the issue such that different strings of text result in different outputs.

Bibliography

[1] H. Zhang et al, "StackGAN: Text to photo-realistic image synthesis with stacked generative adversarial networks," in 2017, . DOI: 10.1109/ICCV.2017.629.

[2] T. Xu et al, "AttnGAN: Fine-Grained Text to Image Generation with Attentional Generative Adversarial Networks," 2017.

[3] S. Reed et al, "Generative Adversarial Text to Image Synthesis," 2016.

[4] T. Lin et al, "Microsoft COCO: Common Objects in Context," 2017. Available: http://cocodataset.org/#home. [Accessed 5 May 2019]

[5] Google Colaboratory, "Welcome to Colaboratory!," nd. [Online]. Available: https://colab.research.google.com/notebooks/welcome.ipynb#scrollTo=5fCEDCU_qrC0. [Accessed 5 May 2019].

[6] H. Huang, P. S. Yu and C. Wang, "An Introduction to Image Synthesis with Generative Adversarial Nets," 2018.

Appendix A: Generator Model Summary

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, 250, 4)	148904
reshape_1 (Reshape)	(None, 250, 4, 1)	0
conv2d_1 (Conv2D)	(None, 249, 1, 16)	144
max_pooling2d_1 (MaxPooling2	(None, 83, 1, 16)	0
batch_normalization_1 (Batch	(None, 83, 1, 16)	64
conv2d_2 (Conv2D)	(None, 76, 1, 64)	8256
max_pooling2d_2 (MaxPooling2	(None, 7, 1, 64)	0
batch_normalization_2 (Batch	(None, 7, 1, 64)	256
flatten_1 (Flatten)	(None, 448)	0
dropout_1 (Dropout)	(None, 448)	0
dense_1 (Dense)	(None, 900)	404100
reshape_2 (Reshape)	(None, 30, 30, 1, 1)	0
up_sampling3d_1 (UpSampling3	(None, 60, 60, 1, 1)	0
conv3d_transpose_1 (Conv3DTr	(None, 63, 63, 2, 16)	528
batch_normalization_3 (Batch	(None, 63, 63, 2, 16)	64
up_sampling3d_2 (UpSampling3	(None, 126, 126, 2, 16)	0
conv3d_transpose_2 (Conv3DTr	(None, 128, 128, 3, 1)	289
batch_normalization_4 (Batch	(None, 128, 128, 3, 1)	4

Total params: 562,609
Trainable params: 562,415
Non-trainable params: 194

Appendix B: Discriminator Model Summary

Layer (type)	Output Shape	Param #
reshape_1 (Reshape)	(None, 128, 128, 3, 1)	0
conv3d_1 (Conv3D)	(None, 128, 128, 3, 32)	608
max_pooling3d_1 (MaxPooling3	(None, 32, 32, 3, 32)	0
batch_normalization_1 (Batch	(None, 32, 32, 3, 32)	128
conv3d_2 (Conv3D)	(None, 32, 32, 3, 2)	1154
max_pooling3d_2 (MaxPooling3	(None, 8, 8, 3, 2)	0
batch_normalization_2 (Batch	(None, 8, 8, 3, 2)	8
flatten_1 (Flatten)	(None, 384)	0
dense_1 (Dense)	(None, 128)	49280
dropout_1 (Dropout)	(None, 128)	0
dense_2 (Dense)	(None, 2)	258

Total params: 51,436
Trainable params: 51,368
Non-trainable params: 68

Appendix C: GAN Model Summary

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 250)               0
_________________________________________________________________
sequential_1 (Sequential)    (None, 128, 128, 3, 1)    562609
_________________________________________________________________
sequential_2 (Sequential)    (None, 2)                 51436
=================================================================
Total params: 614,045
Trainable params: 562,415
Non-trainable params: 51,630
_________________________________________________________________
```