Experimenting with GAN structures on a Malware Image Dataset

Austin Chen

***Abstract*—Generative Adversarial Networks (GAN) are a type of generative neural network structure that rely on a generator and discriminator network structure to create samples that imitate the dataset provided. GANs can be applied on top of many neural network types, and in our case we choose to use a GAN with a Deep Convolutional Neural Network to generate malware images.**

I. Introduction

GENERATIVE neural networks are a powerful type of generative structure in their ability generate various types of samples. We choose to apply GANs to a malware image dataset [5] to assess its ability to generate malware image samples. In particular, we select a small subset of this dataset, the malware family Adialer.C, to focus on analyzing the generative capacity of our network towards this malware family.

We seek to achieve several goals with our adversarial networks:

- We want to obtain images that are most similar to the training images
- We want to train our GAN in an efficient manner.
- We want the GAN to converge sooner and remain converged for longer. [4].

II. Preprocessing

Preprocessing consisted of loading the images with PIL, converting the images to numeric pixel values, and resizing the images to a uniform size.

A. Resizing the Image

There were two options identified for resizing the image: The images could be resized to 512×512, which would make developing the convolutional layers of the generator and the discriminator easy. However, as most of the images were 409×512 or 410×512, this meant that every image had around 100 pixels of padding full of zeroes at the bottom. This is a significant feature that could slow down learning as a discriminator might learn to quickly identity images based on that padding at the bottom, and a generator might later also learn that feature, but this would result in no practical features learned. Therefore, this was tested.

Looking at Figure 1, it is clear that the generator has learned the empty space at the bottom, but the clear visual artifacts combined with the misclassification by the discriminator indicates that both models have focused too much on the empty space at the bottom. For example, we can see clear

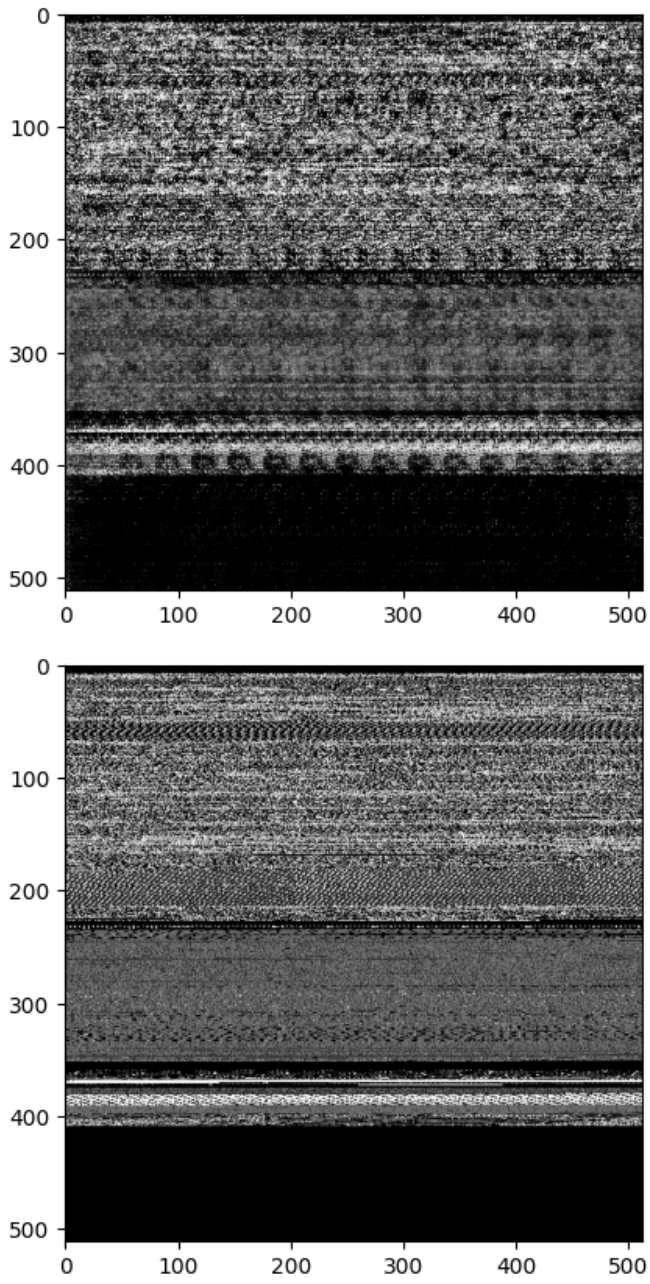

Fig. 1. Generated (Top) vs Real (Bottom) image after 500 epochs. The logit of the top image was 152.46, while the logit of the bottom image was 518.29.

repeating structures at around 400 pixels of the generated image vertically that are not present in the real image.

We can use a different strategy by warping the image to fill a 512×512 space, which does not create new features on its own. Although this changes the original image's features, we can reverse the image transformation to obtain the actual image.

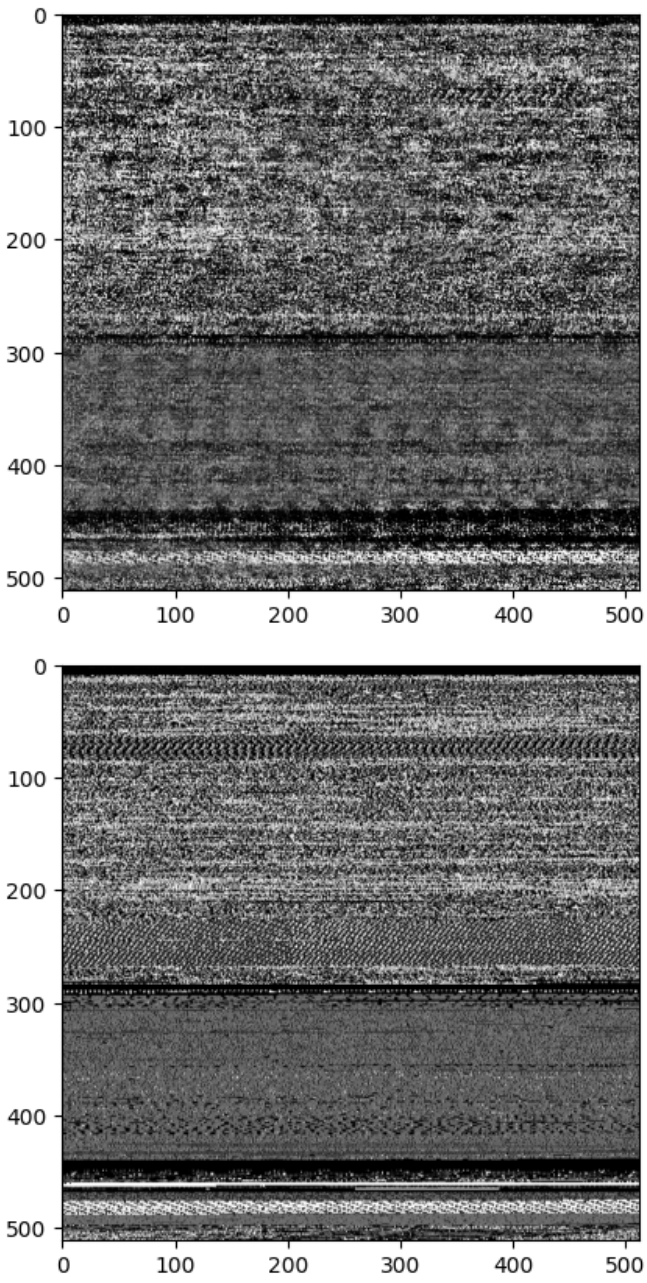

Fig. 2. Generated (Top) vs Real (Bottom) image after 500 epochs. The logit of the top image was 99.99, while the logit of the bottom image was 459.11.

We can see significantly less visual artifacts in the image at the top of Figure 2, meaning that the generator has learned more real features of the malware images. Therefore, we will be using this approach for preprocessing.

III. Training

A. Generator Structure

We seek to compare different generator network structures to see which structure results in more efficient training and greater GAN stability. In particular, we compare two structures, one with more upsampling layers and one with less, according to Figure 3.

```python
def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(8*8*512, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((8, 8, 512)))
    assert model.output_shape == (None, 8, 8, 512)  # Note: None is the batch size

    model.add(layers.Conv2DTranspose(256, (5, 5), strides=(1, 1), padding='same', use_bias=False))
    assert model.output_shape == (None, 8, 8, 256)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    assert model.output_shape == (None, 16, 16, 128)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    assert model.output_shape == (None, 32, 32, 64)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(32, (5, 5), strides=(4, 4), padding='same', use_bias=False))
    assert model.output_shape == (None, 128, 128, 32)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(4, 4), padding='same', use_bias=False, activation='tanh'))
    assert model.output_shape == (None, 512, 512, 1)

    return model
```

```python
def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(32*32*64, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((32, 32, 64)))
    assert model.output_shape == (None, 32, 32, 64)  # Note: None is the batch size

    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(1, 1), padding='same', use_bias=False))
    assert model.output_shape == (None, 32, 32, 64)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(32, (5, 5), strides=(4, 4), padding='same', use_bias=False))
    assert model.output_shape == (None, 128, 128, 32)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(4, 4), padding='same', use_bias=False, activation='tanh'))
    assert model.output_shape == (None, 512, 512, 1)

    return model
```

Fig. 3. Two candidates for generator structure.

We see from the results in Figure 4 that the GAN's performance collapses significantly faster with the generator structure that has less upsampling layers. In particular, the loss spikes are much higher with the generator with less upsampling, indicating that more upsampling layers helps the GAN to be more stable.

B. Alternating Training

We seek to evaluate the effect of alternating training as it is described in Goodfellow's landmark GAN paper [3]. In particular, we will compare the effect of training the discriminator and the generator at the same time compared with training the discriminator and the generator separately.

It is clear that the GAN requires longer to stabilize with alternating training of the generator and discriminator with the loss graph at Figure 5. We can see the GAN stabilize after around 800 epochs, and while the performance of the GAN is not apparent after 1000 epochs, we seek to prioritize efficiency in our GAN results, and therefore alternating training between the generator and discriminator may not be optimal for our purposes.

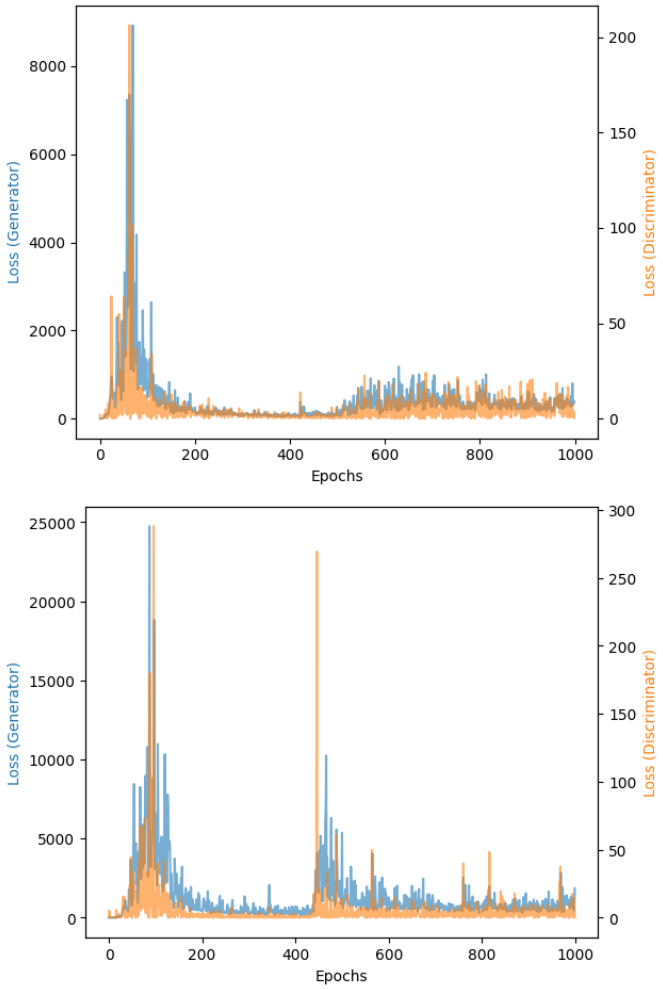

Fig. 4. Loss graph from the generator with more upsampling layers (Top) and more upsampling layers (Bottom)

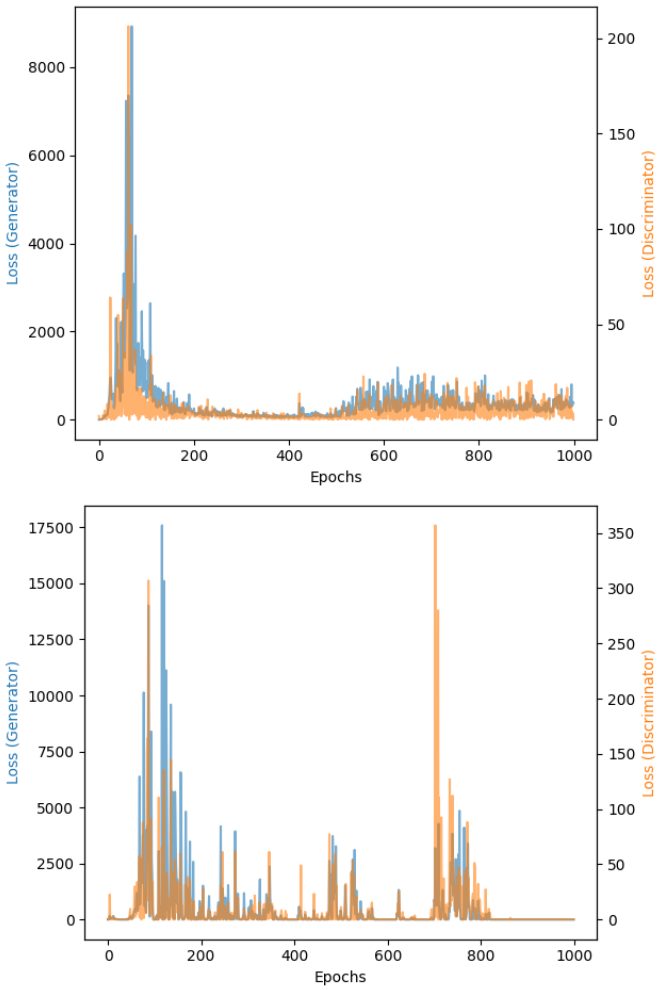

Fig. 5. Loss graph from the generator with simultaneous training (Top) and alternating training (Bottom) of the generator and discriminator.

C. *Wasserstein Loss*

Our current implementation of the loss functions makes use of cross entropy loss, with the loss being a form of cross entropy on both real and generated images for the discriminator and only on generated images for the generator. We seek to evaluate the effect of using Wasserstein Loss [1] instead in the GAN.

Wasserstein loss produced significantly higher quality images that appeared highly similar to the original images. However, the implementation of Wasserstein loss we used caused the loss values to explode, as gradient ascent was performed directly on the logits given by the discriminator, causing the extreme loss values that can be seen in Figure 6. As Wasserstein loss does not differentiate between real and fake outputs in terms of absolute discriminator values, it is impossible to determine the accuracy of the discriminator simply from the discriminator outputs. In other words, while the discriminator used to output negative values for fake outputs and positive values for real outputs with cross entropy loss, this no longer happens with Wasserstein loss.

Analyzing the image generated with Wasserstein loss in Figure 7, we can see significant visual artifacting in the

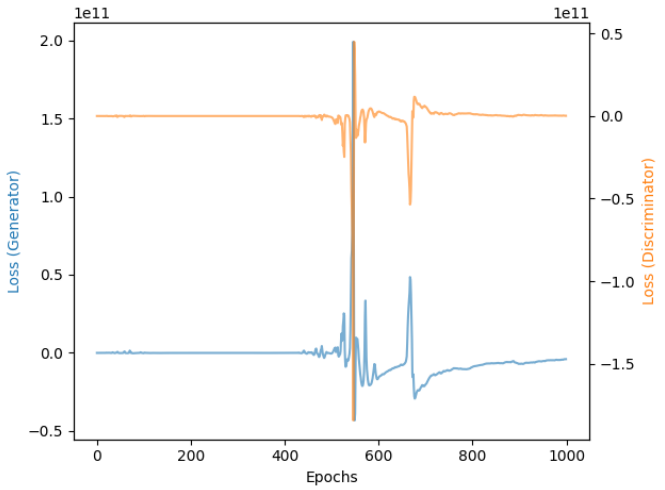

Fig. 6. Loss graph from using Wasserstein loss.

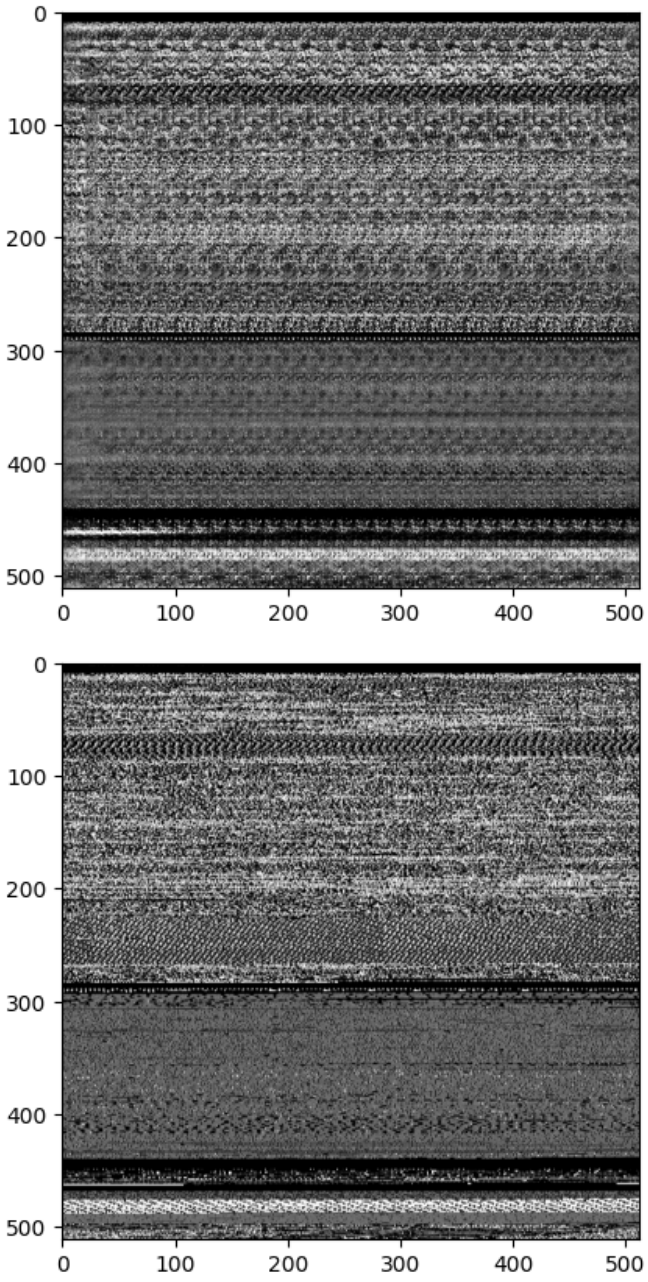

Fig. 7. Generated (Top) vs Real (Bottom) image after 1000 epochs using Wasserstein loss.

repeated structures in the image. However, the image is also able to capture the rippling around 60 pixels in, the small dotted dark band at around 280 pixels, and the dark band around 450 pixels with the light stripe, something previous models were not able to do.

To solve the issue with exploding loss values, the authors identified the Adam optimizer and momentum to be an issue. Following the author's decision to use RMSprop instead of Adam, we compare the loss graphs to see if there is a result.

We can see that the issue of exploding loss values is solved. However, the loss value steadily diverges for the generator, meaning that image quality is significantly lower with the RMSprop optimizer. This is likely due to the lack

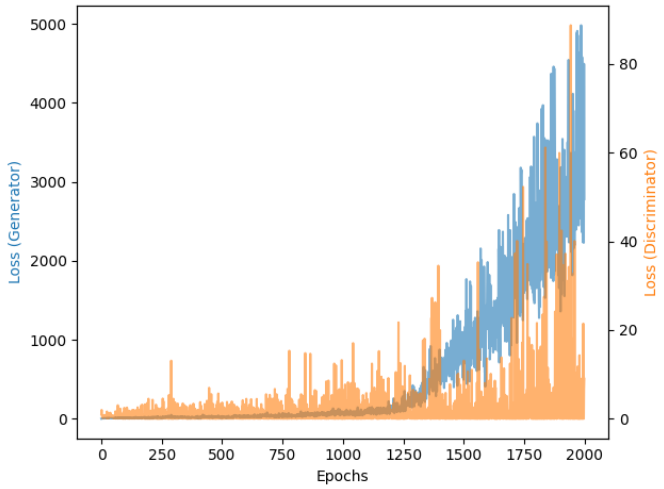

Fig. 8. Wasserstein loss using the RMSprop optimizer.

of momentum reducing the ability of the generator to find a better minima as the learning rate shrinks due to the optimizer. Still, it is apparent that RMSprop does significantly reduce loss values and the exploding gradient problem.

IV. Conclusion

We tested various hyperparameters and their effects on GAN performance on a malware image dataset. Certain changes, such as padding the image, using less upsampling layers, and using alternating training on the generator and the discriminator seemed to worsen the performance of the GAN. On the other hand, using Wasserstein loss resulted in a significant boost to the quality of images generated by the GAN. Finally, we addressed the issue of exploding gradient with respect to Wasserstein loss.

A. Future Work

We did not address the issue of mode collapse in this experiment as we only used a single malware family in training of the GAN. Work can also revolve around combining the effects of multiple hyperparameters to see if there is a significant difference.

References

[1] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan, 2017.

[2] Andrew Brock, Jeff Donahue, and Karen Simonyan. Large scale gan training for high fidelity natural image synthesis, 2019.

[3] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

[4] Naveen Kodali, Jacob Abernethy, James Hays, and Zsolt Kira. On convergence and stability of gans, 2017.

[5] L Nataraj, S Karthikeyan, G Jacob, and B S Manjunath. Malware images: visualization and automatic classification. In *Proceedings of the 8th International Symposium on Visualization for Cyber Security*, number Article 4 in VizSec '11, pages 1–7, New York, NY, USA, July 2011. Association for Computing Machinery.