Wasserstein Generative Adversarial Networks (WGAN)

Andrea Yoss

May 17, 2021

1 Introduction

Generative Adversarial Models (GANs) are powerful deep learning, generative models that were introduced by Goodfellow et al. in 2014[1]. Through an adversarial training process between two neural networks with differing objectives, the network learns to generate data increasingly similar to that of the actual training data. The Wasserstein-GAN (WGAN) was proposed in 2017 by Arjovsky et al. to improve on this architecture. By performing gradient descent on an approximation of the continuous and differentiable Earth-Mover (EM) distance, the WGAN is able to guarantee convergence.

In this paper, I will discuss the traditional GAN architecture, describe the motivation for introducing WGAN, and implement the WGAN algorithm using data from the Fashion-MNIST dataset.

2 Discriminitive vs. Generative Models

A Discriminative model models the decision boundary between the classes. A Generative Model explicitly models the actual distribution of each class. While both models predict the conditional probability, they learn different probabilities in order to do so. A Discriminative model directly learns the conditional probability distribution; on the other hand, a Generative Model learns the joint probability distribution p(x,y), and then ultimately predicting the conditional probability using Bayes Rule.

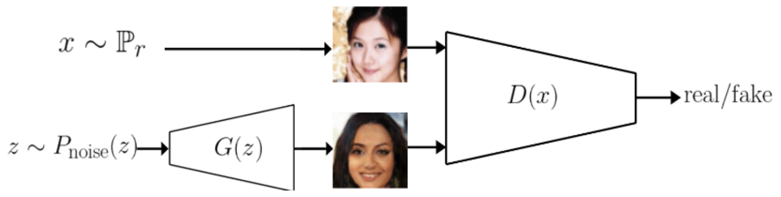
Generative models can create new instances of data from a dataset, while discriminative models actually "discriminate" between classes in a dataset. An example of how these models compare: While a discriminator will be able to learn to tell the difference between the number 7 and the number 9 in the classic MNIST, a generative model will learn the attributes associated with the numbers.

3 Generative Adversarial Networks (GANs)

A Generative Adversarial Network (GAN) is a type of generative model proposed by Goodfellow et al. in 2014[1]. By simultaneously training both a generative model - the generator - and a discriminative model - the discriminator, Goodfellow et al. was able to improve on the performance of prior deep learning generative models by essentially turning the unsupervised learning model into a supervised one.

Both models have different objectives in the training process: the generator seeks to copy the actual underlying distribution of the data, while the discriminator seeks to differentiate between the "real" and generated "fake" data. This "adversarial" process can be viewed as a 2-player minimax, zero-sum game in which the players are pitted against one another. While the discriminator seeks to differentiate between "real" and "fake" data, the generator seeks to generate data similar enough to the actual data in order to "trick" the discriminator into incorrectly labeling the generated data. A graphical representation of the data flow is shown in Figure 1.

Figure 1: Graphical representation of the data flow during the training of a generative adversarial network. Fake images are generated by feeding noise into the generator G(z), where it tries to map the input to the real data distribution, P_r. The real image data (x), which follows some unknown distribution P_r, and the generated fake images are fed into the discriminator D(x). The discriminator classifies the input data as real or fake [3].



3.1 Adversarial Training Process

The adversarial process for a GAN can also be thought of as a minimax game between the discriminator and the generator. During training, the discriminator $D(x;\theta_d)$ is trained to maximize the probability of correctly classifying the data, while the generator $G(z;\theta_g)$ is simultaneously trained to minimize $log(1-D(G(z)))$[1]. This is represented by the value function $V(G,D)$:

$$\min_G \max_D V(D,G) = E_{x\sim p_{data}(x)}[logD(x)] + E_{z\sim p_z(z)}[log(1-D(G(z)))],$$

where $D(x)$ represents the probability that x came from the real training distribution P_{data} as opposed to the generator's distribution P_g, P_z (P_{noise}) is the prior on the input noise variables z, and $G(z)$ represents the mapping to data space. A Nash equilibrium[1] is achieved in this zero-sum game when the generator gets so good that the discriminator can no longer differentiate between the generated data and the real data. At this point, the probability of the discriminator classifying data as "real" or "fake" are both equal to $\frac{1}{2}$. This is essentially the same as random guessing.

3.1.1 Backpropagation

Both networks are modeled as neural networks[2]. During the training process, parameter weights are updated in both networks through backpropagation.

[1]In game theory, a Nash Equilibrium is achieved when no player is better off from changing his or her strategy.

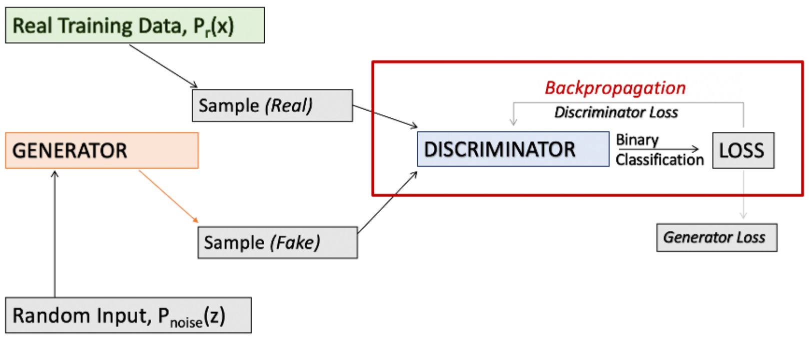
[2]In their original paper (2014), Goodfellow et al. used a multilayer perceptron (MLP) architecture to represent the discriminator and generator models in a GAN. However, in 2016, Radford et al. proposed using convolutional neural networks (CNNs) instead of MLPs in a GAN; they called this variation on Goodfellow's proposed architecture a deep convolutional generative

Backpropagation, or the "backward propagation of errors," is an iterative algorithm for reducing a network's loss by updating its weight parameters based on the contribution of each parameter on that loss. The process works via gradient descent - by computing the gradient of the loss function with respect to each weight, and then backward propagating the errors through the network[4].

In a GAN, parameter weight updates are performed iteratively. While both the generator loss and discriminator loss are calculated by the discriminator and backward propagated through their respective networks, backpropagation only occurs in one network at a time, while the other network's weights are fixed.

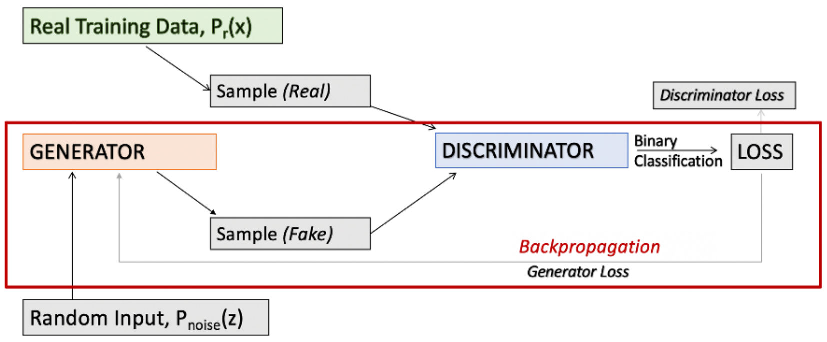
The backpropagation process for the discriminator is summarized in Figure 2.

Figure 2: Backpropagation process for discriminator. (Image by author)



In this stage of the learning process, only the discriminator loss is sent back through the discriminator to obtain its gradients; the generator is not involved in this process. In contrast, when updating the generator, the generator loss is actually backpropagated through both the discriminator and generator to obtain gradients. The process of training the generator over a single iteration is summarized in Figure 3.

Figure 3: Backpropagation process for generator. (Image by author)



While backpropagation is a widely applied method of parameter tuning, its success in generative networks

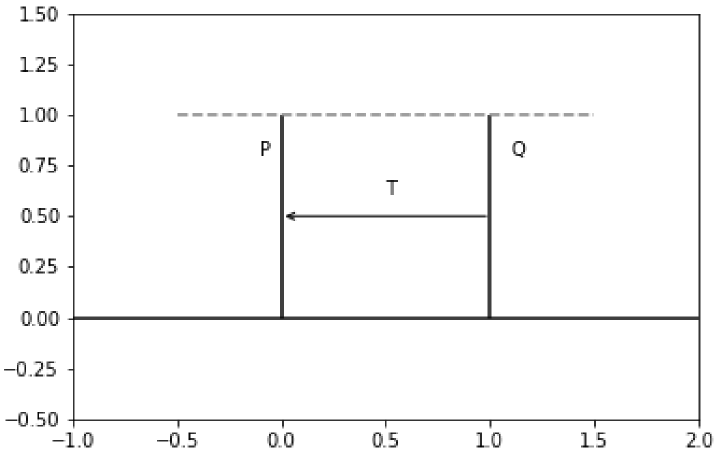
adversarial neural network (DCGAN). DCGANs have been proven to work better than traditional GANs at capturing the data distributions of image datasets such as the MNIST dataset[5].

is very much dependent on the model's loss function, and the similarity metrics at its core.

4 Similarity Metrics

To illustrate the drawbacks of using loss functions with certain similarity metrics when learning distributions supported by low dimensional manifolds, Arjovsky et al. (2017) looked at the simple case of learning two probability distributions parallel to one another in R^2. This scenario is shown in Figure 4.

Figure 4: Visualization of example described in Arjovsky et al. (2017) showing how various similarity metrics behave when provided non-overlapping distributions in low dimensional manifolds. This introduced the appeal of the EM distance compared to the Kullback-Leiber (KL) Divergence and Jensen-Shannon (JS) Divergence due to the continuity exhibited by its resulting loss function. As Q moves T distance approaching the distribution P at 0, Q converges to P under the EM distance. It does not converge under the KL or JS divergences. (Image by author)



When provided two non-overlapping distributions P and Q in low dimensional manifolds, only the EM distance converges to P as Q approached the P distribution at 0. The other metrics behave poorly in this situation.

4.1 Earth Mover Distance

The Earth Mover (EM) Distance[3] is essentially an optimization problem where the optimal value is the infimum[4] of the products of the joint distributions ($\gamma \in \prod(P_r, P_g)$) and the euclidean distance[5] between x and y ($||x - y||$).

[3]The EM Distance is also known as the Wasserstein-1 or Wassersein Distance.

[4]The infimum (inf) is the greatest lower bound.

[5]The euclidean distance is the shortest distance between two points. This is also referred to as the L2 or euclidean norm.

The EM distance formula is

$$W(P_r, P_g) = \inf_{\gamma \in \prod(P_r, P_g)} E_{(x,y)\sim\gamma}[||x - y||],$$

where $\prod(P_r, P_g)$ is the set of all joint distributions $\gamma(x, y)$ with marginal probabilities of P_r and P_g, respectively.

Like the KL and JS divergences, the EM distance measures the similarity between two distinct probability distributions. However, it measures this distance horizontally, as opposed to vertically, which is why it is able to perform well on non-overlapping distributions in low dimensional manifolds.

In the example illustrated in Figure 4, the EM distance will always be equal to T, as it is measured by the horizontal distance between distributions.

4.2 Additional Similarity Metrics

The Kullback-Leiber (KL) divergence between two distributions P_r and P_g is

$$KL(P_r||P_g) = \int log(\frac{P_r(x)}{P_g(x)})P_r(x)d\mu(x).$$

In this example, the KL divergence is $\int log(\frac{P}{Q})Pd\mu(x)$
$\approx log(\frac{0}{1})0 = 0$ when T=0, and
$\approx log(\frac{1}{0})1 \approx \infty$ when T=1.

Next, the Jensen-Shannon (JS) Divergence between two distributions P_r and P_g is

$$JS(P_r, P_g) = KL(P_r||P_m) + KL(P_g||P_m),$$

where P_m is the mixture $\frac{P_r+P_g}{2}$.

In this example, the JS divergence is 0 when T = 1, and is the constant $log2$ when T = 0.

While the JS divergence is symmetrical and always defined, this example proves it is not continuous, and therefore, does not always provide a usable gradient.

This result is significant, as it highlights an issue with training traditional GANs which incorporates an approximation of the JS divergence in its loss function.

Other issues a GAN model may encounter are mode collapse and the vanishing gradient problem. Mode collapse is when the generator's output becomes less diverse. The vanishing gradient problem arises when the discriminator is so good that its gradient essentially "vanishes" preventing the generator from additional learning.

5 Wasserstein Generative Adversarial Networks

In 2017, Arjovsky et al. proposed a variation on the standard GAN they referred to as the Wasserstein Generative Adversarial Network (WGAN) to address the issues encountered when training a traditional GAN[2]. While similar to traditional GANs, there are important distinctions between the WGAN algorithm and that of a traditional GAN. First, instead of minimizing an approximation of the Jensen-Shannon (JS) Divergence, WGAN minimizes an approximation of the Earth Mover (EM) Distance.

Since the EM distance is intractable, Arjovsky et al. introduced a more manageable approximation of the EM distance using the Kantorovich-Rubinstein duality,

$$W(P_r, P_g) = \sup_{||f||_L \leq 1} E_{x \sim P_r}[f(x)] - E_{x \sim P_\theta}[f(x)],$$

where the supremum[6] is taken over all 1-Lipschitz functions.

Instead of having a discriminator network perform binary classification, WGANs use a network that resembles more of a "critic" as it serves as a helper for estimating the EM distance between the distributions,P_r and P_g, by seeking to learn a K-Lipschitz continuous function.

To enforce the Lipschitz continuity constraint during the entirety of the training process, Arjovsky et al. proposed a method called weight clipping[2]. Weight clipping is the process of clamping the weights of a neural network after each gradient update to ensure the parameters lie within a compact space, [-c, c], where c is the fixed clipping parameter.

The WGAN algorithm is summarized in Figure 4.

As opposed to a momentum-based optimizer like Adam, RMSProp is used in the WGAN algorithm. Arjovsky et al. (2017) attribute the issues they encountered when training with momentum-based optimizers to the WGAN having a nonstationary loss function[2].

The critic will train n_critic times for every one iteration the generator trains. In my implementation of the WGAN model, I used the default parameter value of 5 updates of the critic for every one update of the generator.

5.1 Implementation

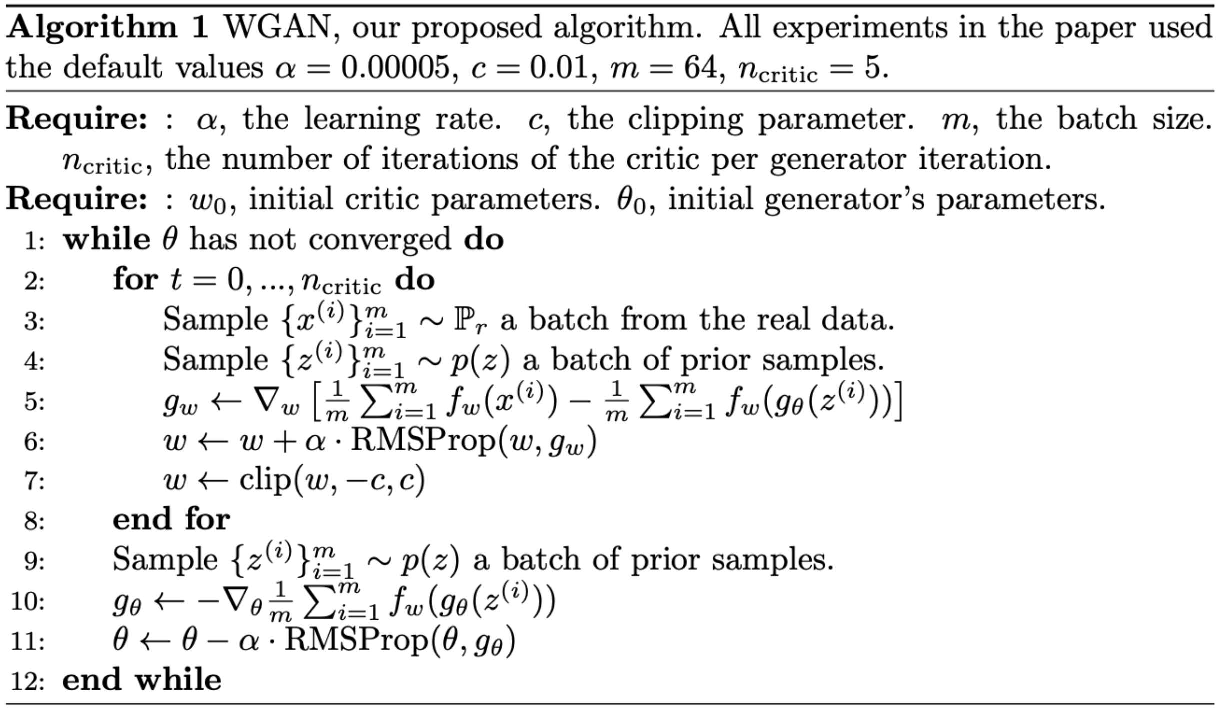
The Fashion-MNIST dataset was created by Zalando Research, and contains 60,000 training and 10,000 test/validation grayscale images, with each image labeled as one of ten types of clothing (such as coat, dress, sneaker, etc.). Sample images for each of the ten classes are displayed in Figure 6.

I trained a random sample of 5000 images from the Fashion-MNIST dataset for 100 epochs on my Wasserstein Generative Adversarial Network.

The complete code for my WGAN can be found in Appendix A of this report, while my code modifications for my implemented WGAN can be found in Appendix B.

[6]The supremum(sup) is the least upper bound.

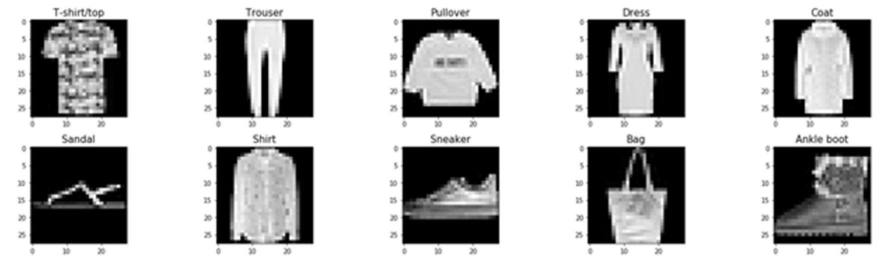
Figure 5: Wasserstein Generative Adversarial Network (WGAN) algorithm [2].



Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

```
Require: : $\alpha$, the learning rate. $c$, the clipping parameter. $m$, the batch size.
    $n_{\text{critic}}$, the number of iterations of the critic per generator iteration.
Require: : $w_0$, initial critic parameters. $\theta_0$, initial generator's parameters.
 1: while $\theta$ has not converged do
 2:     for $t = 0, ..., n_{\text{critic}}$ do
 3:         Sample $\{x^{(i)}\}_{i=1}^{m} \sim \mathbb{P}_r$ a batch from the real data.
 4:         Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
 5:         $g_w \leftarrow \nabla_w \left[\frac{1}{m}\sum_{i=1}^{m} f_w(x^{(i)}) - \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))\right]$
 6:         $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$
 7:         $w \leftarrow \text{clip}(w, -c, c)$
 8:     end for
 9:     Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
10:     $g_\theta \leftarrow -\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))$
11:     $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$
12: end while
```

Figure 6: Sample Images from the Fashion-MNIST Dataset (Image by author)

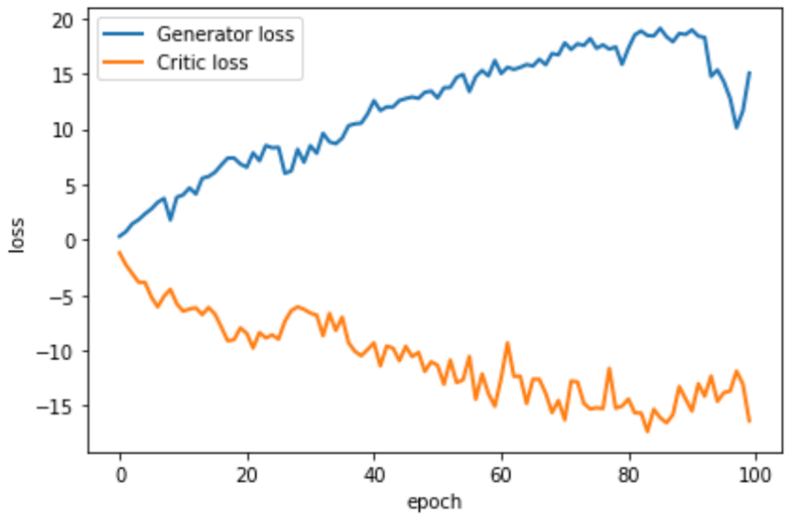


5.1.1 Analysis of Results

My generator and critic losses over 100 epochs is summarized in Figure 7.

After training for 100 epochs, I fed noise through the generator to produce images. Unfortunately these images did not resemble the articles of clothing in the underlying dataset. However, these images in conjunction with with the values of the losses at the end of 100 epochs, highlight the need to train the model over more epochs. Some of the generated images can be found in Appendix C.

Figure 7: Critic and Generator losses throughout the training process. 100 epochs. 5000 samples. (Image by author).



5.2 Limitations

A significant limitation for this project was the availability of computing resources. Ideally, due to the nature of the EM distance, one would train the critic until optimality. Unfortunately, this would require more time and computing power than I have access to. Further, while building my model, I inadvertently exceeded my GPU usage limits on Google Colab. To adapt to no longer having access to a GPU, I reduced the size of my CNN architectures by removing all but two convolutional layers. Instead of using the full Fashion-MNIST dataset, I ultimately chose to use a random sample of 5000 images from the dataset as a proof of concept.

References

[1] Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y. (2014). Generative adversarial networks.

[2] Arjovsky, M., Chintala, S., Bottou, L. (2017). wasserstein gan.

[3] Pieters, M., Wiering, M. (2018). Comparing generative adversarial network techniques for image creation and modification.

[4] Munro, P. (2017). backpropagation. (pp. 93-97). *Springer US.* https://doi.org/10.1007/978-1-4899-7687-1_51

[5] Radford, A., Metz, L., Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks.

Appendix A: Code for WGAN Implementation

This code was used to build the WGAN used to train the Fashion-MNIST data, as described in Section 5.1.

```
#jax imports
import jax.numpy as np
from jax import random, lax, jit, grad, vmap
from jax.experimental import stax, optimizers
from jax.experimental.stax import BatchNorm, Dense, Flatten, Relu,
                                   Tanh, Sigmoid, GeneralConv,
                                   GeneralConvTranspose, Conv,
                                   ConvTranspose, serial,
                                   elementwise
from jax.nn import initializers, leaky_relu, one_hot
from jax.nn.initializers import glorot_normal, normal, ones, zeros
from jax.scipy.special import expit as sigmoid

#additional imports
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import torch
from torchvision import datasets, transforms
import numpy as onp
from tqdm import trange
import itertools
from functools import partial
import time

#defining the architecture of the discriminator network
#similar to the standard GAN, except uses a Dense output layer
##instead of Sigmoid
#using standard conv layer filters: 128, 256, 512, 1024
def Discriminator():
    init_fun, conv_net = stax.serial(Conv(out_chan=128,
                                          filter_shape=(4, 4),
                                          strides=(2, 2),
                                          padding="SAME"),
                                     BatchNorm(),
                                     Relu,
                                     Conv(out_chan=256,
                                          filter_shape=(4, 4),
                                          strides=(2, 2),
                                          padding="SAME"),
                                     BatchNorm(),
                                     Relu,
                                     Conv(out_chan=512,
                                          filter_shape=(4, 4),
                                          strides=(2, 2),
                                                    padding="SAME"),
                                                BatchNorm(),
                                                Relu,
                                                Conv(out_chan=1024,
                                                    filter_shape=(4, 4),
                                                    strides=(2, 2),
                                                    padding="SAME"),
                                                BatchNorm(),
                                                Relu,
                                                Dense(out_dim=1))
    return init_fun, conv_net



#defining the architecture of the Generator Network
#using layer filters: 1024, 512, 256, 128
def Generator():
    init_fun, conv_net = stax.serial(ConvTranspose(out_chan=1024,
                                                    filter_shape=(4, 4),
                                                    strides=(1, 1),
                                                    padding="VALID"),
                                                BatchNorm(),
                                                Relu,
                                                ConvTranspose(out_chan=512,
                                                    filter_shape=(4, 4),
                                                    strides=(2, 2),
                                                    padding="SAME"),
                                                BatchNorm(),
                                                Relu,
                                                ConvTranspose(out_chan=256,
                                                    filter_shape=(4, 4),
                                                    strides=(2, 2)),
                                                    padding="SAME"),
                                                BatchNorm(),
                                                Relu,
                                                ConvTranspose(out_chan=128,
                                                    filter_shape=(4, 4),
                                                    strides=(2, 2),
                                                    padding="SAME"),
                                                BatchNorm(),
                                                Relu,
                                                ConvTranspose(out_chan=1,
                                                    filter_shape=(4, 4),
                                                    strides=(2, 2),
                                                    padding="SAME"),
                                                Tanh)

    return init_fun, conv_net
```

```
class WGAN:
    # Initialize the class
    def __init__(self, init_key):

      # Set up network initialization and evaluation functions
        self.G_init, self.G_apply = Generator()
        self.D_init, self.D_apply = Discriminator()

        #from paper
        #weight clipping
        self.c = 0.01
        #learning rate
        self.alpha = 0.00005
        #batch size
        self.batch_size = 64
        #iterations of critic before 1 generator
        self.n_critic = 5

        # Initialize parameters, not committing to a batch shape
        k1, k2 = random.split(init_key, 2)
        _, G_params = self.G_init(k1, (-1, 1, 1, 100))
        _, D_params = self.D_init(k2, (-1, 64, 64, 1))

        # Use optimizers to set optimizer initialization and update functions
        self.G_opt_init, \
        self.G_opt_update, \
        self.G_get_params = optimizers.rmsprop(step_size = self.alpha)
        self.G_opt_state = self.G_opt_init(G_params)

        self.D_opt_init, \
        self.D_opt_update, \
        self.D_get_params = optimizers.rmsprop(step_size = self.alpha)
        self.D_opt_state = self.D_opt_init(D_params)

        # Logger
        self.G_itercount = itertools.count()
        self.D_itercount = itertools.count()
        self.batch_count = itertools.count()

        #removed log
        self.G_loss = []
        self.D_loss = []


    @partial(jit, static_argnums=(0,))
    def loss_G(self, G_params, D_params, batch):
        _, Z = batch
        # Generate fake image
        fake_image = self.G_apply(G_params, Z)
        # Evaluate discriminator output
        fake_logits = self.D_apply(D_params, fake_image)
        #loss function for generator
        # removed negative sign
        loss = onp.mean(fake_logits)
        return loss

    @partial(jit, static_argnums=(0,))
    def loss_D(self, G_params, D_params, batch):
        X, Z = batch

        # Generate fake image
        fake_image = self.G_apply(G_params, Z)

        # Evaluate discriminator output for real and fake images
        real_logits = self.D_apply(D_params, X)
        fake_logits = self.D_apply(D_params, fake_image)

        # Compute Wasserstein Loss = output of D loss
        wasserstein_loss = onp.mean(real_logits) - onp.mean(fake_logits)
        return wasserstein_loss


    # Define a compiled update step for generator
    @partial(jit, static_argnums=(0,))
    def G_step(self, i, G_opt_state, D_opt_state, batch):
        G_params = self.G_get_params(G_opt_state)
        D_params = self.D_get_params(D_opt_state)
        #gradient of loss function
        g = grad(self.loss_G, 0)(G_params, D_params, batch)
        return self.G_opt_update(i, g, G_opt_state)


    # Define a compiled update step for discriminator/ critic
    #clipping params at c = 0.01
    #using clip_grads() from jax.experimental.optimizers
    @partial(jit, static_argnums=(0,))
    def D_step(self, i, G_opt_state, D_opt_state, batch):
        G_params = self.G_get_params(G_opt_state)
        D_params = self.D_get_params(D_opt_state)
        g = grad(self.loss_D, 1)(G_params, D_params, batch)
        #clipping gradient before updating discriminator params
        g_clip = optimizers.clip_grads(g, self.c)
        return self.D_opt_update(i, g_clip, D_opt_state)
```

```
#getting batches of real data X and noise data Z
#reshaping X to correct dimensions and channel (1, greyscale)
@partial(jit, static_argnums=(0,))
def fetch_batch(self, rng_key, inputs):
    batch_size = inputs.shape[0]
    X = inputs.reshape(batch_size, 64, 64, 1)
    Z = random.normal(rng_key, (batch_size, 1, 1, 100))
    return X, Z

# Optimize parameters in a loop
def train(self, train_loader, num_epochs = 5):

    pbar = trange(num_epochs)

    for epoch in pbar:
      print("")
      #update critic params n_critic times
      for t in range(self.n_critic):
        print("Critic Update: {}".format(t))

        for batch_idx, (inputs, _) in enumerate(train_loader):

            key = random.PRNGKey(next(self.batch_count))


            # Update discriminator (critic)
            batch = self.fetch_batch(key, np.array(inputs))
            self.D_opt_state = self.D_step(next(self.D_itercount),
                                            self.G_opt_state,
                                            self.D_opt_state,
                                            batch)
            #print("Critic Loss for update {}: {}".format(t,
                self.loss_D(G_params, D_params, batch)))

      # Update generator params 1x
      for batch_idx, (inputs, _) in enumerate(train_loader):

          key = random.PRNGKey(next(self.batch_count))

          batch = self.fetch_batch(key, np.array(inputs))
          self.G_opt_state = self.G_step(next(self.G_itercount),
                                            self.G_opt_state,
                                            self.D_opt_state,
                                            batch)
      # Logger
      G_params = self.G_get_params(self.G_opt_state)
      D_params = self.D_get_params(self.D_opt_state)
```

```
            loss_G = self.loss_G(G_params, D_params, batch)
            loss_D = self.loss_D(G_params, D_params, batch)

            self.G_loss.append(loss_G)
            self.D_loss.append(loss_D)
            pbar.set_postfix({'Generator␣loss': loss_G, 'Critic␣loss': loss_D})

    @partial(jit, static_argnums=(0,))
    def generate(self, rng_key, G_params):
        Z = random.normal(rng_key, (1, 1, 1, 100))
        fake_image = self.G_apply(G_params, Z)
        return fake_image
```

Code for downloading the Fashion-MNIST dataset using datasets from torchvision module and converting the data to dataloaders, which divided up our training data into mini-batches of 64 images per batch.

```
#Data Import
batch_size = 64

#to normalize, resize, transform to tensor
transform = transforms.Compose([transforms.Resize(64),
                                 transforms.ToTensor(),
                                transforms.Normalize(mean=(0.5), std = (0.5))])

#importing fashion mnist dataset from torchvision.datasets
#train, test data
train_dataset = datasets.FashionMNIST('traindata', train=True, download = True, transform=
test_dataset = datasets.FashionMNIST('testdata', train=False, download = True, transform=t

#Data Loader
train_loader = torch.utils.data.DataLoader(train_dataset,
                                           batch_size = batch_size,
                                           shuffle = True)

test_loader = torch.utils.data.DataLoader(train_dataset,
                                          batch_size = 1,
                                          shuffle = False)
```

This is the code to initialize and train the model on the dataloader.

```
# initializing model
init_key = random.PRNGKey(0)
model = WGAN(init_key)

# training model
model.train(train_loader, num_epochs = epochs)
```

Appendix B: Code Modifications for Proof of Concept

This code contains the added and modified functions used to train my WGAN model, on a smaller scale, primarily to the limitations introduced in Section 5.2.

Modifications to the Generator/ Discriminator Architectures

```
def Discriminator_mod():
    init_fun, conv_net = stax.serial(Conv(out_chan=256,
                                         filter_shape=(4, 4),
                                         strides=(2, 2),
                                         padding="SAME"),
                                    BatchNorm(),
                                    Relu,
                                    Conv(out_chan=512,
                                         filter_shape=(4, 4),
                                         strides=(2, 2),
                                         padding="SAME"),
                                    BatchNorm(),
                                    Relu,
                                    Dense(out_dim=1))
    return init_fun, conv_net


def Generator_mod():
    init_fun, conv_net = stax.serial(ConvTranspose(out_chan=512,
                                         filter_shape=(4, 4),
                                         strides=(2, 2),
                                         padding="SAME"),
                                    BatchNorm(),
                                    Relu,
                                    ConvTranspose(out_chan=256,
                                         filter_shape=(4, 4),
                                         strides=(2, 2)),
                                         padding="SAME"),
                                    BatchNorm(),
                                    Relu,
                                    ConvTranspose(out_chan=1,
                                         filter_shape=(4, 4),
                                         strides=(2, 2),
                                         padding="SAME"),
                                    Tanh)

    return init_fun, conv_net
```

Code to Obtain Random Sample from Fashion-MNIST Dataset and Create Dataloader using Sample

```
#sampling 5000 images from dataset to address computational limitations
from torch.utils.data import Dataset, DataLoader, SubsetRandomSampler

dataset_size = len(train_dataset)
dataset_indices = list(range(dataset_size))
onp.random.shuffle(dataset_indices)
train_idx = dataset_indices[:5000]
train_sampler = SubsetRandomSampler(train_idx)
mod_train_loader = DataLoader(dataset=train_dataset,
                              shuffle=False, batch_size=64, sampler=train_sampler)
```

Appendix C: Images Produced by Generator at the End of Training, 100 epochs

After feeding noise into the generator, it generated the following images. Please note that the underlying real distribution P_r was images of articles of clothing.

Figure 8: Generated Image 1. (Image by author)

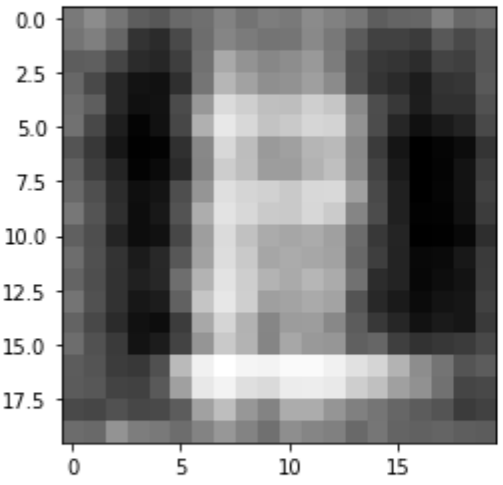


Figure 9: Generated Image 2. (Image by author)

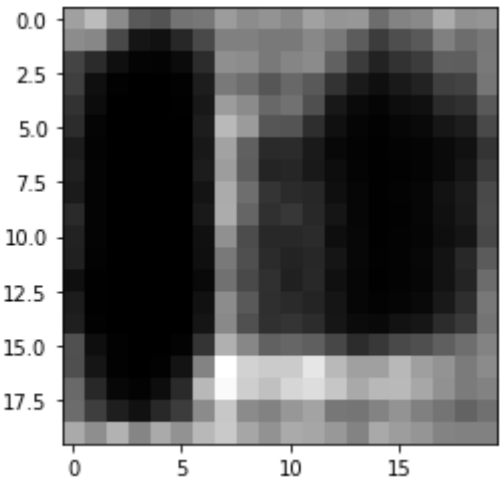
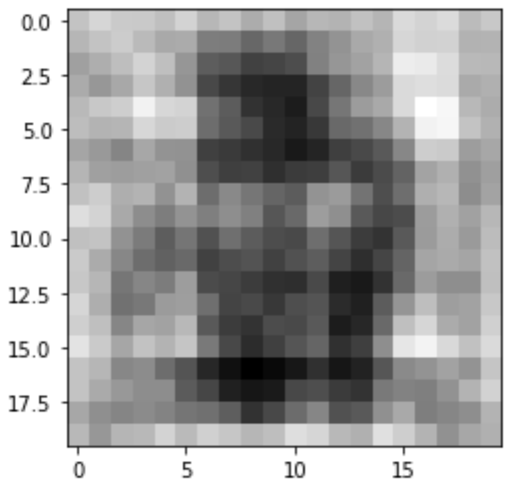


Figure 10: Generated Image 3. (Image by author)


0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0
5
10
15