HW4

Rahul Chamarthi

Introduction:

The goal of this assignment was to train a discriminator/generator pair on the CIFAR-10 dataset utilizing techniques from DCGAN, Wasserstein GAN(WGAN), and ACGAN. Since the implementation of an ACGAN was a bonus, and I was limited on time, this report will summarize my findings experimenting with DCGAN and WGAN. I will first outline my methods and implementation, then discuss key results, share final reflections and provide instructions for running and viewing the code.

Methods and Implementation:

My implementation of DCGAN and WGAN can be broken down into the following key phases:

1. **Setup (Import Libraries, Loading Dataset):** Required PyTorch and Torchvision libraries were imported, and the CIFAR-10 dataset was downloaded and loaded.
2. **Data Preprocessing:** A DataLoader was setup to batch the data. A key step was applying a Normalize transform to scale the images to the [-1, 1] range. This step is required to match the generator's final Tanh activation layer.
3. **Model Architecture (DCGAN):** I defined the Generator (using ConvTranspose2d to up sample noise into an image) and Discriminator (using Conv2d to downsample an image to a probability) as PyTorch nn.Module classes. The architecture followed the core principles of the DCGAN paper, using BatchNorm and ReLU/LeakyReLu as activations.
4. **Model Architecture (WGAN):** The WGAN used the exact same Generator architecture as the DCGAN. The Discriminator was modified to become a "Critic" by removing the final Sigmoid layer. The removal of the Sigmoid layer allowed the output of a score rather than a 0-1 probability.
5. **Training Loop:** Unlike a simple model with the .fit() function, I had to implement a custom training loop for both models. This involved separately calculating losses, zeroing gradients(.zero_grad()), backpropogation(.backward()), and updating weights(.step()) for the Generator(G) and Discriminator(D) in each batch.

Just like in HW3, I decided to implement all this code in a Jupyter notebook. This was even more critical for this assignment. Being able to break the code into chunks and, most importantly, plot the generators output after each epoch was the only real way to debug whether models were learning or just collapsing.

My initial training runs were a complete failure, something I will discuss in more detail in the Discussion on Results. Both models, even after 100 epochs, only produced static noise as seen in Image 1 below. I believe these issues could be traced back to the Discriminator becoming "too good", too fast. The Generator's gradients vanished, and as a result it never learned anything.

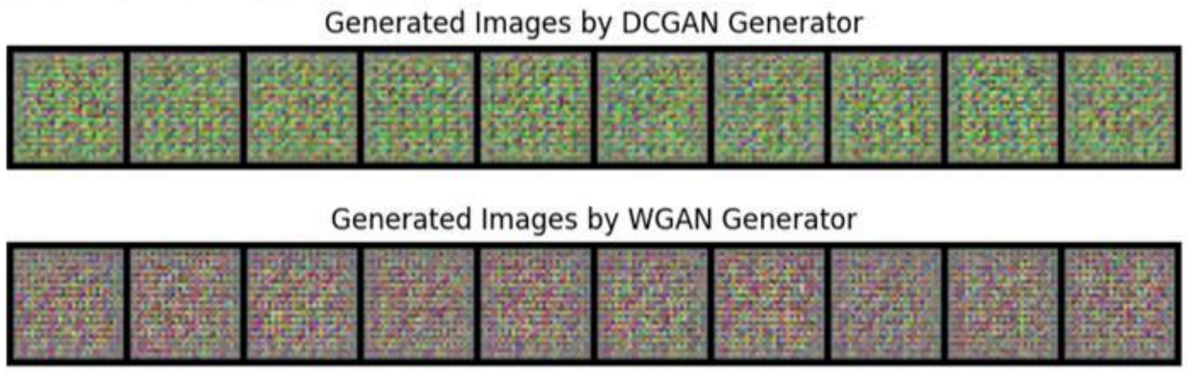
Generated Images by DCGAN Generator
Generated Images by WGAN Generator

Image 1. Static noise in first GAN runs

To fix this I had to implement a few key changes:

- For DCGAN: I had to "de-power" the Discriminator by giving it a slower learning rate(0.0001) than the Generator(0.0002). This gave the Generator a "fighting chance" to learn before the Discriminator became perfect.
- For WGAN: I implemented the specific changes from the WGAN paper:
    - Used the RMSprop optimizer(instead of Adam)
    - Implemented weight clipping, forcing the Critic's weights to stay in a small range [-0.01, 0.01] after each batch.
    - Trained the Critic 5 times for every 1 Generator update.

Discussion on Results:

After implementing the fixes, the two models finally began to train, and results were counter-intuitive but interesting.

DCGAN: Looking at the DCGAN, the loss graph, seen in Image 2 below, told a story of instability. The Discriminator's loss trended down while the Generator's loss climbed, which usually suggests training failure. However, the final generated images, seen in Image 3, completely contradict this. The model produced surprisingly high-quality images of what appear to be birds, animals and other objects.

Upon doing some research into this phenomenon I learned a key lesson with GANs: for a standard DCGAN, the loss graph has very little correlation with image quality. The "minimax game" was chaotic, but in that chaos, the generator still "won" and found a way to produce excellent fakes.

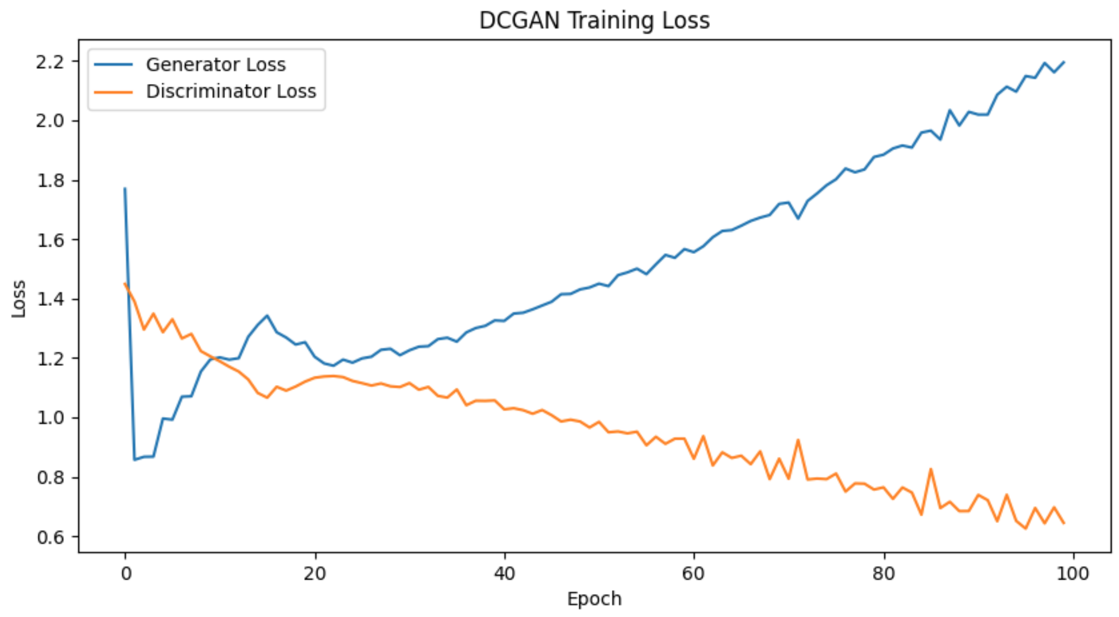

Image 2. DCGAN Training Loss

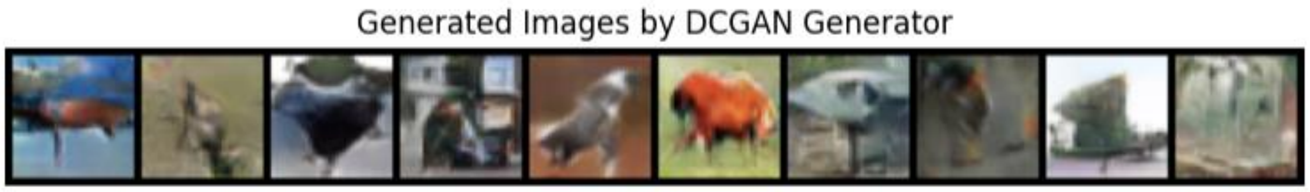

Image 3. DCGAN Top 10 Images

WGAN: The WGAN, on the other hand, did exactly the opposite of the DCGAN. Its loss graph, seen in Image 4, was the perfect picture of convergence. The Critic and Generator losses moved smoothly and stabilized, proving that the WGAN's promise of training stability is true.

However, the final images, seen in Image 5, are blurrier, less defined, and less visually impressive than the DCGAN's. The WGAN converged, but it converged to a "safer" solution that produces fuzzy averages rather than the sharper fakes of the DCGAN. The best explanation I can offer for this is that it could be a side-effect of the WGAN's weight clipping. Weight clipping might be a method to introduce stability but might prevent convergence to the best possible solution.

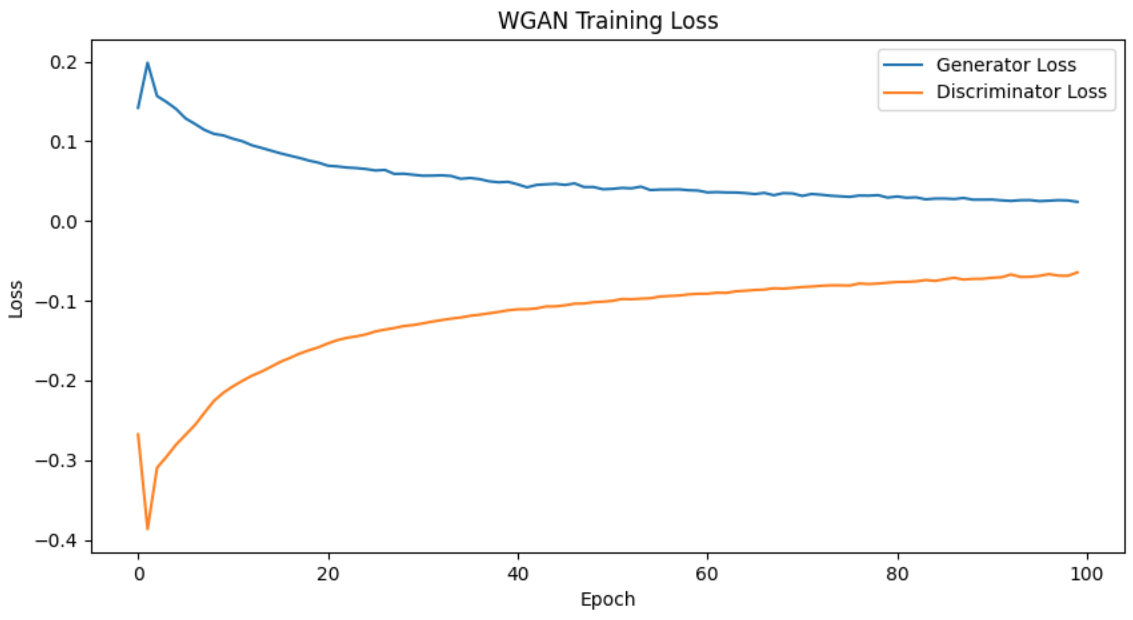

Image 4. WGAN Training Loss

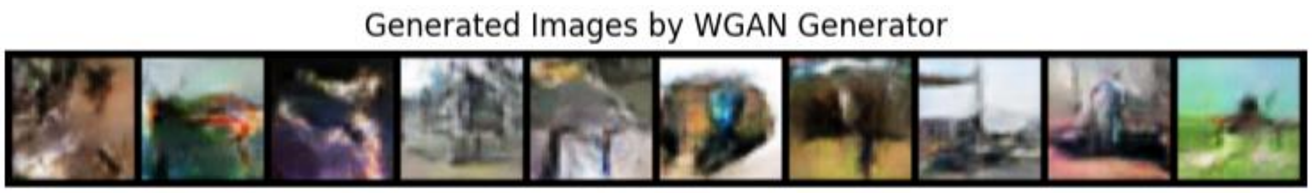

Image 5. WGAN Top 10 Images

Overall, these findings were very thought provoking. The "unstable" model(DCGAN) produced the best pictures, while the "stable" model(WGAN) produced the best loss graph. This highlights a key trade-off: DCGANs are a "lottery" that can fail or produce amazing results. WGANs, in contrast, are a reliable tool to guarantee convergence but at the potential cost of finding the best solution. Given the results, the DCGAN was the clear winner of my experiments despite suggestions to the contrary from the loss graphs.

Final Reflections:

I think that there's been a running theme throughout this class for me of not having time to complete these assignments due to conflicts with my other classes, I still believe this assignment was one of the most challenging yet. The concepts are one thing, but the art of getting a GAN to actually train is another thing entirely.

My key takeaway here is a lesson that the results made perfectly clear: the loss graph does not provide the whole story. In HW3, a good loss curve meant a good F1 score. In HW4, my best-looking model(DCGAN) had a loss graph that looked like a complete failure. My “stable” model(WGAN) had a perfect loss graph but produced blurrier images.

It really drove home the idea that these models aren’t just being “optimized” – they’re in a “game”. The DCGAN’s “game” was chaotic and unstable but produced a winning result. The WGAN’s “game” was stable and converged, but was the less impressive, “safer” solution.

With the above being said, I am happy with how my implementation turned out. Debugging the initial “noise-only” problem and then being able to directly compare these two model results felt like a real breakthrough. The result was as much more interesting outcome than if both models had worked or failed on the first shot.

Instructions on Running:

Given the lack of specific instructions around naming formats for our submission, I have uploaded the Jupyter notebook hw4.ipynb to my GitHub in the folder HW4. Due to the nature of GANs, the trained models are not saved, as the final notebook output shows the end result.

The code can be found at my Clemson GitHub: **https://github.com/rchamarthi-edu/CPSC8430**.