Group 3 Special Project: Generative Adversarial Network (GAN) Creation using the PyTorch and Anacondas Libraries, and the Struggles of CycleGAN

Dean Clark

ENGR 010

Dr. George Witmer

November 29, 2023

Introduction: Before starting the project, I was almost sure that I wanted to do something regarding AI, not only because of its very prevalent nature in day-to-day activities in both the present and future, but also because of the unknown difficulty and barrier to entry surrounding it. From talking to peers, family, or anyone that didn't have a full grasp of what made up these artificial intelligence models. I knew relatively little about the architecture, functionality, and process of creating a model, but knew that there were endless possibilities for what I could learn. Looking at what I was most interested in trying to create, making an AI act like a human was one area that piqued my interest, and as I started to delve deeper into what programmers could do with this, I stumbled upon the process that I wanted to do for this project, which ultimately landed me on Generative Adversarial Networks.

The project below focuses on Generative Adversarial Networks, or GANs, which are a type of machine learning systems that are made to mimic given data from a library or large data set to create more of similar but not identical output. After a set number of epochs, or iterations that the training set will perform on the network to come up with its output, data is then generated, which in this case will be using Matplotlib. GANs are the primary example of a type of unsupervised learning using artificial intelligence, and are also, as the title says, generative, rather than discriminative. Discriminative models do not actually create new outputs from the dataset that they are given, and are more seen as a classification tool that will give a prediction based on what it has been trained on to point at which class of data belongs to. For both types of models, there is a training phase that uses some type of data, and is given a random generator seed so that the experiment can be replicated on any machine. For the discriminator, a two dimensional input gives us a one dimensional output, which is the real data (or data provided by a generator) and the probability that the sample belongs to the real training data, respectively.

Examples of discriminative models can be neural networks, logistic regression models, and support vector machines (SVMs).

On the other hand, generative models are used to create what computer scientists call synthetic data, meaning data that is fully created by the model using the contents that it was trained on. These data points follow similarly to the training data distribution, but are completely made up in the sense that the model is the only entity that created it. Generative models use both a discriminator and a generator, to first be taught what data is similar to its test data set, and then something to produce a similar result. For input, it takes data from latent space, hidden or compressed data (which gets into a whole other topic computer science). If still struggling with latent space data, it is similar to when a human sees a many times in real life, so much that they can form a general image of a dog in their head without needing to see another or if someone was to describe a dog to them, they could come up with a picture from thin air without having to have the dog directly in front of them or seeing an image of the dog itself. After the training stages of a discriminator are done, the generator uses this latent space and generates an output that resembles this image that it is coming up with in latent space. This type of learning, using data that is unlabeled or raw in some way is known as unsupervised learning, and starkly opposed the discriminative model's form of supervised learning which relies on clearly labeled and differentiated data that can be put into specific categories, which is the entire job of the discriminator. Being able to determine implicit patterns in the data that is thrown at them is the first part of a GAN, and then being able to classify and replicate a similar type of result using machine learning is executed after. Generally, discriminative models can be broken down into two separate groups: classification, which is the process of the model accurately assigning the data into specific categories, and regression, which uses the model to understand differences

between an independent and dependent variable, being able to predict trends on a graph given points. Unsupervised models, like generative models, are meant to tackle other types of tasks, but use the ideas of classification and regression to make them work. There is clustering, which is a data mining technique often meant to group data that is seemingly unrelated into its own determined groups. Association is also a type of unsupervised learning which finds relationships between variables in a data set, used to then recommend/predict other instances of that variable that should be in the datasets from its training. Finally, least like any of the discriminative models is dimensionality reduction, which takes a group of data that has a lot of dimensions, or features to it, and tries to reduce that number to make it more generalized while still preserving the integrity of the data.

A few things that need to be covered before diving into the architecture of GANs are some of the mathematical functions behind the networks, which gets extremely complicated, but will be simplified for this report. The first thing that has to go into the GAN is a loss function, which tries to guess or reflect the distribution of the data generated by the GAN and the distribution of the real data. Typically, one loss function is used for the discriminator, and one for the generator. The loss function is tried to be minimized through every epoch for the generator and maximized for the discriminator, as shown in the implementation in the loop for running every epoch, which effectively saturates the generator so it keeps producing proper output. The loss function described in the first two generators is a Binary Cross Entropy function, which is defined as such in the PyTorch implementation:

$$\ell(x,y)=L=\{l_1,\dots,l_N\}^\top \text{ and } l_n=-w_n[y_n\cdot\log x_n+(1-y_n)\cdot\log(1-x_n)]$$

Summarizing what this is, ℓ(x,y) represents the loss for a single data point where x is equal to the input that is being given to the model and y is equal to the label given to this input. L

is the vector that has all of the individual losses for each data point in the set. $\{l1,\ldots,lN\}^T$ transposes the vector L. This is the generator loss, and is trying to be maximized over the iterations of epochs. As for the discriminator's loss function, l_n is the loss for the nth data point. The weight of the nth data point is denoted as w_n , which helps assign different levels of importance for different samples. Next, y_n is the true label for the data point, which is binary, meaning that it is either 0 or 1 respectively. Finally, x_n is the predicted probability that the nth data point belongs to class 1. The class of 0 holds all of the synthetic (or perceived synthetic) samples, whereas the 1 class holds the real (or perceived real) samples. Additionally, for the weight update rules, the Adam (Adaptive Moment Estimation) algorithm is used as the extension of the Stochastic Gradient Descent (SGD) optimization algorithm which is specifically designed for efficient deep neural networks. Initializing the first and second moments of gradients, m and v, to 0, we use this to update the first and second moment estimates,

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \mid v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$$

correct for the bias in the first and second moment estimates,

$$m_t = m_t \backslash 1-\beta_1^t, \mid v_t = v_t \backslash 1-\beta_2^t$$

and then updates the parameters to

$$\theta t+1 = \theta t - \alpha \cdot m_t \backslash \operatorname{sqrt}(v_t) + \varepsilon .$$

This adjusts the learning rates for each parameter individually, which is based on the assumption that the generator and discriminator may have learning rate requirements. This leads to faster convergence, works on varied batch sizes, and makes the optimization process more stable and efficient overall. On top of all of the different algorithms that are used to train the data that are described here are other algorithms such as nn.linear, nn.ReLU, nn.Droupout, and nn.Sigmoid

that all apply their own functions from the neural network library in PyTorch to build the discriminators and generators.

There are many different kinds of GAN architectures that are implemented to train a lot of the AIs that we know that use this type of model. The one that I mainly focused on learning about (and attempted to use) was CycleGAN, a GAN used to simply transform images into different styles of images, and its successor, Contrastive Unpaired Translation (CUT). CUT is built off of CycleGAN, but is much faster and less memory intensive. CycleGAN uses two mapping functions, G : X-> Y and F: Y-> X with associated discriminators D_x and D_y , to make it so that the images from G(X) are almost nearly indistinguishable from Y. It learns the inverse mapping function in order to better create its own output. Additionally, using loss functions similar to the ones described above as well as a cycle-consistency loss function, which enforces that F(G(X)) is approximately X, as well as G(F(X)) is approximately X. This type of transitivity is an important part of the training processes without paired images. At the cutting edge of computer vision and machine learning are these models, which both take a large library of images of similar types to change some aspect of the image drastically. One of the first ideas for the GAN was changing a horse into a zebra in a picture, and evolved to changing pictures to be taken in the morning to the night, changing a dog to become a cat, or, the one that I was working on until I couldn't, turning any picture of a cat into the popular cat on the internet, Grumpy Cat. With multilayer patchwork contrastive learning, a part of the image is then broken down into chunks, changed in the spots that needed to be, and then run through the discriminator again to test which allows for unpaired cyclical transfer. Cycle consistency applies the transitive training idea of turning X into Y and Y into X to allow the creation of bijective maps between the two. Spatial consistency is a large part of CUT's differences in comparison to CycleGAN, preserving the structure of the translated

images. CUT also uses global and local discriminators, with the global overseeing the entire structure of the image, whereas the local discriminator focuses on finer details. The process of the GAN's training is all the same, but is a lot more specified in the sense that it uses many custom loss functions, methods to train the discriminator and generator, as well as actually outputting the information to the user that involves math well above my pay grade. However, from the knowledge that I did gain from this, it follows the preparation of the data to be trained, the definition of the discriminator and generator, the training and then outputting of the data for the user. The scope of this library is massive, and is in turn extremely complex because of how much goes into it. Now that the theory behind Generative Adversarial Networks is understood, we can specifically talk about the GANs that were defined in my project.

Creation of GANs: At the beginning of this process, I had no idea what any of this was or how to start, so I resorted to RealPython's Articles and Guide: "Generative Adversarial Networks: Build Your First Models". The first step was downloading the PyTorch library, the Anaconda Python Distribution, and the conda package to the system. By creating an environment, and then working in a Jupyter Notebook locally attached to that environment, building the GAN could be done. To save a GAN so that it works on any system that runs it, PyTorch allows for a random seed to initialize the neural network's weights. Each input has its own associated weight to it, which will be modified to minimize the loss function using the backpropagation by calculating how much of an effect that the weight has on the output. However, it does not do this without flowing back through the discriminator and its weights first, showing the cyclical nature of each epoch that the model has to go through to better fine tune its data.

The first part of both creations was initializing the data before being trained with either the discriminator or the generator, where mainly PyTorch's tensors (sort of like a NumPy array),

or a multi-dimensional matrix of a singular data type is used to store the training data. For the handwritten digits, you need to check if the machine that is running the code has a GPU using the CUDA add-on which is compatible with Nvidia GPUs, and if not, uses the CPU to train the data which takes quite the long time (about 2 minutes per epoch, which for me was 200 minutes since I ran 100 epochs). Then, by making sure that the first time the code is run, the MNIST data set will be downloaded to the current directory, which is a collection of 70,000 handwritten digits and their respective labels, and then creates a data loader. For both GANs, we first need to create a discriminator and a generator, which for the handwritten digits, will take in a 28x28 pixel image and provide a probability of the image being in the original dataset. You can vectorize them so that the neural networks receive vectors with a certain amount of coefficients. After the class and method declarations for the Discriminator, we need to instantiate it and send it to the GPU (if the GPU is active). After this, we increase the dimensions of the input for latent space, which then is used for creating the class and methods for the Generator class. Then, to actually train the models, we define the learning rate, number of epochs, and the loss function, as well as the optimizer for the discriminator and the generator. For the first example, we used a higher learning rate, however to obtain a better result, we decreased the learning rate. We use the binary cross entropy function because it works to classify the data into real and fake groups, which is an observable result. We also use the Adam algorithm to create the optimizers, and then implement a training loop. We take examples from latent space, feed it into the generator, then plot the results which are shown in the picture below.

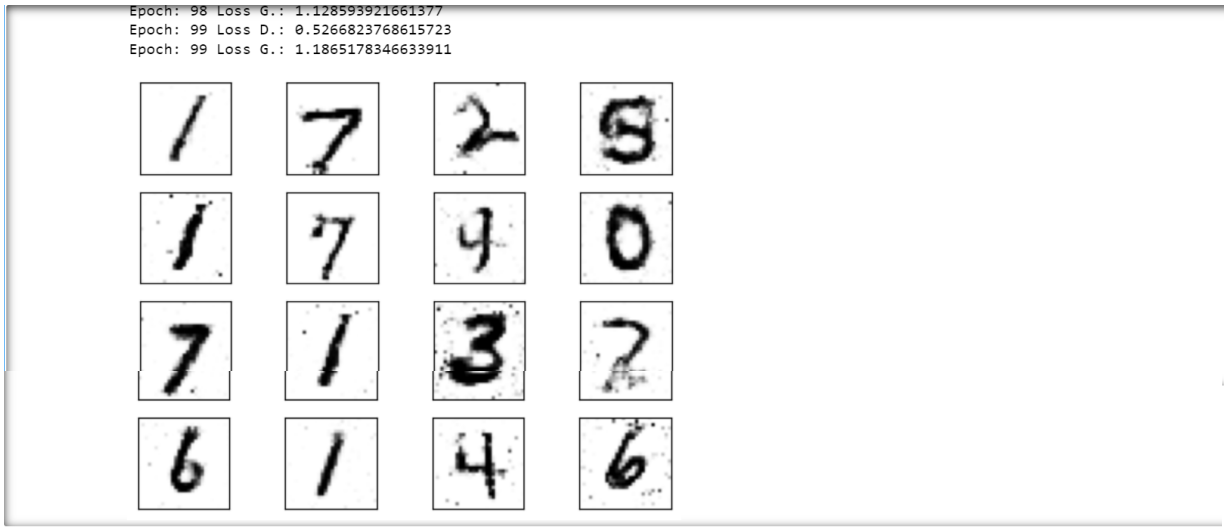

Figure 1: Image of the handwritten digits that were generated after 100 epochs

Scope/Complexity: To preface, I understand that the project technically was not entirely what I wanted to do, since I was trying to work with CycleGAN/CUT which would have more practical usage/a larger scope of usage for actual users and not just for learning. However, still learning about the library allowed me to better understand the process of training and building upon an already existing GAN to give it new usage. The complexity of the project was, to be frank, a bit simpler than I would've wanted it to be, but the knowledge gained cannot be understated, as I really researched how all of these functions worked together. I didn't just try to look up the code and what to do, but tried to take all of the different mathematical processes and building of the architecture into account so that given more time and new prior knowledge that I have, I would be able to build a more advanced model than I did. Without going into it too much, as the section is next, the setback of not having a GPU compatible with CUDA was the only setback, which I tried to solve but was unable to, meaning I could not train the data on my system. While my project specifically probably would not be able to be used on other types of data, a very similar idea would work for other types of user generated data in mass datasets which could then produce output from its latent space.

Setbacks: There were little to no setbacks regarding the setup process for the first 2 GANs, as the tutorial clearly laid out everything that needed to be downloaded and set up in preparation for the actual formation of them. That being said, when it came to my independent research for one of the GANs, there were many problems which unfortunately did not allow me to finish the one in time. For CycleGAN and CUT, the network is trained using an Nvidia GPU and the CUDA toolkit, which has GPU-accelerated libraries which allow for the training to occur. Initially, since the first two GANs could use any GPU or even could use systems that only required a CPU to run the training and the displaying of the generated data, I thought it would be similar for CUT. However, when running the command to try to train the generator for the grumpifycat, or the GAN that would turn any cat picture into grumpy cat, it gave me the error that CUDA did not exist. Remembering back to the specs of my laptop, as well as confirming this with my task manager, I have an AMD CPU and GPU in it, meaning that no matter how many times I ran the command, it would not work. I tried to change the two files test.py and train.py to use the AMD equivalent of this, ROCm and OpenCL, but was unable to make it work for my GPU. In an attempt to fix this, I tried downloading Ubuntu 18.04 LTS, and had to use the Windows powershell to download a proper version of Linux so that Ubuntu would work, but I could not get CUT to run on Ubuntu. While this was disappointing, I instead shifted my focus to the documentation of the code that I already had, as well as understanding the functions that made GANs like CycleGAN work. Part of the issue stemmed from needing to download Python into the virtual environment, since for some reason, it could not run Python files. Additionally, I was switching between a few command lines, including Git Bash to download and clone the repository for CUT and CycleGAN, Anaconda before I knew I couldn’t use CUDA (and for the other two GANs to run Jupyter Notebooks locally on my laptop), Ubuntu in my attempts to use

Linux-based command line and ROCm in replacement of CUDA for CUT, as well as the Windows PowerShell to actually download the version of Windows Subsystem for Linux that I needed for Ubuntu to run properly. All of the commands, which libraries were imported where, and which ones had access to what blended together even in my best attempt to differentiate everything, and led to many setbacks almost like working in different languages.

Conclusions/Key Takeaways: From this project, I got to delve deep into the vast world of Generative Adversarial Networks, taking much away from not only the principles of how generative and discriminative models work and writing/training them, but also a look into how software developers have to navigate working in different environments and trying to debug/find workarounds when certain libraries or programs are not compatible with each other. To preface, while I was unable to get CUT to work with the GPU on my system, meaning I was unable to complete one of the goals that I set out when beginning this project, it was not out of lack of effort and it was not wasted time in the slightest. Still getting to understand how this GAN worked on a deeper level allowed me to hopefully tackle something along similar lines in the future to more success, and I was still able to produce other working programs that, while maybe not as impressive or practical, showed learning and growth throughout the course of the semester. The skills that I attained by simply trying to get ROCm to work with CUT instead of CUDA was something that could definitely happen in the real world, trying to get either a piece of unsupported or outdated piece of software working on different/new machines. This process was very similar to debugging for the Engineering Assignments and Lab Assignments in this course, but was stumped in the face of this challenge. Despite my best efforts and searching the internet for a solution, I could not find one.

Sources:

And super-resolution arxiv:1603.08155v1 [CS.CV] 27 Mar 2016. (n.d.-a). https://arxiv.org/pdf/1603.08155.pdf

Antoniadis, W. by: P. (2023, June 11). *Latent space in deep learning*. Baeldung on Computer Science. https://www.baeldung.com/cs/dl-latent-space

ArXiv.org e-print archive. (n.d.-b). https://arxiv.org/pdf/1703.10593.pdf

BCELOSS¶. BCELoss - PyTorch 2.1 documentation. (n.d.). https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html

Contrastive Learning for unpaired image-to-image translation. in ECCV, 2020. Contrastive Learning for Unpaired Image-to-Image Translation. (n.d.). https://taesung.me/ContrastiveUnpairedTranslation/

Craigloewen-Msft. (n.d.). *Manual installation steps for older versions of WSL*. Microsoft Learn. https://learn.microsoft.com/en-us/windows/wsl/install-manual#step-4---download-the-linux-kernel-update-package

Das, S. (2023, September 4). *6 gan architectures you really should know*. neptune.ai. https://neptune.ai/blog/6-gan-architectures#:~:text=pixel%2Drnn%2Dtensorflow-,text%2D2%2Dimage,cat%20could%20be%20very%20random.

Delua, J. (2021, March 12). *Supervised vs. unsupervised learning: What's the difference?* IBM Blog. https://www.ibm.com/blog/supervised-vs-unsupervised-learning/

Dwivedi, H. (2023, August 29). *Understanding gan loss functions*. neptune.ai. https://neptune.ai/blog/gan-loss-functions#:~:text=Standard%20GAN%20loss%20function%20

Erkam. (2022, February 13). *A review of contrastive learning for unpaired image to image translation*. Medium. https://medium.com/@erkams/a-review-of-contrastive-learning-for-unpaired-image-to-image-translation-6df94ac9610a

Ganesh, K. S. (2022, September 7). *What's the role of weights and bias in a neural network?* Medium. https://towardsdatascience.com/whats-the-role-of-weights-and-bias-in-a-neural-network-4cf7e9888a0f

Generative adversarial nets - neurips. (n.d.-c). https://proceedings.neurips.cc/paper_files/paper/2014/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf

Google. (n.d.). *The generator | machine learning | google for developers*. Google. https://developers.google.com/machine-learning/gan/generator

How to install WSL and ubuntu 18.04 LTS on Windows 10. University of Iowa College of Public Health. (n.d.). https://www.public-health.uiowa.edu/it/support/kb48549/

Quick start (windows). Quick Start (Windows) - ROCm 5.7.1 Documentation Home. (n.d.). https://rocm.docs.amd.com/en/latest/deploy/windows/quick_start.html

Real Python. (2022a, July 28). *Setting up python for machine learning on windows*. https://realpython.com/python-windows-machine-learning-setup/

Real Python. (2022b, October 7). *Generative Adversarial Networks: Build Your First Models*. https://realpython.com/generative-adversarial-networks/#handwritten-digits-generator-with-a-gan

Tiu, E. (2020, February 4). *Understanding latent space in machine learning*. Medium. https://towardsdatascience.com/understanding-latent-space-in-machine-learning-de5a7c687d8d

Unpaired image-to-image translation using cycle-consistent adversarial networks. CycleGAN Project Page. (n.d.). https://junyanz.github.io/CycleGAN/

Varile, M. (2019, February 14). *Train neural networks using AMD gpus and Keras*. Medium. https://towardsdatascience.com/train-neural-networks-using-amd-gpus-and-keras-37189c453878