CS 613 Final Project COT-GAN Sprite Generation

Tuote Huang th579@drexel.edu
Tajung Jang tyj27@drexel.edu
Chris Oriente co449@drexel.edu
Marcus Sullivan ms5262@drexel.edu

Abstract

This paper studies an image generation method using a Generative Adversarial Net (GAN). We train an ensemble of models on different animation frames to simulate creating a walk animation. We also demonstrate that the use of artificial enhancement on the generated data can improve the quality of the generated images and significantly reduce the number of required training epochs.

Background

In the process of game development, some visually dynamic content requires a lot of complicated animation art production processes such as intensive pixel modification. That requires high time and capital costs. Using procedural algorithms is a very convenient way to reduce costs. One such graphic resource development technique is called the sprite generation technique. This technique is a method of expressing motion while continuously changing a large number of images. We want to better understand how the generation of new and novel sprites or the generation of future frames is implemented in this technique to animate a given sprite.

The most general algorithms are iterating equivalent building blocks to build sprites that lack hand-crafted image detail, exploring carefully crafted algorithms using pseudo-random seeds, and generative-adversarial networks (GANs) in deep neural networks. In our paper, we choose to use GANs. GAN uses a different training method than general artificial neural networks. It consists of two networks, a generator, and a discriminator, which compete and train against each other. Based on the GAN, there is a model called Causal Optimal Transport Generative Adversarial Net (COT-GAN)[5]. The Optimal transport describes the algorithm that finds the shortest distance between probability densities. In our paper, the probability density will represent the pixels of a sprite to predict the steps to move to a different probability density function. We will try to combine optimal transport and the laws of causality to predict time-dependent data distributions. Finally, put it all together and create a new GAN to generate these animation steps and use the Fréchet Inception Distance to estimate our GAN performance quantitatively.

Related Work

Past Work in generating sprites

The works we have based our motivation from for this project include the resources [1] [3] [5] [6]. They all generate sprites using a generative model, either CNN autoencoder or GAN. Within the works we were inspired from, there are two main problems they tackle. Either the generation of new and novel sprites or the generation of future frames to animate a given sprite.

Approach

Project Plan: Phases 1-3

We plan out our work with certain milestones to achieve. Firstly we want to generate a new sprite image that looks similar to the images from the Liberated Pixel Cup. This involved creating the framework for a GAN and testing it on a single animation frame. In phase 2, we plan to follow [1] and add a deep convolution layer to our GAN. This improvement directly impacts model generalizability and we will test expanding our data set to assess the models generalizability. In phase 3, time willing, we will implement the COT-GAN algorithm described in [5]. This will be able to take sprite image, and predict how the individuals pixels should move to complete an animation. The algorithm relies on two facets; causality, which enables to model to preserve the order of events, and optimal transport.

Thorough our progress on the project we encountered a few bottleneck mentioned further on, so we were only able to complete the first phase of the project, and started to experiment with adding a deep convolutions layer to our GAN.

Generative Adversarial Networks (GANs)

A Generative network is a type of Artificial Neural Network that specializes in generating artificial data from random noise. The term adversarial comes from the fact that the network is actually comprised of two distinct models, the generator and the discriminator, that ”compete” against each other. The goal of the generator is to generate data that can trick the discriminator into classifying them as ”real.” As implied, the goal of the discriminator is to properly classify a combination of real and generated data as either real or fake.

Training GANs

The generator learns based on whether or not it was able to trick the discriminator. The discriminator learns based on its ability to properly classify the data. As such, GANs require two distinct forward and backward propagation steps per epoch. The first pass is used to train the discriminator. The generator generates fake data from random input. This data is then forward propagated through the discriminator. The gradient for the discriminator objective function (often log loss) with respect the discriminator's output is then calculated and backward propagated to its fully connected layer where the weights and biases will be updated. Now that the discriminator has been trained, the generator can also be trained to compete with the updated discriminator. The generator once again generates fake data. This time, the data is propagated by themselves (without real data) through the discriminator. Once again the gradient is calculated and backward propagated, but this time the fully connected layer of the discriminator is not updated. Instead the gradient continues to be propagated back to the fully connected layer of the generator, whose weights and biases are then updated. The general overview of the architecture is depicted in Figure 1.

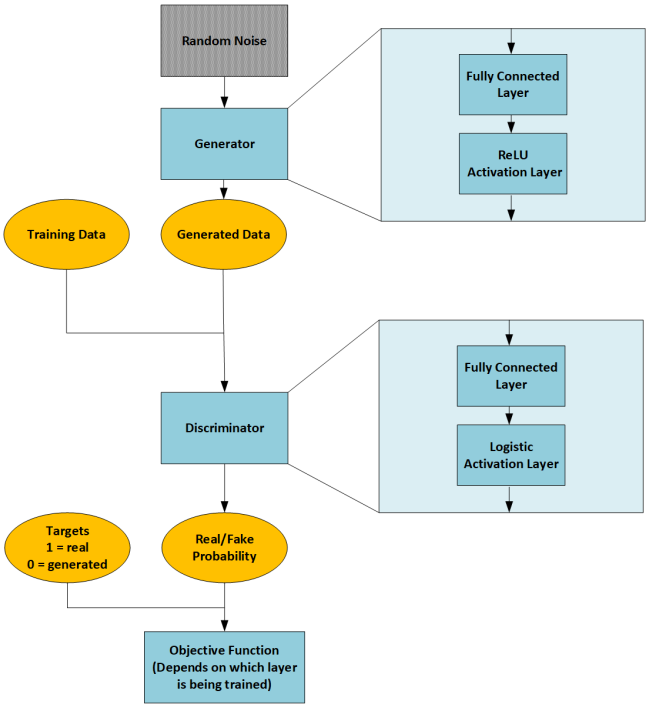

Figure 1: GAN architecture

Implementation

Training Data

In order to train the GAN we used open-source sprite sheets from the Liberated Pixel Cup [4].

Each sprite sheet contains all the necessary animation frames for one of the six components required to generate a complete sprite. These components include hair, eyes, body, top wear, bottom wear, and shoes. Each row of the sprite sheet corresponds to a unique animation and every column corresponds to a unique frame within that animation. A total of 1256 unique 64x64 pixel sprites with 4 channels (RGB and image alpha) were generated by programmatically combining the appropriate frames for each component as shown in Figure 2. The individual frames for each animation were saved as separate PNG images (examples included in Figure 3). When played in sequence, these sprites form an animation (Figure 4). The code used to generate the sprites was acquired from the following repository: https://github.com/YingzhenLi/Sprites.git. Once generated, the desired frames were again, programmatically selected. Each image was converted into an array and the alpha channel was dropped since it was irrelevant for the generated images. The resulting 64x64x3 arrays were then flattened and stacked in order to generate a 1256x12288 input array for the GAN. These input arrays were then saved using the Pickle library for convenience.

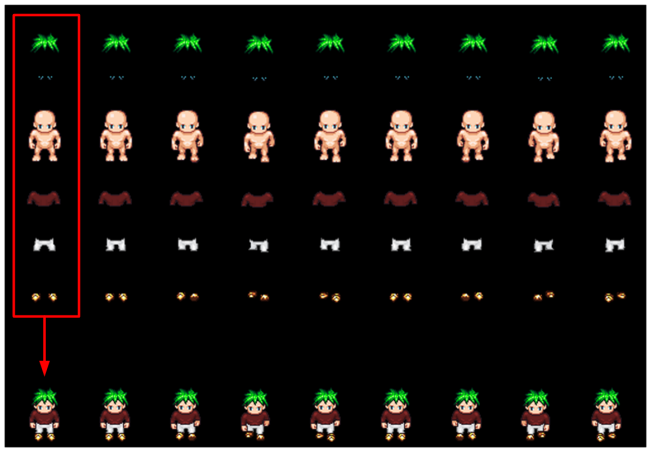

Figure 2: Generating individual frames for each animation

Figure 3: Example Sprites

GAN Architecture

The GAN architecture utilized was relatively simple. The Generator consisted of a single Fully Connected layer (features in: 12288, features out: 12288) with a Rectified Linear Unit (ReLU) activation layer. The Discriminator consisted of a single Fully Connected layer (features in: 12288, features out: 1) as well as Logistic activation layer which output the probability (between 0 and 1) that a generated image was real. A log loss objective function was used to evaluate the

Figure 4: Animations - Walk, spell & slash animations

probabilities produced by the discrimination. The following objective function was used for the generator:

$$J_G = -\sum_{i=1}^{N} ln(\hat{Y}_i) \quad (1)$$

Where $\hat{Y}_i$ is defined as the probability output from the discriminator.

The model was trained using stochastic gradient descent with a batch size of 100. As input, for the Generator, a set of random features was generated from a normal (Gaussian) distribution using standard NumPy functions where μ and σ were calculated from the entire training data set. Note that since the features of the training data were all on the same scale (i.e., 0 to 255), the training and input data was not z-scored. A summary of the hyperparameters is included in Table 1.

Table 1: Hyperparameter Summary Table

Hyperparameter	Value
Generator Learning Rate	$1x10^{-5}$
Discriminator Learning Rate	$1x10^{-5}$
Stochastic Batch Size	100

Evaluating GAN Performance

In order to provide a quantitative evaluation of the generated images the Fréchet Inception Distance (FID) [2] was calculated between the best image from the generator and a sample real image from the training data. The best image from the generator was defined as the generated image with the largest corresponding probability from the Discriminator. The FID compares the distribution of generated images with the distribution of real images using the equation below:

$$\|\mu - \mu_w\|_2^2 + tr(\sum + \sum_w - 2(\sum^{1/2} \sum_w \sum^{1/2})^{1/2})$$

where, μ and σ are the mean and covariance matrix of the generated image. μ_w and σ_w are the mean and covariance matrix of the world (i.e., real) image.

A lower value for the FID indicates a lower distance between the two images (i.e., more similar images).

GPU

In order to speed up training to enable better evaluation and parameter tuning, CUDA was used to enable parallel matrix operations. To get around the fact that the CUDA API is targeted for C/C++ programs, we used the CuPy library. The CuPy library works out of the box to replace existing NumPy code and run it on CUDA devices. By doing so the expectation was that it would significantly improve our training times. Unfortunately, the resulting images created by the CUDA implementation did not produce sprites as we hoped. This is likely due to data not maintaining its integrity between either some propagation steps or when passing between the discriminator and generator. The source of this issue is still being investigated.

The following is in acknowledgment that a true runtime comparison is not valid between the original CPU and CUDA implementations due to the outputs not matching. The computations remained the same between implementations. The only difference between the two were the representation of arrays as CuPy arrays rather than Numpy arrays. Run-times for training were significantly faster on the CUDA implementation. With the Numpy functions replaced with CuPy, the run-times decreased with an average speed up of 2.75x (see Figure 5)

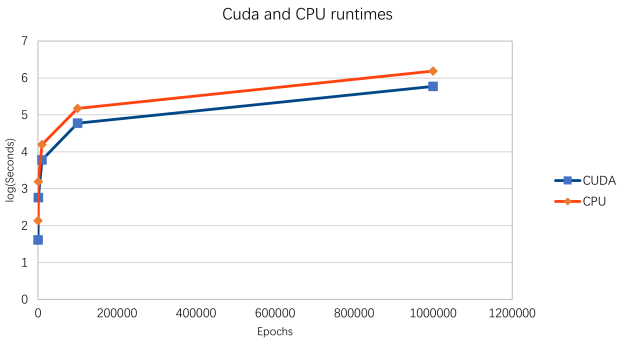

Figure 5: The Cuda and CPU runtimes.

Evaluation

Initially, during training, it appeared as if the GAN was not learning to generate anything but a black block of 64x64 pixels. After closer examination, however, it became clear that features of a new sprite were beginning to appear, they were just extremely faint. To better visualize the outputs, the best generated images were artificially enhanced by applying the following equation to each feature in the image:

Where, x is the feature value, s is the scaling factor, t is the threshold.

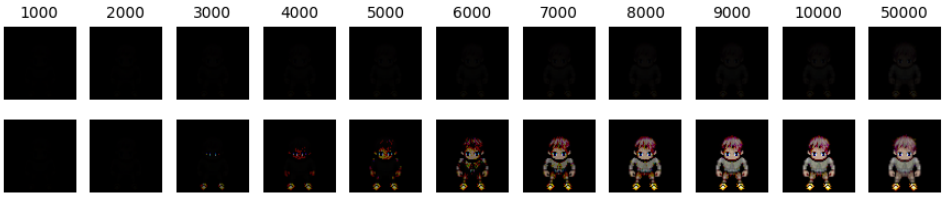

Figure 6: Normal (Top) and Enhanced (Bot) GAN outputs across 50000 epochs.

The results showing the regular GAN outputs across 50000 epochs are included in Figure 6. A standard scaling factor and threshold of 15 and 5, respectively, was used for all images with the exception of the 50,000 epoch image, which used a scaling factor of 9 (to avoid over saturation). As is evident in the figure, the artificial enhancement resulted marked improvement in the quality of the generated images.

In order to better evaluate the performance of the GAN, the FID was calculated for the generated images. Table 2 provides the FID values for both the standard and enhanced generated images as well as the change in the enhanced FID relative to the last checkpoint. Note that the FID values were scaled to be between 0 and 1 by dividing each FID by the maximum FID for that dataset. Figures 7 and 8 depict the improvement in FID during training for both the standard and enhanced generated images, respectively. The FID for the standard images improved after each training checkpoint and ranged from 1.00 to 0.637. Similarly, the FID for the enhanced images also improves after each training checkpoint and ranges from 1.00 to 0.198. Notably, the total relative improvement in FID for the enhanced images (0.802) was far greater than that of the standard images (0.363).

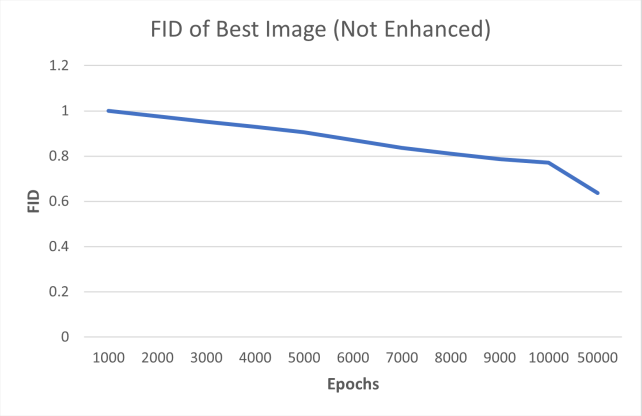

Figure 7: FID of Best Image(not enhanced)

By using the data from Table 2 and Figure 8, we were able to identify a point of diminishing returns in terms of epochs. The change in enhanced FID between checkpoints 8000 and 9000, was only 0.017 (half that of the previous checkpoint). As such, we determined that the optimal training duration was 8000 epochs for future models.

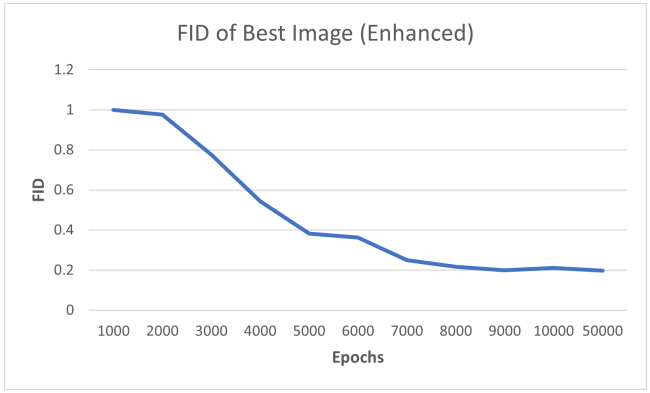

Figure 8: FID of Best Image(enhanced)

Table 2: Evaluating the performance of the GAN

Number of Epochs (checkpoint)	FID*	Enhanced FID* (s = 15, t = 5)	Change in Enhanced
1000	1.000	1.000	N/A
2000	0.976	0.976	0.023
3000	0.951	0.776	0.200
4000	0.928	0.544	0.231
5000	0.905	0.382	0.161
6000	0.870	0.363	0.019
7000	0.836	0.250	0.112
8000	0.810	0.216	0.034
9000	0.786	0.198	0.017
10000	0.771	0.211	0.013
50000	0.637	0.198**	0.013

* Scaled to be between 1 and 0 by dividing by the maximum FID.

**Scaling factor of 9 was used.

Simulating Sequence Data with Multiple GANs

In order test the consistency of the GAN across sequence data we took slices of the data set and separated the frames of the walk animation. We trained 2 more GANs on frame 3 and frame 7 of the walk animation for 8000 epochs. The best images from each model were then enhanced similarly to the neutral sprite. Our result from these models are in Figure 9, the neutral frame is repeated. The results indicate a similar level of qualitative performance across the sequence data.

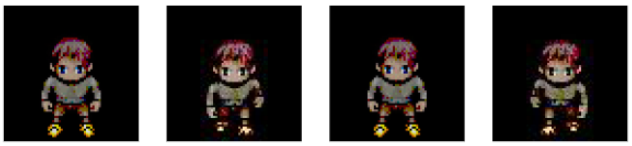

Figure 9: Walk Animation

Analysis and Conclusions

Based on the data, a simple GAN model is able to learn to generate images with accurate representations of the real sprite data. These representations, however, are extremely under saturated relative to their real counterparts resulting in qualitatively unconvincing images. Artificial enhancement of the generated images is confirmed to be a valid strategy for generating realistic 2D sprites using a simple GAN. Moreover, the observed drop off in FID improvement over time indicates that models that utilize this strategy would benefit from implementing shorter training times when using an image enhancement layer. This could be useful when simulating sequence data using multiple GANs.

Future Work

Transposed Convolution

Another approach to improving training times would be to utilize a DCGAN. In this model, the fully connected layer of the generator is typically much smaller is followed by a series of convolutional 2D Transpose layers that upscale the data based on learned filters (sets of kernels). Previous studies have shown promising results using this type of architecture for image generation with a GAN. [5]

Optimal Transport

In the final phase of our project we would like to use to create a full animation of a single character. We will do this by using Optimal transport. Optimal Transport is an age old algorithm dating back to Napoleonic era. Napoleon army used it to move mounds of earth from on distribution to another, we will be using it to move the pixels on the screen from one position to the other. More specifically, the equation is used to find the shortest distance between distributions. The picture below lays out the parts of this equation we need to understand to solve. We have two distributions a and b. We want to find out how we can move the mass in a to get it to look like b. This is the transport plan P. The transport cost is a mapping of weights representing the amount of work to move the mass from the position in a to the position in b. We would like to apply this concept to our GAN so we can create a larger number of novel animation.

Through training the model will learn and update the weight of the transportation cost to reflect the realistic movement of pixel in the training data.

CUDA

The CuPy library offers both the ability for JIT compilation via Python AST or invocation of C/C++ style kernel definitions within the python module. A better implementation would to create separate kernels for the discriminator and generator and split them across the cores of the GPU. Weights, biases, and other related data specific to each kernel can reside in shared memory between cores. Data that needs to be transferred between kernels can be stored in global memory. By utilizing CUDA streams, CUDA kernel operations can be ran concurrently and pipe-lined such that explicit kernel transfers from CPU are unnecessary. Such implementations are reserved for future work.

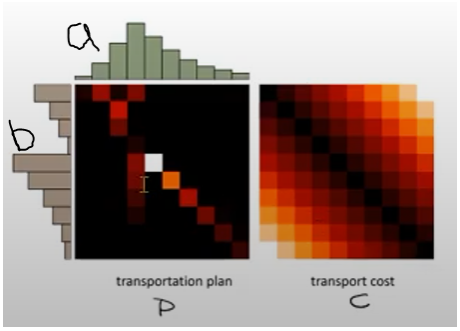

Figure 10: Optimal Transport Density Plot.

References

[1] C. Henrichs, S. Wani, and S. Feroz. CNN Sprite Generator. https://curthenrichs.github.io/CS534-Term-Project-Website, last accessed on June 03, 2022.

[2] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. 2017.

[3] S. Hong, S. Kim, and S. K. and. Game Sprite Generator Using a Multi Discriminator GAN. *KSII Transactions on Internet and Information Systems*, 13(8):4255–4269, August 2019.

[4] Pennomi. Liberated Pixel Cup. https://opengameart.org/content/liberated-pixel-cup-0, last accessed on June 03, 2022.

[5] T. Xu, L. K. Wenliang, M. Munn, and B. Acciaio. COT-GAN: Generating Sequential Data via Causal Optimal Transport, 2020.

[6] T. Xue, J. Wu, K. L. Bouman, and W. T. Freeman. Visual Dynamics: Probabilistic Future Frame Synthesis via Cross Convolutional Networks, 2016.