Unconditional-to-conditional Transfer and Optimization for Web-based Skybox GAN

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

By

Crystal Kwong

Spring 2025

The Designated Project Committee Approves the Project Titled

Unconditional-to-conditional Transfer and Optimization for Web-based Skybox GAN

By

Crystal Kwong

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2025

Dr. Chris Pollett	Department of Computer Science
Dr. Mark Stamp	Department of Computer Science
Prof. Kevin Smith	Department of Computer Science

Abstract

Generative adversarial networks (GANs) are known for their ability to generate high quality images mimicking real life or even particular art styles. Yet for all their capability, casually training a GAN on an average machine can be infeasible as GANs require an enormous amount of time and data to train. Even with a trained GAN, model inference demands heavy computations, making GANs difficult to deploy on applications. To address these limitations, techniques such as transfer learning and quantization have been leveraged to speed up training of GANs and lighten computational cost of GAN inference. This project aims to use such techniques to efficiently train and optimize a sky image GAN for deployment on a skybox generator web page. We conducted experiments which demonstrated that transfer learning, in the form of direct weight splicing from a pre-trained unconditional model to a conditional model, accelerates the conditional model's training. After 100 thousands of images (kimg) of training, the model with transferred weights achieved an FID score of 29.01, outperforming the conditional model trained from scratch which obtained an FID score of 134.51. As for quantization, our results appear less impressive as the GAN model size of 120 megabytes is shrunk only to 117 megabytes, and inference speed seems to remain unaffected. The final web page deploys this model through ONNX Runtime Web and presents an interface allowing users to generate skyboxes based on cloud types, fulfilling a use case of a choice-based AI skybox generator.

Keywords: GAN (Generative adversarial network), StyleGAN2-ADA, Unconditional to conditional transfer, Quantization, ONNX, Web deployment, Skybox

Table of Contents

List of Figures

List of Tables

I. Introduction

Generation of relevant text, images, and videos with AI can be applied to many fields. For example, in the media and entertainment industry, AI can be used to create visual effects and stylized environments for movies and games [1]. Skyboxes may be considered a kind of stylized environment as they are a 3D technique commonly used to render immersive environments in video games. Popular titles such as Halo, Call of Duty, and Half-Life 2, to name a few, all make use of skyboxes [2, 3]. Skyboxes offer computationally efficient rendering because they consist of fixed images, and this same characteristic of skyboxes can be taken advantage of by AI image generators to create AI-generated skyboxes [4]. Compared to generating characters, AI-generated skybox backgrounds may avoid the uncanny features associated with generated human-like characters, making skyboxes an appealing target for AI generation with the goal of speeding up game development [5].

A common type of generative AI model is the generative adversarial network (GAN). GANs, however, are limited by resource intensive training as well as computationally heavy inference that complicates their deployment to real-world applications [6, 7]. Fortunately, these issues can be reduced by techniques like transfer learning and quantization. Transfer learning reduces the amount of data needed to train a GAN, and quantization decreases the computational and memory cost of inference [6, 7]. Transfer learning involves transferring weights from a pre-trained model to a target model, and Frégier and Gouray (2021) suggest that this technique can speed up training from 6 to 656 times depending on the size of the network [8]. After the model is trained, the model can be compressed through quantization, which replaces high-precision computations with lower precision computations [6]. The logic behind quantization is that lower precision weights require less memory for storage and less energy to

perform calculations with [9]. While research on quantizing GANs is sparse, the benefit of quantization has been shown on other types of models such as binary neural networks, which have achieved a 32 times smaller model size when quantized from 32-bit precision to 1-bit precision [10].

This project's goal is to first train and compress a sky image GAN using transfer learning and quantization techniques. The specific GAN used for this project is StyleGAN2-ADA, a model capable of generating highly realistic images [11]. After the model is trained and quantized, it is deployed in a web application that generates a sky image. The sky image is transformed into the final skybox in the form of a panorama-like image, which is used to render the illusion of an infinite background in a 3D scene. The concept of an AI skybox generator website is not entirely new; similar websites already exist such as Skybox AI and Rosebud AI [12, 13]. The Skybox AI website receives notable traffic, accumulating 93.34K recorded visits during the month of March 2025 [14]. Rosebud AI received 315.92K visits in a month, although it is likely that not all visits were made for skybox generation since the site also offers AI generation for games, websites, and applications in general [13]. To generate a skybox, both websites' skybox generators request a text prompt as input. Some sample prompts on Skybox AI even reach over thirty words [15]. Text prompt enables tailored skybox generation, but such lengthy prompts require effort in producing the prompt itself. While these websites satisfy the use case of seemingly unlimited possibilities in skybox generation, this project aims to fulfill a simpler use case of a choice-based skybox generator which bypasses the cumbersome process of textually detailing a skybox.

The organization of this project report is covered as follows: Chapter 2 provides an overview of previous work done on unconditional-to-conditional knowledge transfer, GAN

quantization, Skybox AI generators and StyleGAN2-ADA web page deployment. Chapter 3 lists the background information and tools referenced in the project. Chapter 4 states the dataset and the base pre-trained model selected for the experiments, which Chapter 5 lists and describes in detail. Chapter 6 explains the model web deployment process and the web page interface, then walks through usage of the web application up to implementing the generated skybox image in Unity. The conclusion of the project and future work are covered in Chapter 7.

II. Related work

A. Unconditional to conditional GAN Transfer with Hyper-modulation -

Transfer learning has been established as an effective method to reduce model training time [16]. In the context of GANs, transfer learning may be conducted between unconditional or conditional GANs. Unconditional-to-unconditional GAN transfer has been largely studied, but there is a gap in research on unconditional-to-conditional knowledge transfer for GANs. Laria et al. (2022) addresses this gap and proposes hyper-modulation as a technique to facilitate unconditional to conditional knowledge transfer for GANs, using StyleGAN as their example model [17]. While effective, the technique is inconvenient to implement as it involves modification of the base GAN architecture as well as the construction of a separate hypernetwork [17].

B. Quantization of GANs

Quantization of GANs is considered more challenging than quantization of classifiers since GANs are more complex models. Studies on GAN quantization are also sparse compared to studies on classifier quantization, and even the StyleGAN quantization study by Andreev and Fritzler (2022) fell short of truly quantizing the StyleGAN model. Instead, they simulated quantization by utilizing the PyTorch fake quantization interface, and they were able to detail the quality of quantized 4-bit and 8-bit images without performing real quantization [6].

C. Quantization of Sequential StyleGAN2

A StyleGAN2 quantization script is provided in an engineering design optimization project managed by Intel [18]. However, the StyleGAN2 model used in the script is in fact a rewritten sequential version of the StyleGAN2 model [19]. Given a lack of quantized original

StyleGAN2 models, this suggests that quantization has largely been achieved on simplified StyleGAN2 versions rather than the original StyleGAN2 model.

D. AI Skybox Generator Websites

Two notable AI-powered skybox generator websites are Skybox AI and Rosebud AI [12, 13]. Skybox AI uses a model to generate a 360-degree environment image which can be used as a skybox, and the websites provide a 3D preview of the generated skybox [20]. These websites accept a text prompt input to generate the skybox. With the flexibility of text prompt input, these skybox generators may render more than just skies: they can generate scenery from landscapes to stylized trees to castles, demonstrating the seemingly limitless potential of AI skybox generation.

E. Web Deployment of StyleGAN2-ADA Model

In the web example titled “Running StyleGAN2-ADA in browser” by Guido De Jong [21], a StyleGAN2-ADA model is converted into ONNX format and deployed on a web page through ONNX Runtime. The web page lists advantages of this web deployment approach including accessibility and flexibility on multiple devices and platforms. On the other hand, disadvantages include slowness on CPU and high bandwidth usage [21]. The successful web deployment of StyleGAN2-ADA inspires us to follow this approach by using ONNX Runtime to deploy the StyleGAN2-ADA model.

III. Background

This chapter defines relevant terms and concepts used in this project. First, this chapter defines a traditional GAN, focusing on its architecture and the process of training it. StyleGAN is introduced as a GAN modified from the traditional architecture to generate even higher quality images than before. The improved versions StyleGAN2 and StyleGAN2-ADA are also described briefly. As for the model training process, this chapter defines the commonly used terms: Frechet Inception Distance (FID) score, transfer learning, and quantization. Finally, an overview of relevant technologies for utilizing and deploying the project GAN model is given. Those relevant technologies include 3D skyboxes, Open Neural Network Exchange (ONNX), and Unity.

A. Generative Adversarial Network (GAN) -

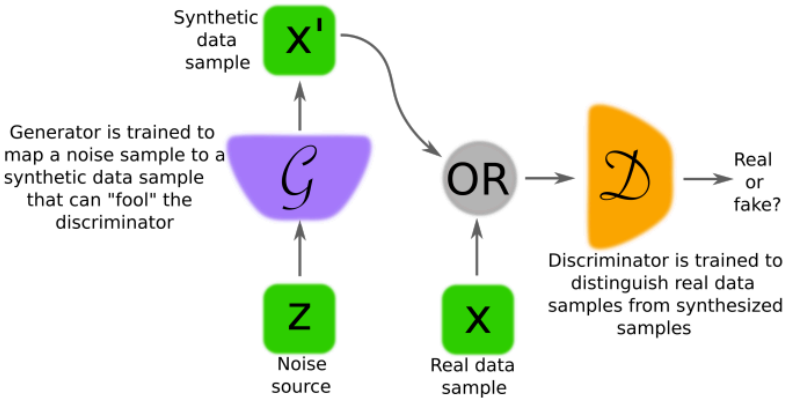

Figure 1: Traditional GAN design [23]

A generative adversarial network (GAN) is a type of generative AI model which can generate any content encoded as data such as text, images, and music. Generation involves feeding a latent code, typically represented by a randomized vector, into the input layer of the generator, which eventually converts the latent code into an output [22]. While responsible for synthesizing output, the generator model is only one component of a GAN. A traditional GAN consists of two models as illustrated in Figure 1: a generator model which generates the content,

and a discriminator model that classifies the generator output as real or fake. Training a GAN involves alternating training between those two models. In alternating training, one model is trained while the weights of the other model remain fixed. The generator synthesizes an output that becomes the input of the discriminator, and the discriminator returns a probability of whether the output is real or synthesized. This probability, or error signal, is effectively a loss function used to improve the discriminator as well as the generator. Through training, both models improve simultaneously: the generator synthesizes increasingly realistic images, while the discriminator better learns to distinguish real images from the improved fake images. Ideally, the GAN is trained until the synthesized images are indistinguishable from the real images such that the discriminator classifies the synthesized images correctly only half of the time [23].

1. Conditional GAN:

A conditional GAN (cGAN) is a type of GAN that can generate class-conditioned outputs as opposed to the random unlabeled outputs generated by an unconditional GAN [23]. To obtain a cGAN, a conditioning label can be added to the input of both the generator and the discriminator as shown in Figure 2 [24].

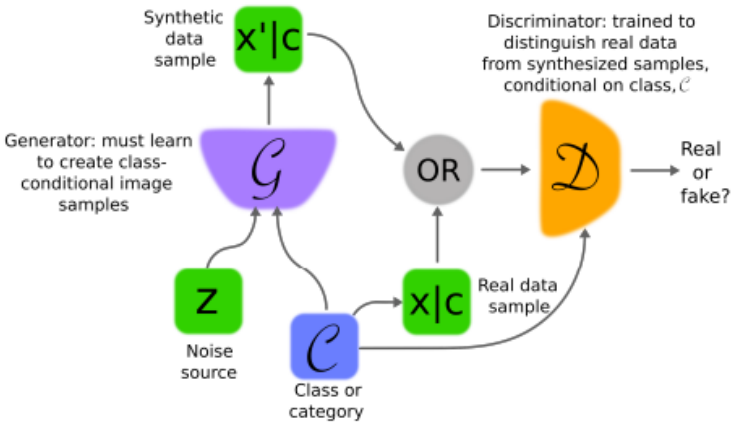

Figure 2: Conditional GAN design [23]

B. StyleGAN -

StyleGAN is a GAN with a generator architecture that is redesigned to enable style transfer. While the traditional GAN takes an input latent code through the input layer, StyleGAN's redesigned architecture instead maps the input latent code to an intermediate latent space 'w' as shown in Figure 3. This mapping contributes to greater disentanglement, or separation, of image attributes such as hair style or face shape. Disentanglement is favorable as a disentangled representation is suggested to make generating realistic images easier. The mapped input is then converted by the model's learned affine transforms into styles, which control the adaptive instance normalization (AdaIN) operations inside the synthesis network [22]. The AdaIN layers adjust "the mean and variance of the content input to match those of the style input" [25]. In other words, AdaIN transfers the style's appearance to the target image, similar to merging a style image with a base image to produce an image drawn in the new style. Another feature of StyleGAN is the addition of noise at each layer of the synthesis network. This noise helps create stochastic features such as freckles. Finally, a progressive growing technique grows a smaller resolution image to a larger resolution image until StyleGAN outputs the final high resolution image. Overall, this new generator architecture allows StyleGAN to generate high quality images with greater disentanglement of image attributes [22].

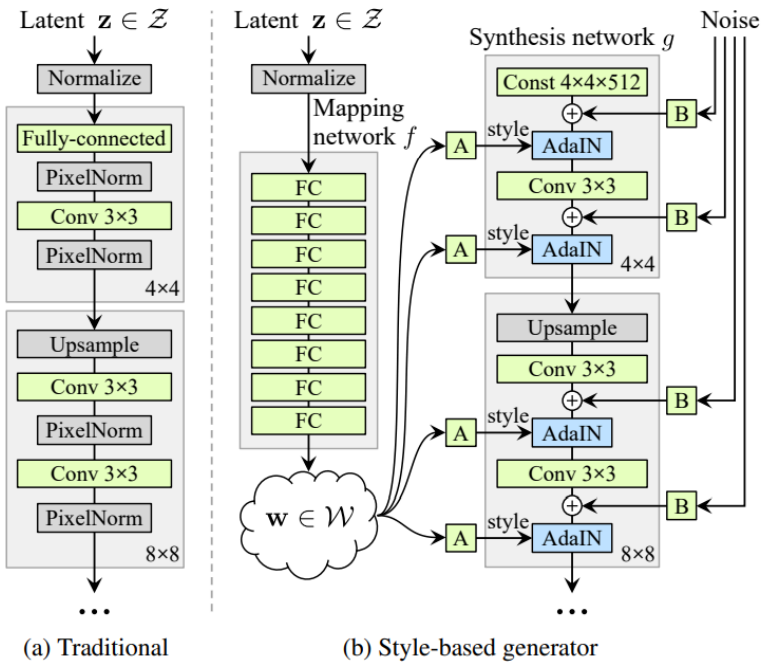

Figure 3: Traditional vs Style-based generator architecture [22]

C. StyleGAN2 and StyleGAN2-ADA -

StyleGAN2 is built on top of StyleGAN with changes to the generator aimed at removing an artifact found in StyleGAN, as well as further improving image quality through the path length regularization introduced to the generator. The artifact was caused by normalization, and to fix this, StyleGAN2 replaced adaptive instance normalization with a "demodulation" operation. Demodulation scaled output feature maps by dividing by the standard deviation of the feature map. The difference between instance normalization and demodulation was that the "demodulation technique is weaker because it is based on statistical assumptions about the signal instead of actual contents of the feature maps" [26]. As for path length regularization, perceptual path length (PPL) was a metric introduced in StyleGAN, and it represented the difference between two images during a linear interpolation of those two images [22]. Lower perceptual path length was associated with better quality images, and the implementation of path length regularization aimed to encourage lower perceptual path length [26]. StyleGAN2-ADA further

built on StyleGAN2 and introduced an “adaptive discriminator augmentation” (ADA) mechanism that aimed to reduce discriminator overfitting, which tends to happen on smaller datasets. This allowed high quality results to be obtained from training on small datasets [11].

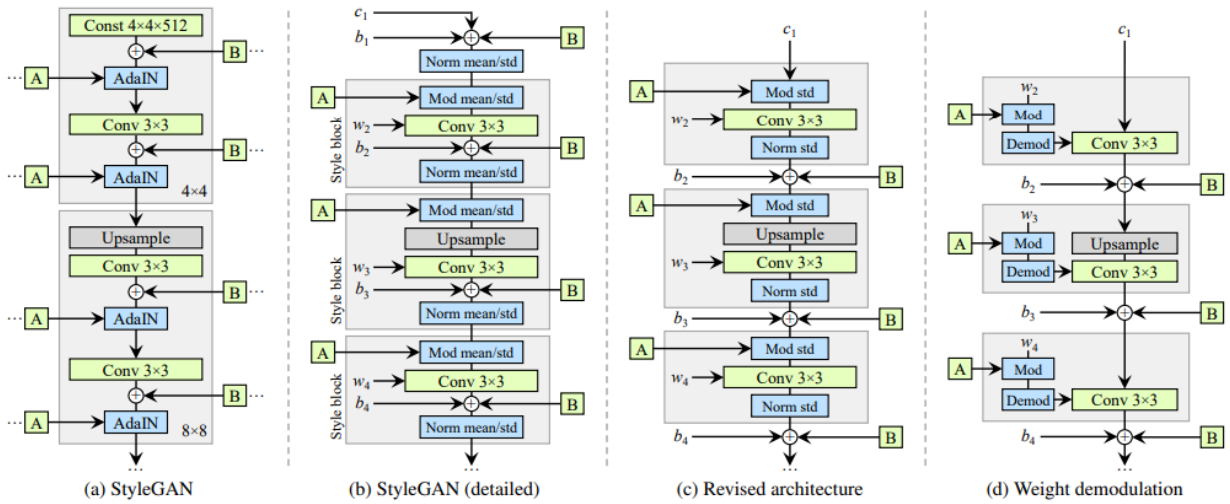

Figure 4: Comparison between StyleGAN and StyleGAN2 architectures [11]

D. Training a model -

Training a model refers to an iterative process of feeding an input through the model’s layers to obtain an output. Through feedback from the loss function, backpropagation, and repeated iterations, the model’s weights are gradually adjusted closer toward the ‘correct’ value until desired weights have been reached such that the model performance is satisfactory [27].

E. Frechet Inception Distance (FID) Score -

FID score is a metric to measure both diversity and quality of images generated by a GAN. The lower the FID score, the more diverse and high quality that the GAN’s generated images are. The FID score “calculates the distance between images in pixel space” through the formula shown in Figure 5 [7]. In the formula, ‘u’ is defined as the mean, ‘Tr’ is the function of summing the main diagonal – the top left to bottom right elements – of the matrix, and sigma is

the covariance matrix. ‘X’ represents the true image and ‘g’ represents the model-generated image.

$$FID\,(x,g) = ||u_x - u_g||^2 + Tr\left(\sum_x + \sum_g -2\left(\sum_x \sum_g\right)^{\frac{1}{2}}\right)$$

Figure 5: FID score formula. Adapted from [7]

F - Transfer Learning

Transfer learning is a method of utilizing a pre-trained model’s weights to accelerate a target model’s training. When training from scratch, a model is initialized with completely random weights. The model is trained for the full duration until its weights have been adjusted to produce a satisfactory model. Training for less than the full duration while keeping quality results is possible through transfer learning, where the weights of a pre-trained model are leveraged to bypass the random weight initialization. The model starts training from a state of partial knowledge, thus requiring less time to train than if it had started from a randomized state of no knowledge. Since it saves time as well as computational resources, transfer learning is an efficient technique for training GANs [16].

Transfer learning covers several different methods, one common method being fine-tuning. Fine-tuning is simple as it involves retraining every weight in a pre-trained model, gradually adapting the entire model to the new data. It is also possible to only retrain select layers of the model. The purpose of restricting fine-tuning to certain layers is to preserve certain knowledge of the model inside untouched layers while having the rest of the layers re-learn from new data. This alternate method may be referred to as partial fine-tuning [7].

G - Quantization

In the context of AI models, quantization involves replacing higher-precision computations with lower-precision computations in order to reduce model size and inference workload. For example, quantization may be implemented by replacing high-precision weights with low-precision weights. A model's weights are typically stored in floating-point formats like 32-bit or 16-bit since these higher-precision formats can carry more information. It is possible for weights to be stored in lower-precision integer formats of 8-bit or 4-bit, and these formats are commonly used for quantization because they take up less storage space than floating-point formats. Besides shrinking model size, quantization may also speed up model inference since calculations using smaller formats can be computed more quickly. Overall, quantization reduces computation cost and model size at the cost of accuracy loss from saving weights in a lower-precision format [6]. Ideally, a balance is found to optimize the model without sacrificing too much model accuracy.

Quantization commonly comes in modes of static and dynamic quantization as well as post-training and quantization-aware training. Static quantization refers to quantization calculations being performed outside of model inference, while dynamic quantization refers to calculating quantization parameters during actual model inference. The more efficient method is static quantization because it uses pre-computed parameters when calculating activations, while dynamic quantization must re-compute activations during inference. As for post-training and quantization-aware techniques, post-training quantization refers to quantizing an already trained model, while quantization-aware training refers to quantizing the model during training [6]. These two techniques can be combined with static and dynamic quantization. For example, the PyTorch quantization library provides post training dynamic quantization, post training static

quantization, and static quantization aware training. PyTorch post-training static quantization involves quantizing the weights and activations of a model after training and before inference. PyTorch post-training dynamic quantization also refers to quantization of weights performed before inference on a trained model, but it leaves activation calculations to be done during inference. Finally, quantization aware training (QAT) can be applied to static, dynamic, or weight only quantization according to the Pytorch quantization library, and it involves performing calculations during training while simulating fake quantization before actually quantizing the model, allowing "for higher accuracy compared to other quantization methods" [28].

H - Open Neural Network Exchange (ONNX) model

Open Neural Network Exchange (ONNX) is an interoperability tool that can represent models from various frameworks, such as PyTorch and TensorFlow, as a common file format. In this file format, the converted model is represented as a collection of nodes forming a graph. During the process of model conversion, the model is further optimized through graph optimizations such as node fusions [29].

I - Skybox

In 3D graphics, a skybox refers to a technique of wrapping a background over the viewer to create the illusion of an encompassing sky or environment. To make a skybox, a cubemap can be wrapped around the inside of a cube, or alternatively, a panorama image can be wrapped around the inside of a sphere. For a cube shaped skybox, the six texture sides must blend together at the seams to create a smooth 3D environment look as illustrated in Figure 6. For a sphere shaped skybox, a 360 panorama image wraps seamlessly around the inside of a sphere [4].

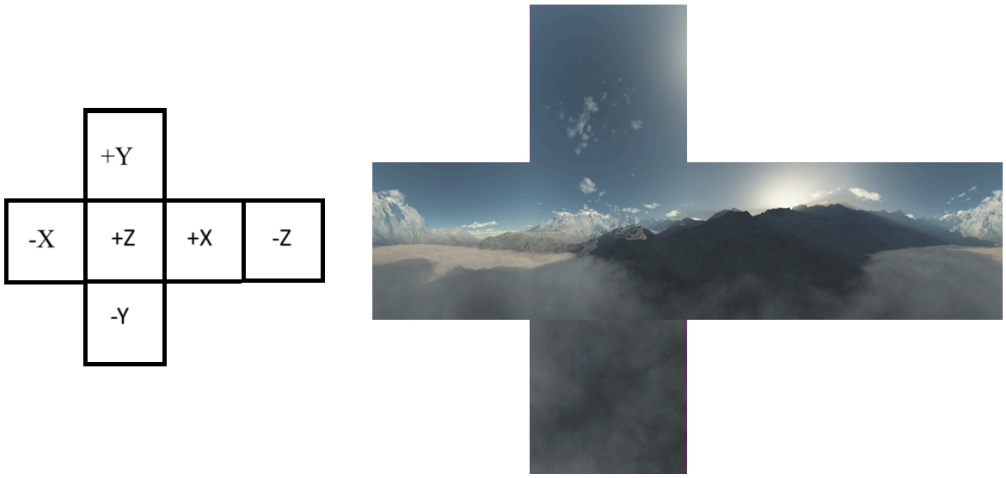

Figure 6: Example Skybox layout for Unity [30]. Adapted from ClassiCube Skyboxes [31]

J - Unity

Unity is a game engine that allows a user to add objects and lighting to a scene, making it suitable for 2D and 3D game creation [32]. Skyboxes are supported in Unity and can be created by arranging an imported set of image files into a 6-sided skybox, which can then be applied to the Unity camera and viewed in a Unity scene. Alternatively, instead of a 6-sided skybox, a single panorama image can also be used with the panoramic skybox setting.

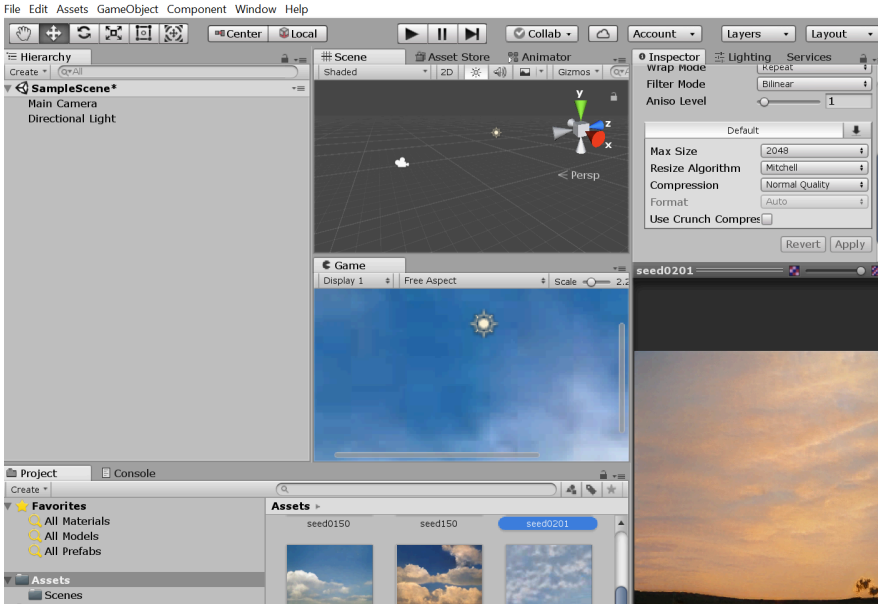

Figure 7: Unity engine interface

IV. Dataset and Base Pre-trained Model

The Cirrus Cumulus Stratus Nimbus (CCSN) dataset is used to train the final sky image model. This dataset contains 2543 cloud images divided between eleven classes of cloud types. The images are sized 512 by 512 pixels and saved in .jpg format [33].

Before the dataset can be used to train a conditional StyleGAN2-ADA model, a JSON file associating each image in the dataset with the appropriate class label must be obtained. This is done using a script that takes the dataset directory as an input and generates the JSON file as an output [34]. The generated JSON file is then placed in the dataset directory at the folder level. From here, conditional model training can be done by calling the train.py method with the "--cond" flag set to 1.

A pre-trained model of texture images is obtained from an online collection of pre-trained StyleGAN2 models. This model is trained on the Describable Textures Dataset (DTD) and outputs colored texture images of 512 x 512 resolution [35]. The purpose of this model is to serve as a base model that provides starting weights for the target sky image model.

V. Experiments

This chapter describes the experiments undertaken in training and optimizing the final model. While most experiments were performed in a conda virtual environment with Python 3.9.20 and PyTorch 1.7.1+cu102, the hyper-modulation experiment was done in a different conda virtual environment with Python 3.8.8 and PyTorch 1.9.1+cu102. All model training is done with two K40m GPUs.

A - Fine-tuning a Pre-trained Model

Fine-tuning is a transfer learning technique that can rapidly adapt a model's output towards a desired domain by tuning the weights of the model. This method is applied to a pre-trained texture image model, and its effectiveness is compared against training a model from scratch. To begin the experiment, a StyleGAN2-ADA texture image model is downloaded as a Python pickle (.pkl) file. Next, the model is trained on the Cirrus Cumulus Stratus Nimbus (CCSN) dataset for 100 "kimg", where "kimg" is defined as "thousands of real images shown to the discriminator" [11]. The total training time was approximately twenty hours. A second model is trained from scratch using the same dataset and configuration in order to act as a base case for comparison.

The results of training showed that, even with only 100 kimg of training, fine-tuned image quality appeared to improve greatly. The improvement is reflected in the rapidly decreasing FID scores. As for the base case, images still appear grainy by the end of training, and the higher FID scores reflect the poorer image quality.

kimg	FID
0	265.07
20	62.21
40	35.25
60	28.23
80	27.16
100	25.55

Table 1: FID scores per 20 kimg of fine-tuned GAN

kimg	FID
0	331.11
20	395.83
40	250.42
60	214.35
80	173.39
100	198.09

Table 2: FID scores per 20 kimg of GAN trained from scratch

This experiment establishes that fine-tuning is a faster and effective method to train a new GAN model. The success of this experiment provides a base for the next step of unconditional to conditional knowledge transfer, as fine-tuning may be used to train the conditional model together with weight transfer.

B - Unconditional to conditional transfer

This chapter outlines experiments performed on two methods to achieve unconditional to conditional transfer: hyper-modulation and the direct transfer of pre-trained weights into the target model. While both methods are found to be effective in transferring knowledge, the method of direct transfer is preferred for its simplicity and ease of implementation.

1. Hyper-modulation

The main idea of hyper-modulation for unconditional to conditional transfer, as described in [17], is utilization of a hypernetwork to generate the weights for all classes. The hypernetwork takes in a given source model weight along with a class embedding and outputs the desired target weight, and this process is done in real time during training. The paper also introduces "self alignment" as a method to initialize the newly introduced hypernetwork parameters with the goal of reducing training time [17]. After performing self-alignment, the model can be trained using hyper-modulation.

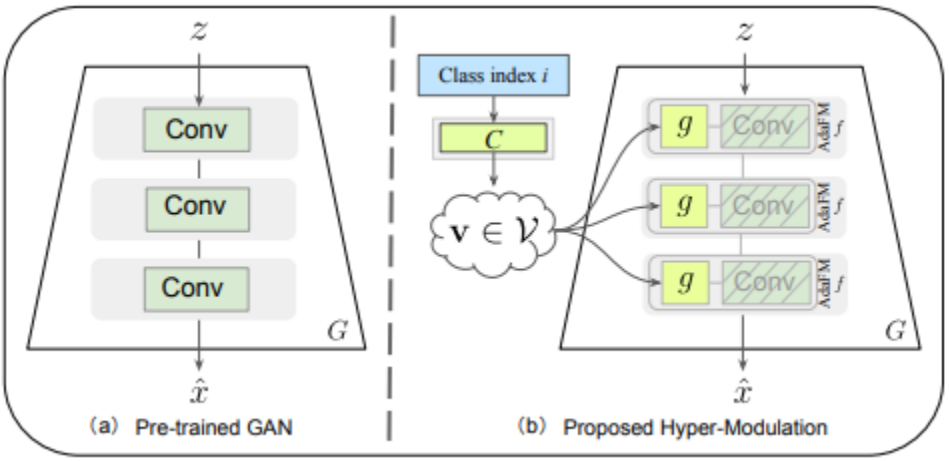

Figure 8: Hyper-modulation architecture [17]

The paper's results are based on 200,002 iterations of training [17]. Given that this project aims to use techniques that facilitate fast and efficient training, an experiment is performed to observe whether high quality results remain after training a model for only a fraction of those iterations.

The training dataset used is the AFHQ dataset, which consists of animal faces divided into three classes of cat, dog, and wildlife. The base unconditional model is a model pre-trained on the FFHQ dataset, which consists of human faces. To observe the

effect of hyper-modulation on a training amount smaller than the paper's 200,002 iterations, the base unconditional model is trained on the AFHQ dataset for approximately 38,000 iterations, which took approximately 57 hours. The interpolation results are shown in Figure 9.

Figure 9: Latent interpolation of classes, cat (left) to wild (middle) to dog (right)

Although FID scores were not calculated, from viewing the resulting interpolation it is clear that the animal classes are distinguishable from left to right as cat, wildlife, and dog, showing that this paper's method was effective. The distortion in the lower rows may not necessarily reflect an issue with the hyper-modulation, but may be recognized as the "blob-like artifacts" commonly found in StyleGAN [26]. These artifacts are addressed

with a fix in StyleGAN2 [22, 26]. It follows that a possible solution is to re-implement the hyper-modulation in StyleGAN2. Regardless, while hyper-modulation appears to be effective in transferring class knowledge, the method involves modification of the StyleGAN architecture itself, and a simpler method may be preferred as rewriting a model to implement unconditional to conditional knowledge transfer may not always be feasible.

2. Direct weight transfer

In a paper on GAN transfer, Wang et al. demonstrated that simply initializing the weights of a conditional GAN by "copying the values from the unconditional GAN" [36] is sufficient to improve the model's training efficiency. Taking inspiration from this, we conduct an experiment where we transfer the desired weights from the pre-trained model's generator into the generator of a fresh conditional model, which will be fine-tuned after receiving the new weights.

To obtain the new and untrained conditional model, the StyleGAN2-ADA code snippet for training a conditional GAN is run briefly to generate the initial model snapshot. The purpose of this is simply to obtain a .pkl file with the structure of a conditional model; the weights are spliced in later. Viewing the contents of the .pkl file confirms the existence of randomly initialized weights of both the generator and the discriminator.

```
print(unpickled_cond_pkl['G'].state_dict()['synthesis.b4.conv1.weight'])

tensor([[[[ 1.8962e+00, -3.8448e-01, -1.7691e+00],
          [-7.9470e-01, -1.0124e+00, -3.7435e-01],
          [-3.6861e-01, -3.1082e-01, -7.8457e-02]],
```

Figure 10: Randomly initialized weights of untrained conditional model generator

```
print(unpickled_cond_pkl['D'].state_dict()['b512.conv0.weight'])
```

```
tensor([[[[ 0.3646, -0.9306,  0.7521],
          [-1.4088,  1.5027,  1.4323],
          [-2.2484,  0.0455,  0.3302]],
```

Figure 11: Randomly initialized weights of untrained conditional model discriminator

The conditional GAN's differing structure is also revealed by the presence of layers that had not existed in an unconditional GAN, as shown in Figure 12.

```
  (b16): DiscriminatorBlock(
    (conv0): Conv2dLayer()
    (conv1): Conv2dLayer()
    (skip): Conv2dLayer()
  )
  (b8): DiscriminatorBlock(
    (conv0): Conv2dLayer()
    (conv1): Conv2dLayer()
    (skip): Conv2dLayer()
  )
  (b4): DiscriminatorEpilogue(
    (mbstd): MinibatchStdLayer()
    (conv): Conv2dLayer()
    (fc): FullyConnectedLayer()
    (out): FullyConnectedLayer()
  )
)
```

```
  (b8): DiscriminatorBlock(
    (conv0): Conv2dLayer()
    (conv1): Conv2dLayer()
    (skip): Conv2dLayer()
  )
  (mapping): MappingNetwork(
    (embed): FullyConnectedLayer()
    (fc0): FullyConnectedLayer()
    (fc1): FullyConnectedLayer()
    (fc2): FullyConnectedLayer()
    (fc3): FullyConnectedLayer()
    (fc4): FullyConnectedLayer()
    (fc5): FullyConnectedLayer()
    (fc6): FullyConnectedLayer()
    (fc7): FullyConnectedLayer()
  )
  (b4): DiscriminatorEpilogue(
    (mbstd): MinibatchStdLayer()
    (conv): Conv2dLayer()
    (fc): FullyConnectedLayer()
    (out): FullyConnectedLayer()
  )
)
```

Figure 12: Unconditional GAN layers (left) and conditional GAN layers (right)

In PyTorch, a state_dict object contains a model's mappings between layers and learnable parameters such as weights and biases [37]. In other words, the state_dict contains the desired weights to be transferred. Thus the next step is to extract the weights from the state_dict of the source generator. However, since the source generator is from an unconditional GAN, the layers of its state_dict do not exactly match the state_dict of the target conditional generator. To bypass this issue, non-matching layers are excluded when creating the filtered dictionary "filtered_dict" as shown in Figure [14].

After filtering the state_dict for non-matching layers, the weights are transferred through loading the source state_dict into the target generator. To confirm the weight transfer, a sample of weights from before and after are printed.

```
print(unpickled_cond_pkl['G'].state_dict()['synthesis.b4.conv1.weight'])

tensor([[[[ 1.8962e+00, -3.8448e-01, -1.7691e+00],
          [-7.9470e-01, -1.0124e+00, -3.7435e-01],
          [-3.6861e-01, -3.1082e-01, -7.8457e-02]],
```

Figure 13: Target conditional model generator weights before update

```
target_dict = unpickled_textures_finetuned['G'].state_dict() # dictionary of target values

# filter out all keys starting with 'mapping'
filtered_dict = {key: val for key, val in target_dict.items() if not key.startswith("mapping")}

# load the new state dict
unpickled_cond_pkl['G'].load_state_dict(filtered_dict, strict=False)

# load new state dict for 'G_ema' too
unpickled_cond_pkl['G_ema'].load_state_dict(filtered_dict, strict=False)
```

Figure 14: Filter out non-matching keys from state_dict to be loaded

```
print(unpickled_cond_pkl['G'].state_dict()['synthesis.b4.conv1.weight'])

tensor([[[[ 1.5431,  2.7893, -0.4025],
          [-1.1014, -2.9943,  3.2309],
          [-0.9261, -0.5723,  0.2979]],
```

Figure 15: Target conditional model generator weights after successful update

The generator, now containing transferred weights, is re-saved as a .pkl file. Using the .pkl file, training is resumed and treated as fine-tuning a conditional model normally. In this model, only the generator's weights have been transferred. Noting that Wang et al. stated the benefit of transferring both the generator and the discriminator, a second model is created, containing transferred weights from both the pre-trained unconditional generator and the discriminator [36]. The same technique of loading the

state_dict was used to transfer weights for the discriminator. Finally, a third model is simply trained from scratch to serve as a base model for comparison. Training for all three models is performed for a duration of 100 kimg.

As a result, the best model is obtained from transferring weights for both the generator ‘G’ and discriminator ‘D’. With a final FID score of 29.01, it outperforms the other models’ final FID scores by a large margin: FID of 196.47 obtained by the model trained on a transferred generator only, and FID of 134.51 obtained by the model trained from scratch. Transferring only the generator weights does not seem to help, and in fact seems to hurt the GAN's training performance compared to the model trained from scratch with default randomly initialized generator and discriminator weights. This is reasonable considering that a GAN ideally trains its generator and discriminator at the same rate, and if one model is more well-trained than the other, then the GAN may encounter difficulty training.

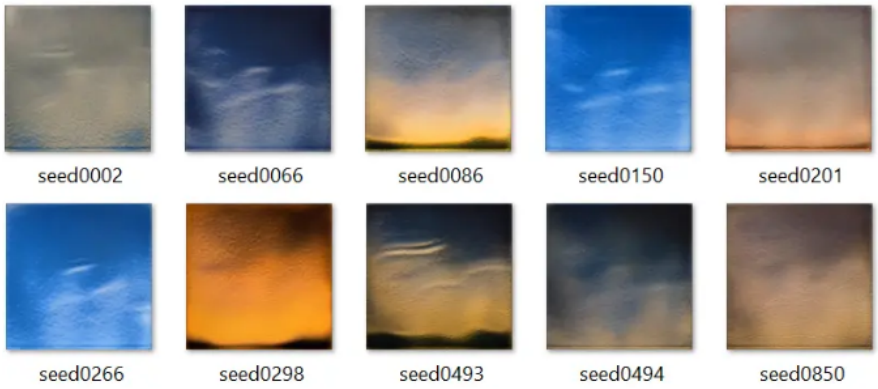

Figure 16: Generated images from cGAN with only generator weights transferred

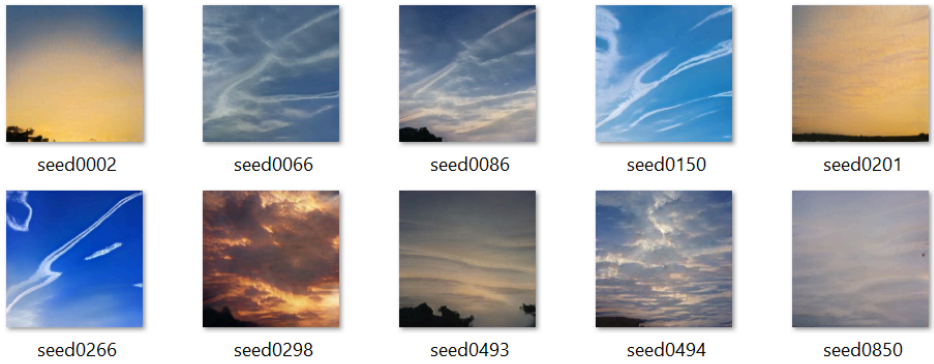

Figure 17: Generated images from cGAN with both G and D weights transferred

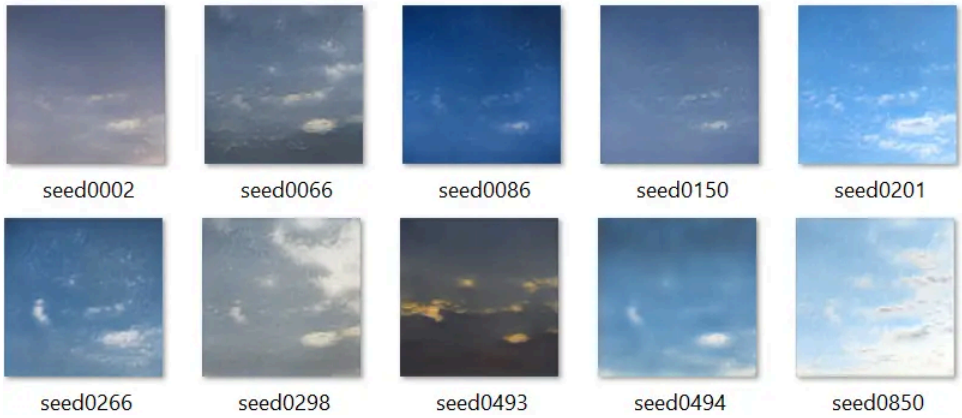

Figure 18: Sample generated images from conditional GAN trained from scratch

In this experiment, there is an untested case of transferring the discriminator weights only. This decision was intentionally made as Wang et al. (2018) suggests that transferring only weights of the discriminator produces inferior results when compared to transferring weights of both the generator and the discriminator, which also produced the best results in their research [36]. With satisfactory results already obtained from transferring both of the generator and discriminator weights, the discriminator-only transfer case is deemed unnecessary.

C - Quantization

Before quantization is applied, the model is first converted into an ONNX format. To convert the model, the generator is loaded into the torch.onnx.export function, which exports the PyTorch generator into an ONNX model. Once the ONNX model is obtained, we test two quantization tools: the ONNX Neural Compressor tool and the quantize_dynamic function provided by ONNX Runtime [38, 39]. We additionally test both quantization tools on a large language model (LLM) to compare the quantization results between an LLM and a GAN.

1. ONNX Neural Compressor Tool:

ONNX Neural Compressor offers weight-only quantization and three different algorithms to achieve it: round-to-nearest (RTN), generalized post-training quantization (GPTQ), and activation-aware weight quantization (AWQ). RTN does not require a calibration dataset while the other two algorithms require a calibration dataset. For simplicity, the RTN algorithm is selected to round the weights to smaller types.

```
from onnx_neural_compressor.quantization import matmul_nbits_quantizer

algo_config = matmul_nbits_quantizer.RTNWeightOnlyQuantConfig()
quant = matmul_nbits_quantizer.MatMulNBitsQuantizer(
    onnx_model,
    n_bits=4,
    block_size=32,
    is_symmetric=True,
    algo_config=algo_config,
)
quant.process()
best_model = quant.model
```

Figure 19: Code implementing ONNX Neural Compressor RTN Weight Quantization

After model quantization, the model size shrunk slightly from 120 megabytes to 117 megabytes. Inference speed was superficially timed on Google Colab using the time() function over several runs, and we found a typical inference time of approximately 2.5 seconds for both the unquantized and quantized models, suggesting no significant difference in inference speed between the two models.

It may be noted that the quantization tool was promoted for large language models (LLMs) [38]. With this in mind, we tested a pre-trained LLM model named "bigbird_Opset16.onnx" that was obtained from an ONNX model zoo [40]. After quantization, we obtained a quantized LLM model size of 486,563 kb from the original size of 498,094 kb. This negligible difference suggests that, even for LLMs, model size is not reduced significantly with round-to-nearest weight-only quantization provided by the ONNX Neural Compressor tool.

2. ONNX quantize_dynamic:

Since we found weight-only RTN quantization to be ineffective in reducing model size for both StyleGAN2-ADA and the LLM model, we tried another quantization tool, ONNX quantize_dynamic provided by ONNX Runtime [39]. When testing it on the ONNX-converted StyleGAN2-ADA model, the model size changed even less from 120,261 kb to 120,058 kb. Additionally, we noticed a warning stating that the model opset does not support node fusions, thus leading to not as optimized performance. To compound the issue, StyleGAN2-ADA model conversion to ONNX format fails when using any model opset > 10. Therefore, ONNX Runtime quantization may not work fully on an ONNX-converted StyleGAN2-ADA model in general.

As for the LLM model, ONNX Runtime's quantize_dynamic function returned much better results than the ONNX Neural Compressor tool, reducing the model size greatly from 498,094 kb to 125,161 kb. Unfortunately, due to the opset issue, the ONNX-converted StyleGAN2-ADA model cannot make optimal use of the ONNX Runtime quantization. Therefore, we keep the StyleGAN2-ADA model quantized using the ONNX Neural Compressor tool.

VI. Web Page Deployment and Results

A - Web Deployment

The model is deployed to a web page through ONNX Runtime Web, a Javascript library that enables ONNX model web application deployment [41]. Output data obtained from model inference is accessed through Javascript. Since the model is conditional, a class should be selected, and class selection is done through a dropdown menu interface that offers a selection of available cloud types. To give an idea of what each selection might generate, sample training images labeled by cloud class are displayed on the right side column. After a cloud type is selected, the user can click the labeled button to start model inference.

The model's generated output consists of numerical data that must be converted into pixel values. Through a conversion formula, the output array values are converted into red, green, and blue (RGB) values, which can be used to display the resulting image on an HTML Canvas. The generated result, being a two-dimensional flat image, must be modified before it can be used as a skybox since a skybox is typically created from either a 6-sided texture or a panoramic image [20, 30]. An issue with directly converting the original image into six smaller images is that the seams must be blended together with image editing software. Figure 20 shows a cubemap that was created by simply cutting out six images from the original image without any seam blending, while Figure 21 highlights sides that must match in a seamless skybox. We considered using HTML Canvas gradients to blend the edges, but simulating the gradient showed poor results as seen in Figure 22.

Click on the boxes to download them.

Cirrostratus Generate Image

Done!

Figure 20: Cubemap directly cut from generated 2D image

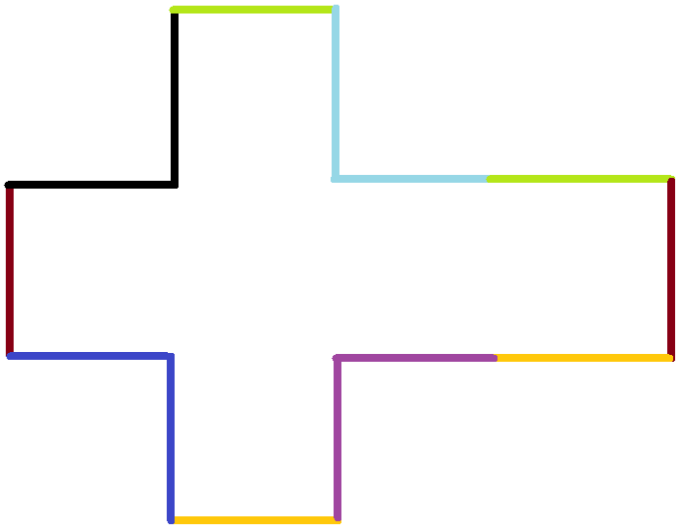

Figure 21: Matching seams (color-coded) of skybox cubemap

Figure 22: Preview of skybox with gradient blending at seams

Therefore, we chose to create a skybox from a panoramic image. A panoramic image must wrap smoothly from left to right. To achieve this, one half of the image was taken and stitched against the same half but flipped, resulting in an image like Figure 23 where the image wraps seamlessly from left to right. This resulting panorama image can be downloaded by clicking on it on the webpage, and then the image can be used in a program such as Unity to create a skybox. Additionally, a 3D preview of the skybox using the generated image can be viewed directly on the webpage by clicking on the A-Frame icon.

Figure 23: Panorama-like image converted from 2D non-panorama image

B - Results

On the web application, a cloud type is first selected from the dropdown menu. Next, the button to generate is clicked. After waiting a few seconds, the web page returns a panorama image as shown in Figure 24. To preview the skybox, the A-Frame button is clicked, and after the skybox is found satisfactory, the panorama image is downloaded and uploaded into a Unity assets folder. A new skybox material is created inside Unity and the panorama image is inserted into the material. The final skybox where the view is fully enclosed in a virtual 3D environment can be previewed inside Unity, as in Figure 25 which shows two different views of the skybox.

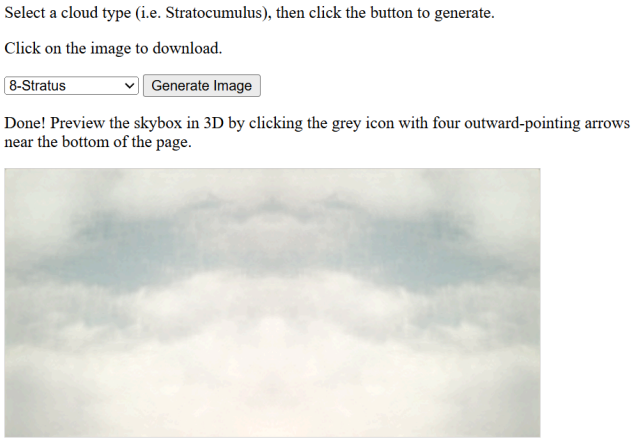

Figure 24: Snapshot of Generated Panorama Image on Web page

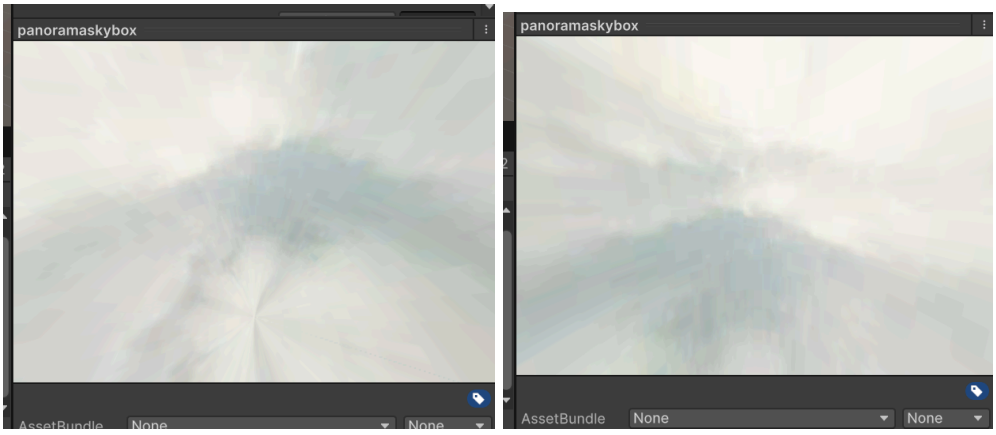

Figure 25: Skybox View in Unity

VII. Conclusion and Future Work

We trained a conditional StyleGAN2-ADA model using borrowed weights from a pre-trained unconditional generator and discriminator. This method achieved an FID score of 29.01 after 100 kimg of training, outperforming the FID score of 134.51 obtained from the model trained from scratch for the same duration. Next, we converted the model into ONNX format and quantized the model using the weight-only round-to-nearest number quantization option provided by the ONNX Neural Compressor tool. This quantization method resulted in a file size reduction of 3 megabytes down from 120 megabytes. Timing the model inference did not show noticeable speedup. Regardless, higher quality tests may need to be utilized to measure inference speed, since the time() function that we used is convenient but unreliable. Additionally, model resource usage was not measured. Ideally, a tool that analyzes CPU or GPU usage would be used to measure the quantized model's precise computation costs. Finally, we deployed the final model to a webpage which converts the model's generated image into a panoramic image that can be downloaded and used to directly create a skybox. The webpage fulfills a use case of choice-based content generation as opposed to the already existing text-based prompt AI skybox generators.

In addition to measuring resource usage of the quantized model, we could further study the quantization itself. The weight-only quantization down to 4-bit weights barely changed the model size, while a different technique called QGAN was able to reduce the model size of older types of GANs like "DCGAN[11], WGAN[34] and LSGAN" by factors ranging from 8 times to 32 times for 1 bit to 4 bit quantizations respectively, according to Tantawy et al. [9]. Further investigation could be done to see whether the difference in model size reduction is from an oversight or other factors related to the StyleGAN2-ADA model itself or even the ONNX neural

compressor tool, especially considering that no studies on quantizing unmodified StyleGAN2-ADA models could be found for result comparison, as well as the neural compressor tool being demonstrated for LLMs rather than GANs.

Currently, the skyboxes produced using the model's generated images appear blurry. Even when an image is pulled from the training dataset to create a skybox, the resulting skybox still appears blurry, meaning the issue cannot be fixed by training the model for more iterations. Instead, the cause of the issue is the small 512 x 512 pixel resolution of the generated images. While it is possible to train a new model on higher resolution images, the training time required grows as resolution increases as shown in Table 3.

Resolution	GPUs	1000 kimg
128x128	1	4h 05m
128x128	2	2h 06m
256x256	1	6h 36m
256x256	2	3h 27m
512x512	1	21h 03m
512x512	2	10h 59m
1024x1024	1	1d 20h
1024x1024	2	23h 09m

Table 3: Expected training times. Adapted from Karras et al. [11]

For even higher image resolutions, the training time will continue to climb according to the trend in the table. Instead of attempting to train a model on higher resolution images, the current 512 x 512 model could be combined with a second ready-to-use model that upscales the image, increasing resolution along with image size. As a result, the larger resolution image will produce a higher quality skybox without needing a new model to be trained.

Currently, the strategy of taking one image half and stitching it with the flipped image half wastes unique pixels on the right half of the image, as illustrated in Figure 26. Ideally, only a smaller stripe of the image is taken from one side, flipped, and blended in with the other side, as shown in Figure 27 where the green lines indicate the flipped section. However, this technique requires precise image editing that may not be viable to do in HTML Canvas. Perhaps an improved formula or another model could be used to transform the image into a panorama image. Alternatively, if a dataset of panoramic images could be obtained, the model could be trained on panoramic images to begin with.

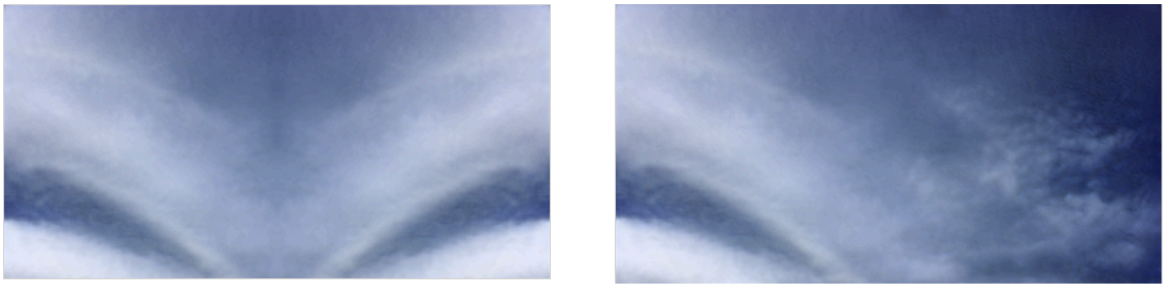

Figure 26: Converted panorama (left) versus original image (right)

Figure 27: Ideal converted panorama (pre-blended)

References

[1] Ramdurai, B., & Adhithya, P. (2023). The impact, advancements and applications of generative AI. *International Journal of Computer Science and Engineering*, *10*(6), 1-8.

[2] Pearson, L. C. (2016). Noclip World: Drawing the disruptions of virtual video game environments. *Drawing: Research, Theory, Practice*, *2*(1), 23-40.

[3] Conway, K. R. (2012). Game mods, engines and architecture. In *Game Mods: design, theory and criticism* (pp. 87-112).

[4] Ahn, S. K. A Review of Efficient Interactive Rendering of Natural Scenes Suitable for VR.

[5] Tinwell, A., Grimshaw, M., Nabi, D. A., & Williams, A. (2011). Facial expression of emotion and perception of the Uncanny Valley in virtual characters. *Computers in Human behavior*, *27*(2), 741-749.

[6] Andreev, P., & Fritzler, A. (2022, August). Quantization of generative adversarial networks for efficient inference: A methodological study. In *2022 26th International Conference on Pattern Recognition* (ICPR) (pp. 2179-2185). IEEE.

[7] Park, S. W., Kim, J. Y., Park, J., Jung, S. H., & Sim, C. B. (2023). How to train your pre-trained GAN models. *Applied Intelligence, 53*(22), 27001-27026.

[8] Frégier, Y., & Gouray, J. B. (2021). Mind2Mind: transfer learning for GANs. In *Geometric Science of Information: 5th International Conference, GSI 2021, Paris, France, July 21–23, 2021, Proceedings 5* (pp. 851-859). Springer International Publishing.

[9] Tantawy, D., Zahran, M., & Wassal, A. (2021). A survey on GAN acceleration using memory compression techniques. *Journal of Engineering and Applied Science*, *68*(1), 47.

[10] Wang, P., Wang, D., Ji, Y., Xie, X., Song, H., Liu, X., ... & Xie, Y. (2019). QGAN: Quantized generative adversarial networks. *arXiv preprint arXiv:1901.08263*.

[11] Karras, T., Aittala, M., Hellsten, J., Laine, S., Lehtinen, J., & Aila, T. (2020). Training generative adversarial networks with limited data. *Advances in neural information processing systems*, 33, 12104-12114

[12] Skybox AI. (n.d.). https://skybox.blockadelabs.com/

[13] *CREATE WITH ROSEBUD AI*. Rosebud AI. (n.d.). https://rosebud.ai/

[14] *Website Traffic Checker*. Semrush. (n.d.). https://www.semrush.com/website/

[15] *Epic Cave on Skybox Ai*. Skybox AI. (n.d.). https://skybox.blockadelabs.com/starter/765d72e8dfb16811efebb61e4053f819

[16] Back, J. (2021). Fine-tuning stylegan2 for cartoon face generation. *arXiv preprint arXiv:2106.12445*.

[17] Laria, H., Wang, Y., van de Weijer, J., & Raducanu, B. (2022). Transferring unconditional to conditional GANs with hyper-modulation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 3840-3849).

[18] *quantize_stylegan2_inc.py* [Computer Software]. (2024). Retrieved from https://github.com/oneapi-src/engineering-design-optimization/blob/main/src/quantize_stylegan2_inc.py

[19] Nobari, A. H., Rashad, M. F., & Ahmed, F. (2021). Creativegan: Editing generative adversarial networks for creative design synthesis. *arXiv preprint arXiv:2103.06242*.

[20] *360° Ultra Realism is Here with Skybox AI Model 3.1*. Blockade Labs. (2024, June 11). https://www.blockadelabs.com/post/360-ultra-realism-is-here-with-skybox-ai-model-3-1

[21] *Running stylegan2-ada in browser*. Guido & AI. (n.d.). https://www.guidodejong.nl/hack/running-stylegan2-ada-in-browser/

[22] Karras, T., Laine, S., & Aila, T. (2019). A style-based generator architecture for generative adversarial networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition* (pp. 4401-4410).

[23] Creswell, A., White, T., Dumoulin, V., Arulkumaran, K., Sengupta, B., & Bharath, A. A. (2018). Generative adversarial networks: An overview. *IEEE signal processing magazine*, *35*(1), 53-65.

[24] Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... & Bengio, Y. (2014). Generative adversarial nets. *Advances in neural information processing systems*, *27*.

[25] Huang, X., & Belongie, S. (2017). Arbitrary style transfer in real-time with adaptive instance normalization. In *Proceedings of the IEEE international conference on computer vision* (pp. 1501-1510).

[26] Karras, T., Laine, S., Aittala, M., Hellsten, J., Lehtinen, J., & Aila, T. (2020). Analyzing and improving the image quality of stylegan. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition* (pp. 8110-8119).

[27] Yamashita, R., Nishio, M., Do, R. K. G., & Togashi, K. (2018). Convolutional neural networks: an overview and application in radiology. *Insights into imaging*, *9*, 611-629.

[28] *Quantization*. Quantization - PyTorch 2.7 documentation. (n.d.). https://pytorch.org/docs/stable/quantization.html

[29] Jajal, P., Jiang, W., Tewari, A., Kocinare, E., Woo, J., Sarraf, A., ... & Davis, J. C. (2024, September). Interoperability in deep learning: A user survey and failure analysis of onnx model converters. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 1466-1478).

[30] *Create a cubemap*. Unity. (n.d.). https://docs.unity3d.com/Manual/class-Cubemap-create.html

[31] ClassiCube Skyboxes. (n.d.). https://skyboxes.weebly.com/

[32] *Real-time 3D development platform & editor*. Unity. (n.d.). https://unity.com/products/unity-engine

[33] Liu, Pu. (2019). *Cirrus Cumulus Stratus Nimbus (CCSN) Database* (Harvard Dataverse; Version V2) [Data set]. Harvard Dataverse. https://doi.org/10.7910/DVN/CADDPD

[34] *StyleGAN2-ADA PyTorch multiclass labels* [Computer Software]. (2021). Retrieved from https://github.com/JulianPinzaru/stylegan2-ada-pytorch-multiclass-labels

[35] Pinkney, J. N. M. *Awesome pretrained StyleGAN2* [Data set]

[36] Wang, Y., Wu, C., Herranz, L., Van de Weijer, J., Gonzalez-Garcia, A., & Raducanu, B. (2018). Transferring gans: generating images from limited data. In *Proceedings of the European conference on computer vision (ECCV)* (pp. 218-234).

[37] *What is a state_dict in PyTorch*. What is a state_dict in PyTorch - PyTorch Tutorials 2.5.0+cu124 documentation. (n.d.). https://pytorch.org/tutorials/recipes/recipes/what_is_state_dict.html

[38] *Onnx/neural-compressor: Model compression for ONNX* [Software]. GitHub. https://github.com/onnx/neural-compressor

[39] *Quantize ONNX Models.* ONNX Runtime. https://onnxruntime.ai/docs/performance/model-optimizations/quantization.html

[40] *Onnx/models: ONNX Model Zoo* [Software]. GitHub. https://github.com/onnx/models

[41] Onnx. (n.d.). *ONNX Runtime Web*. NPM. https://www.npmjs.com/package/onnxruntime-web