Transfer Learning and Optimization for Sky Image GAN Deployment in Unity

Project Report

Presented to

Dr. Chris Pollett

Department of Computer Science

San Jose State University

In Partial Fulfillment

Of the Requirements for the Class

CS 297

By

Crystal Kwong

December 2024

ABSTRACT

Entertainment media has been ubiquitous since the invention of screens, and generative artificial intelligence (AI) adds to the possibilities. GANs, short for generative adversarial networks, are AI models that can create media including text and images. Yet, GANs are computationally expensive in both training and inference, making the deployment of GANs difficult. Fortunately, there exist techniques to reduce the size and training times of AI models. This project aims to explore those techniques on a GAN and then deploy the model in a game engine such as Unity to efficiently generate skyboxes on the fly. To achieve this, four deliverables were completed this semester. The first deliverable sets up an environment for running a pre-trained unconditional GAN. This lays the groundwork for the second deliverable, which involves fine-tuning that pre-trained model and comparing the results against a model trained from scratch. The third deliverable covers transfer learning from an unconditional GAN to a conditional GAN. The fourth deliverable experiments with simple quantization of a GAN model, focusing on manual modification of StyleGAN2-ADA weight values. With the completion of these deliverables this semester, a fine-tuned GAN model for generating weather images has been built, and knowledge for the next steps of unconditional to conditional transfer and quantization has been gained.

Keywords - *Artificial Intelligence (AI), GANs (Generative adversarial networks), Transfer Learning, Fine-tuning, Quantization*

I. INTRODUCTION

Much attention is focused towards making media entertainment, and the rise of artificial intelligence introduces new possibilities to the process. Media content like images or videos is easily generated by generative AI models, and StyleGAN2-ADA is one such model capable of generating high quality realistic images [1]. As for generating content in games, the content is typically pre-generated outside of the game engine and then converted into a file to be read by the engine for use. Real-time content generation, on the other hand, is limited due to the slow speed of GANs. Despite this, utilizing neural network models for real-time generation has not been abandoned: for example, real-time style transfer to generate texture images has been implemented in the Unity engine using Barracuda, a tool to execute pre-trained models [2]. Similarly, this project aims to deploy a GAN in Unity to generate images in real time. This project will experiment on quantization techniques to reduce model size and improve inference speed, as well as transfer learning techniques to minimize training needed while maintaining satisfactory image quality. With these techniques, especially quantization, efficiency is expected at the expense of quality; then, to minimize the effect on perceived image quality, a more abstract and forgiving subject for generated images may be preferred, such as weather. Thus, this project aims to train a StyleGAN2-ADA model on weather images. The final model will be deployed in a Unity engine to generate in real time skyboxes usable in entertainment media like games or movies.

In preparation for the final project, four deliverables have been completed this semester. Each section in this paper covers the deliverable's purpose and implementation. The first deliverable sets up a pre-trained GAN to be experimented on in subsequent deliverables. The second deliverable fine-tunes the pre-trained GAN and additionally compares the results against

a GAN trained from scratch. The third deliverable covers transfer learning from unconditional to conditional GAN models. Finally, the fourth deliverable discusses quantization as a technique to reduce model size, and demonstrates an attempt at basic quantization on a GAN by manually converting StyleGAN2-ADA weights. The conclusion reviews the knowledge gained from this semester and the next steps for this project.

II. DELIVERABLE ONE: SETTING UP PRE-TRAINED GAN

Training a GAN from scratch requires enormous amounts of data and time, as backed up by the training times posted on the StyleGAN2-ADA page [1], [3]. According to that page, the time required to train a model from scratch can range from hours to days depending on desired quality of the model. For example, the time taken to train a GAN for 1024x1024 resolution images on one GPU for 25,000 kimg, where "kimg" means a thousand images shown to the discriminator, is listed as over 46 days. Fortunately, there exist pre-trained GANs that are ready to use without additional training. The purpose of this deliverable is to set up an environment for running a pre-trained StyleGAN2-ADA model, which will eventually be modified into the final desired model.

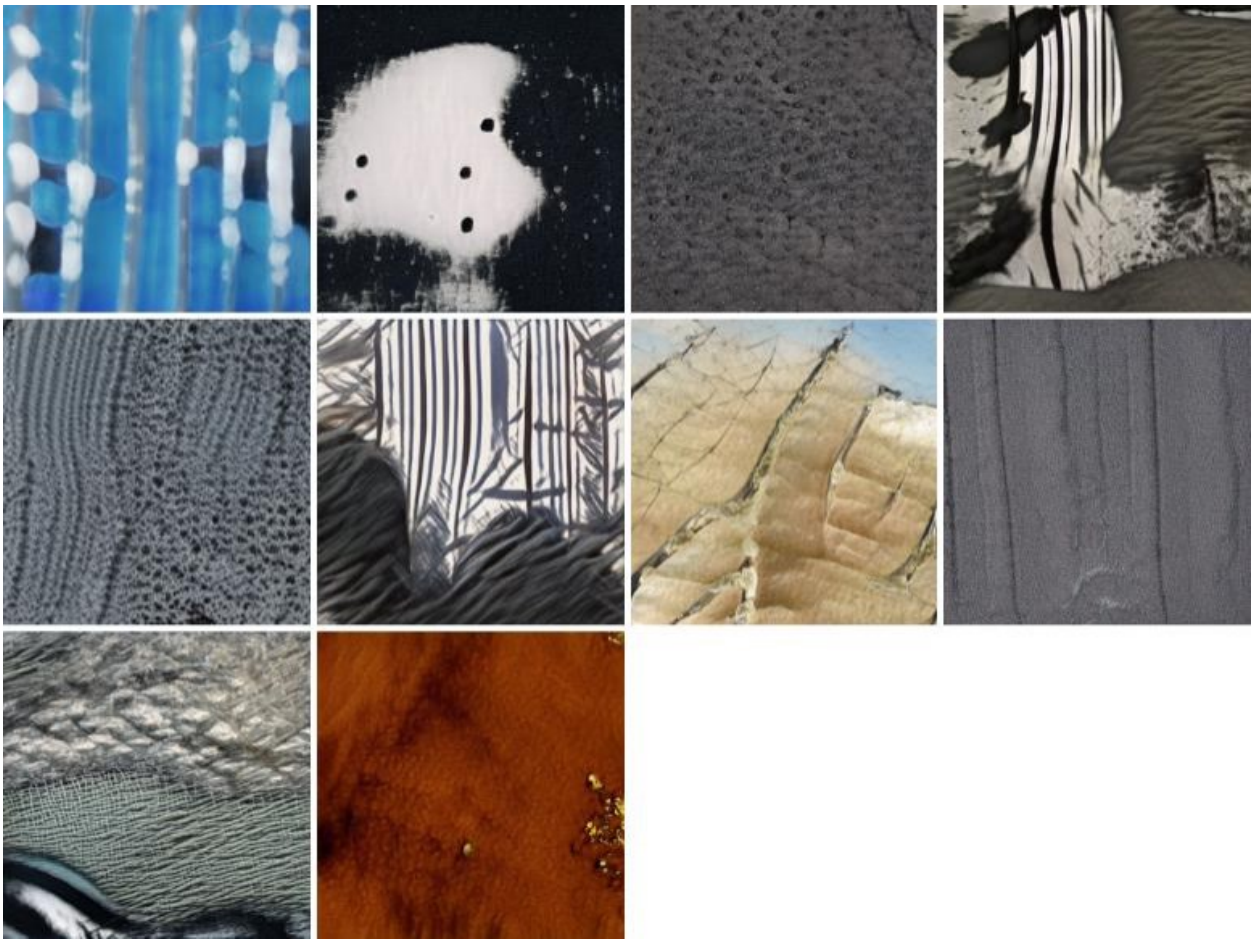
The pre-trained StyleGAN2-ADA model is obtained from Justin Pinkney's pretrained StyleGAN2 model collection [4]. For each model, sample generated images are provided, ranging from images of cars to human faces. The 'textures' model in particular produces output resembling sky images most closely. Considering that domain similarity may be useful in fine-tuning experiments, as suggested by the paper on fine-tuning cartoon faces [5], the 'textures' model is selected for this deliverable.

The selected model is downloaded into Google Drive as a .pkl file. Next, a Google Colab notebook is created. After the notebook is verified to be running on GPU and using Python 3.10.12 with PyTorch 2.4.1+cu121, the next steps are to download required Python libraries and clone the StyleGAN2-ADA repository. Once the environment is set up, generate.py is called with an input set of seed values to generate a unique image for each seed value. The time taken to generate ten images was approximately 1 minute. The function call and generated images are

shown below.

```
[12] !python generate.py --outdir=out --trunc=1 --seeds=1,65,85,149,200,265,297,490,491,849 \
        --network=textures.pkl
```

```
Loading networks from "textures.pkl"...
Generating image for seed 1 (0/10) ...
Setting up PyTorch plugin "bias_act_plugin"... /usr/local/lib/python3.10/dist-packages/torch/utils/cpp_extension.p
If this is not desired, please set os.environ['TORCH_CUDA_ARCH_LIST'].
  warnings.warn(
Done.
Setting up PyTorch plugin "upfirdn2d_plugin"... /usr/local/lib/python3.10/dist-packages/torch/utils/cpp_extension.
If this is not desired, please set os.environ['TORCH_CUDA_ARCH_LIST'].
  warnings.warn(
Done.
Generating image for seed 65 (1/10) ...
Generating image for seed 85 (2/10) ...
Generating image for seed 149 (3/10) ...
Generating image for seed 200 (4/10) ...
Generating image for seed 265 (5/10) ...
Generating image for seed 297 (6/10) ...
Generating image for seed 490 (7/10) ...
Generating image for seed 491 (8/10) ...
Generating image for seed 849 (9/10) ...
```



III. DELIVERABLE TWO: FINE-TUNING A PRE-TRAINED GAN

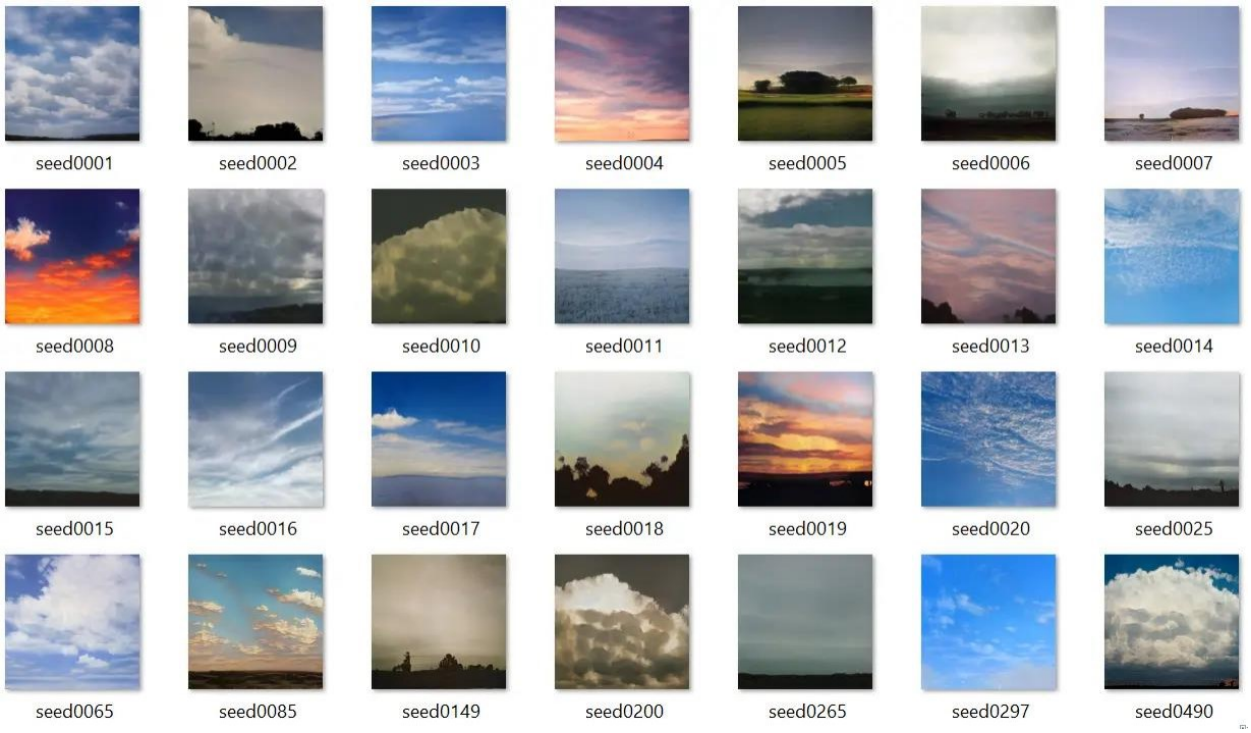
The pre-trained model outputs texture images, but for this project the desired output is a set of sky images. To transform the model's output, fine-tuning can be utilized. Fine-tuning is a transfer learning technique that tunes a pre-trained model's weights in order to modify its generated output, and in this deliverable, fine-tuning is applied to adapt the pre-trained model's output from texture images to sky images.

Training a GAN is very time-consuming as stated earlier in the first deliverable. To avoid session length constraints on Google Colab, the high performance computing (HPC) cluster is chosen as the new model environment. To begin using the model in the new environment, the pre-trained model file is uploaded to the HPC. Inside a conda virtual environment, Python version is verified to be 3.9.20, and required modules and library paths are set. Additionally, a pre-built PyTorch 1.7.1 binary compatible with CUDA compute capability 3.5 is installed to match the HPC GPU's compute capability [6].

After installing the prebuilt PyTorch binary, the model is ready to be fine-tuned. The function train.py is called using parameters of two GPUs and kimg = 100, where 'kimg' measures thousands of images shown to the discriminator during training. The training dataset used is the Cirrus Cumulus Stratus Nimbus (CCSN) dataset, containing 2543 total images of clouds with 11 different cloud categories [7]. Before training, the dataset is prepared using StyleGAN2-ADA's dataset tool function. A small sample of two classes from the dataset, where

each row represents one class of clouds, is shown below.

After training over twenty-four hours, a set of sample images can be viewed.



These images clearly resemble skies and clouds at a glance. Rather than relying on eyesight however, a more tangible metric to measure image quality with may be preferred. FID score is a metric that measures quality and diversity of images generated by a GAN [3]. A lower FID score means that the GAN generates more diverse and higher quality images, while a perfect FID score of 0 means that the set of images generated are identical to the images in the dataset

[3]. During training, the fine-tuned model's initial FID score of 265.07 rapidly drops down to 62.21 in the next 20 kimg of training. At 100 kimg trained, a FID score of 25.55 is reached. The decreasing FID score suggests that image quality has improved greatly. To provide a base case for comparison, a model is trained from scratch on the same dataset using the same parameters. Sample images generated by this model trained from scratch are shown below.

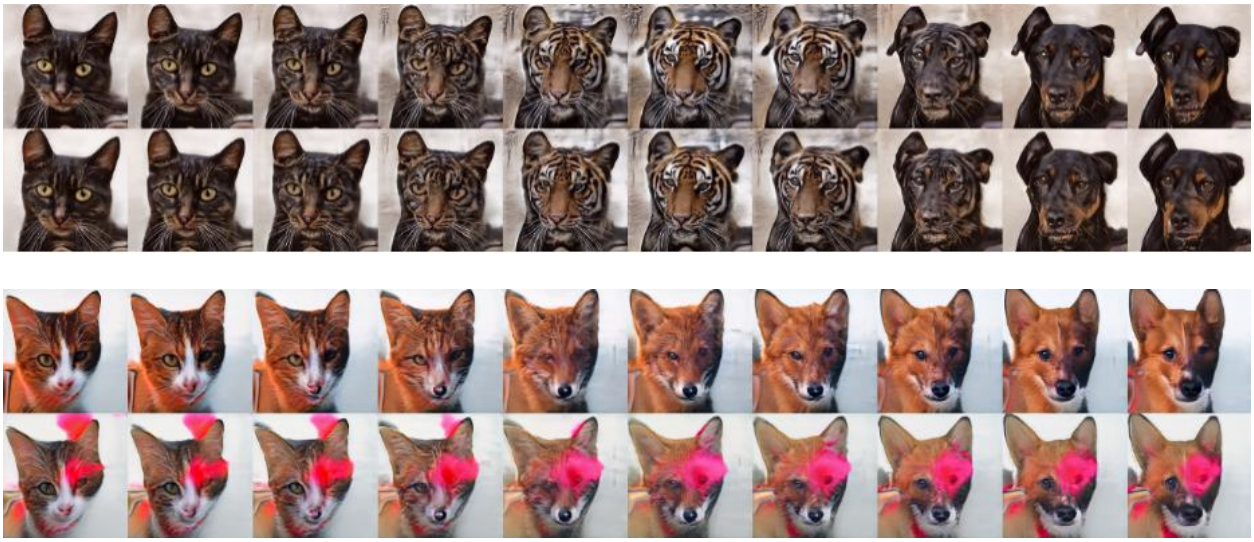
The quality appears grainy and lower than that of the images generated by the fine-tuned model. Looking at FID scores, the model trained from scratch has an initial FID score of 331.11 that drops to 198.09 after 100 kimg of training. This FID score is much higher than the final FID score of 25.55 achieved by the fine-tuned model, showing that the model trained from scratch indeed produces lower quality images than the fine-tuned model. This supports the idea that a fine-tuned pretrained model converges faster than a model trained from scratch.

IV. DELIVERABLE THREE: TRANSFER LEARNING FROM UNCONDITIONAL TO CONDITIONAL GAN

While unconditional GANs provide no control over generated output, conditional GANs do provide control by allowing generation of images by class label, making conditional GANs more useful where a particular output is desired [8]. In this project, generating sequences of non-random weather images can lend to more realistic weather depictions, making a conditional GAN preferred for this project. However, since the current fine-tuned model is an unconditional model, it cannot be trained as a conditional model with the current architecture of StyleGAN2-ADA. Alternatively, a pretrained conditional GAN could be obtained and then fine-tuned into the desired conditional model; however, pretrained conditional GANs are less common than pretrained unconditional GANs [8]. Therefore, simply finding a pre-trained conditional GAN may not be viable [8]. Instead, transfer learning from pre-trained unconditional GANs to conditional GANs may be explored. This deliverable examines the effectiveness of unconditional to conditional transfer on a modified version of StyleGAN. Quality results may justify modifying the StyleGAN2-ADA architecture to support unconditional-to-conditional weight transfer.

The first step is to download the pretrained unconditional StyleGAN model provided by the Hyper-Modulation GitHub repository [8]. This model is pre-trained on the FFHQ dataset, a dataset of human faces. The next step is to download the AFHQ dataset, a dataset of animal faces divided into three classes. This dataset will be used to train the conditional model. Finally, the Hyper-Modulation GitHub repository is cloned, and training is begun with the recommended configuration of 200,002 iterations, although the training is ended early at 38,000 iterations in order to conserve time. The total time elapsed for training over 38,000 iterations was approximately 57 hours. Using the model checkpoint, a latent interpolation is generated. A

snippet of the latent interpolation is shown below.



The different classes can be distinguished. However, the presence of an artifact is obvious. The StyleGAN2 page mentions the removal of this artifact from StyleGAN through modification of the normalization [1]. Perhaps with implementation of unconditional-to-conditional transfer in StyleGAN2-ADA, the artifact may disappear and image quality will improve.

V. DELIVERABLE FOUR: GAN QUANTIZATION

While transfer learning can help speed up training, models can still take up huge amounts of space after training. Quantization is a technique to reduce model size and increase inference speed [9]. Given this project's goal of deploying a model to generate images on the fly, fast inference speeds should be prioritized.

The initial goal of this deliverable was to quantize a StyleGAN2-ADA model. From the GAN quantization paper, the simplest form of quantization was defined as transforming weights from a float representation into an integer representation [9]. Quantization API is also provided by PyTorch; however, it was not successfully applied to StyleGAN2-ADA in this deliverable. As an alternative, an attempt is made to modify the weights directly in the StyleGAN2-ADA model for this deliverable.

A Google Colab environment is set up with Python 3.10.12 and Pytorch 2.4.1+cu121, and a StyleGAN2-ADA pretrained model file in either .pt or .pkl format is downloaded in Google Drive. Code for all classes contained inside StyleGAN2-ADA's 'networks.py' file is duplicated inside the Google Colab notebook in order to allow creation of a StyleGAN2-ADA generator instance, which will be used later in this deliverable.

Before modifying the weights directly, StyleGAN2-ADA's weight storage method should be understood. Inside the load_network_pkl function, a code snippet shows that StyleGAN2-ADA saves instances of 'Module' inside its .pkl file [1]. Then, those instances can be accessed to view the model weights: according to the official PyTorch source, each module has a state_dict, which is the mapping of layers to learnable parameters [10].

```
# Validate contents.
assert isinstance(data['G'], torch.nn.Module)
assert isinstance(data['D'], torch.nn.Module)
assert isinstance(data['G_ema'], torch.nn.Module)
assert isinstance(data['training_set_kwargs'], (dict, type(None)))
assert isinstance(data['augment_pipe'], (torch.nn.Module, type(None)))
```

With accessible StyleGAN2-ADA weights, the next goal is to manually change the weight from float32 to int8 representation. An experiment with conversion of a singular tensor is performed before attempting to convert every weight in all layers of the generator model. After conversion of a singular tensor is successful, the same conversion from float32 to int8 is performed for every weight in the original model. Those modified weights are saved in a newly initialized state_dict, which is then loaded into a new generator instance. To verify quantization, the generator's state_dict is printed. However, the state_dict unexpectedly outputs floats. A cropped output is shown in the code snippet below.

```
print(generator_int8.state_dict())
[[ 1., -2., 0.], [ 0., 0., 0.], [ 0., 0., 2.]], ...
```

Despite this issue, image generation is attempted using the modified generator. With modified float weights, the difference in generated image quality is expected to change. To achieve image generation, the quantized generator instance is first saved into a copy of the original StyleGAN2-ADA .pt file, replacing the previous generator module. Then the .pt file is converted into a .pkl file readable by StyleGAN2-ADA. Finally, generate.py is called using this modified.pkl file, but an error is returned, stating "Can't get attribute 'Generator' occurred." During attempted debugging, comparison between print outputs of the original loaded model and the modified model seemed to yield identical results, so the .pkl file structure did not seem to be an issue, but it will not be ruled out.

Ultimately, the results of the weight modification were not obtained, but StyleGAN2-ADA weights were successfully modified directly. As for model quantization, the method utilized was a very simplistic method of quantization by simply converting weights from a higher precision to lower precision representation. The PyTorch quantization API provides more complete versions of quantization which may quantize activations as well as weights. A future plan could be to further study the PyTorch quantization API and then attempt a second time to apply it to StyleGAN2-ADA.

VI. CONCLUSION

With these deliverables completed, a GAN model fine-tuned for generating weather images has been obtained. Additionally, insight has been gained on model optimization techniques like unconditional to conditional knowledge transfer and quantization. The first deliverable introduces using GAN models by setting up a pre-trained GAN model and generating sample images with it. The second deliverable follows up by fine-tuning the pre-trained model into generating the desired weather images. Through fine-tuning experiments, the second deliverable also demonstrates that better image quality is reached in less time with fine-tuning as compared to training from scratch. Therefore, fine-tuning may be used to save time in training the final model for this project. The third and fourth deliverables cover model optimization techniques to further improve the model. Control over generated output, given a pre-trained unconditional GAN, is explored in the third deliverable as another form of transfer learning. Finally, the fourth deliverable explores quantization to reduce model computation load and size, which will ease deployment of the final model. Although GAN quantization was not implemented, manual modification of model weights was performed successfully. Alongside modifying the weights, knowledge of weight storage and model file format was also gained, which may be useful in code implementation.

The next semester aims to build on the current fine-tuned model by adding control over generated output given an unconditional pre-trained GAN. Being able to control the generated output is desirable because it enables customized weather transitions rather than leaving generation completely up to chance. Next, the project looks to quantize the model and compare its size and inference speed against an unquantized model. The final model will be deployed in a Unity engine where it will be used to generate custom weather skyboxes. The goal is to produce

a lightweight and easily trained model capable of generating convincing and appealing weather backgrounds to enhance scenes.

REFERENCES

[1] 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

[2] Park, H., & Baek, N. (2024). Design and Implementation of Style-Transfer Operations in a Game Engine. *International Journal of Advanced Computer Science & Applications, 15*(8).

[3] 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.

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

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

[6] Liu, Nelson. pytorch-manylinux-binaries [GitHub dataset]

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

[8] 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).

[9] 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.

[10] *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