CSCE 479/879 Homework 3: [Generative Models]

[Matthew Penne, Mason Lien, Mrinal Rawool, Yves Shamavu]

April 3, 2021

Abstract

In this project, we build a model that is trained on 162,770 celebrity faces to generate artificial faces of nonexistent people. This can be achieved using various techniques such as Variational Autoencoders (VAE), Generative Adversarial Networks or GANs, PixelCNNs, etc. The task requires images to be generated from random noise with no extract input dimensions. Thus, the models we experimented with generate images through learned latent vectors on a fixed 512-dimensional hypersphere. The group tried Variational Autoencoders, Info GANs, and Deep Convolutional GANs (DCGANs). However, we ultimately settled on Progressive Growing of GANs adapted from [3]. Our adaptation starts from a 16×16 model and progressively grows up to a 256×256 model, resizing the output to $218 \times 178 \times 3$ from the 32×32 GAN onwards. After training the 128×128 model the binary cross-entropy loss was 0.3141 for the discriminator loss and 5.5596 for the generator loss. The first training epoch for the 256x256 model is still in progress as of this reporting. Its cross-entropy loss is hovering around 0.2523 for the discriminator and 6.4841 for the generator as described in the report.

1 Introduction

A Generative Adversarial Network or GAN is a combination of a generator - discriminator network. During training, a GAN learns characteristics of the training data and generates new data using the same. The training happens in two phases. In the first phase, the discriminator is trained by feeding it a mix of real and synthesized data that is labeled. Error generated due to misclassification is used to update the weights of the discriminator. Once the phase 1 for training the discriminator is complete , its weights are frozen and phase 2 begins for training the generator. In phase 2, the generator generates images that are fed to the discriminator to be classified as real or fake. The goal for the generator is to create synthetic images that appear to be sampled from the original training distribution, thereby making the discriminator label them as real. Error is generated when the discriminator is able to successfully tell the real images apart from the fake ones.

A Progressive Growing GAN is described in [3]. The GAN trains on low resolution data data starting at 4×4, then it transfers the weights on to a model with the resolution of 8×8 and retrains the model. This process continues until an image dimension of 256×256 in this project or 1024×1024 for the CelebA example used in [3]. An InfoGAN is described in [1]. Normally, a GAN is a black box method whose image generation cannot be manipulated in a determined way. An InfoGAN develops latent variables that can slightly change the image. For example, changing the latent variables in an InfoGAN trained on the CelebA dataset can change the angle the face is looking, the lighting, and the facial structure by changing the latent variables. Additionally, a basic variational autoencoder (VAE) based on [2] was trained on the dataset for preliminary analysis during model exploration phase. A VAE is a type of probabilistic autoencoder that uses a latent representation that is sampled from a Gaussian distribution with a mean μ and standard deviation σ. The latter quantities are generated by the encoder and used while calculating the latent loss for the model.

2 Problem Description

CelebFaces Attributes Dataset (CelebA) [5] is a dataset of celebrity faces with several different categories. It contains around 200K celebrity images with 40 attribute annotations. This dataset has previously been used for various computer vision tasks such as image recognition, attribute recognition, landmark recognition, face synthesis etc. The task is to train a model that can generate images with similar characteristics as the training set with no external input from the user. One way of achieving this objective is to train the image generator to create high resolution images using low dimensional Gaussian noise as the input. In the next section, we discuss the various approaches that our team discussed and the final approach that the team decided on pursuing.

3 Approaches

The group tried generating images with Progressive Growing GANs (PGGAN), InfoGANs, and Variable Autoencoders. Ultimately, the Progressive Growing GANs produced the best results and in the correct form for the submission.

3.1 PGGANs

A new training methodology is described in [3] for Generative Adversarial Networks where their primary contribution is to start with low-resolution images, and then progressively increase the resolution by adding layers to the networks. This type of workflow allows the training of large-scale structures of the image distribution and then progressively directs attention to finer details. This approach is different from traditional methods where all levels of scales are learned simultaneously.

Their approach simplifies the generation of images and produces stability due to less class information to handle. It sequentially increases the resolution over time instead of mapping the entire 1024x1024 latent vectors; ultimately reduces training time since most iterations are done at lower resolutions. Related studies that have followed similar methods of growing GANs progressively have investigated how the number of generators, discriminators, and the use of layering (hierarchical GANs) directly affects the ability of the model to map from latents to high-resolution images. [3] did not use a hierarchical approach. Rather, they used a single GAN with a layer-wise training of auotoencoders similar to [4]. For this homework we chose to use [3] approach, which is further explained in the experimental setup.

3.2 Info GAN

While a PGGAN and other GANs can produce photo realistic images, they offer little to no control over the image produced. An InfoGAN detailed in [1]. This model uses continuous control variable to change the style of the image. The control variables are normally generated from -1 to 1. The control variable gradually change an aspect about the produced image. For the CelebA produced image, the facial angle or orientation, the hair style and color, the facial structure can all be changed by varying the latent variables. The group decided to implement the InfoGAN in Matlab and try to export the model to Tensorflow. In previous assignments, the group's experiments with Matlab had proven that using Matlab decreased the training time when compared to Tensorflow. Since image generation models generally take significant amounts of time to train, the group decided to explore some Matlab based models to get a better idea of the relative time and performance factors. Listed below are some features of InfoGAN that the group implemented:

1. A preprocessing layer normalizes the input data and crops each image to $64 \times 64 \times 3$.

2. The generator uses 4 transposed convolutional layers with 1024-512-256-128 filters. Batch normalization is used between each layer after the first transposed convolutional layer. All layers use ReLu activation except the last layer that uses hyperbolic tangent activation.

3. The discriminator uses 4 convolutional layers with 128-256-512-1024 filters. Again batch normalization is used between each layer after the first transposed convolutional layer. All layers use ReLu activation except the last layer that has no activation function.

The model is trained for 5 epochs with a batch size of 16 on a GTX1070. The training took 664 minutes to complete, or 133 minutes per epoch. The resulting images are shown in 1. New images are generated along the Y-axis while a latent variable is varied from -2 to 2 for the images along the X-axis. The lower the value of the latent variable, the more the persons' hair appears to blend into

Figure 1: Images Generated from the Matlab InfoGAN

the background. With higher values of the latent variable, the artifacts around the persons' hair become less prominent.

The goal of using Matlab instead of Python is that the training and exporting of a model from Matlab to Python was expected to faster than just training the model in Python. The Matlab function *exportONNXnetwork()* is used to export the model to the ONNX format, which can then be imported to Tensorflow. Attempts to run the imported model in Tensorflow were unsuccessful because of the difference in the model formats across the two platforms. The exported model was in Number-Channel-Height-Width format; while Tensorflow only supports the Number-Height-Width-Channel format. Converting from one format to the other is its own research problem and has no clear solution. This error was encountered in the group's homework one assignment, but was felt could be over come because the group built the Matlab model with the input format of number-height-width-channel. It appears to be a problem within Matlab and time constraints forced the group to stick with the PGGAN model.

3.3 Variational Autoencoder

The variational autoencoder model was based on [2] with some minor variations to adapt it to the CelebA dataset. The highlights of this approach was

1. Using a preprocessing layer to convert pixel values from the range 0 to 255 to floats in the range 0 to 1.
2. The encoder uses Dense layers with 'selu' activations while the last layer uses 'tanh' activation.
3. Batch Normalization layers were added after the first layer to assist training

4. The loss is calculated using binary cross-entropy as well as latent loss

Training this model proved to be challenging for multiple reasons. The estimated time for one epoch was around 5 hours on Quadro RTX 5000. Initial versions of the model were unable to calculate the loss. Batch normalization was added to remedy this issue. However, the model did not seem to learn as the losses kept on increasing despite of tweaking the number of layers, coding size, learning rate, optimizers etc. The instabilities encountered early on during exploration led to the team pursuing PGGANs as the model of choice for this assignment.

4 Experimental Setup

The code for implementing our PGGANs model has been adapted from [3]. The highlights of this approach over a normal GAN are

1. Fading in New Layers: PGGANs employ fading technique to seamlessly merge the previous output with the current output both on the generator and discriminator side. The generator applies upsampling to the previous generator version's output while the discriminator applies down sampling.

2. Mini-batch Standard Deviation: This layer is used at the output of the discriminator to gauge the diversity of generator outputs. Mode collapse occurs when the generator starts producing outputs with similar features. Usually, this mode is the one that can optimally confuse the discriminator. However, such a phenomenon occurs when the discriminator is under-trained and thus not learning anything. This allows the generators to over-fit the current version of discriminator. In mini-batch standard deviation, standard deviations across all channels, all instances in a batch are calculated and averaged across all points to arrive at a single value. Thus, if the diversity in generator output for a batch is low, the resulting averaged standard deviation would also be low across feature maps. This statistic can help the discriminator recognize the symptoms of mode collapse and provide feedback to the generator accordingly.

3. Equalized learning rate: This is a type of normalization technique used to ensure all layers learn at the same speed thereby stabilizing training. This is achieved by scaling the weights for all layers.

4. Pixel wise normalization: Normalizing the pixel values in the same image and location across channels to values between 0 and 1. This is a type of normalization technique used to avoid explosive activation that may arise due to the competition between generator and discriminator. It uses a small factor ϵ to avoid division by zero while scaling. Pixel wise normalization is used after every 2D convolution layer.

The training technique used for our model was adapted from the example of training a normal GAN provided in the course textbook [2]. A major tweak

was to leave all layers, from both the discriminator and the generator, trainable at each training step. For normal GANs, the discriminator's trainable attribute is set to false when the generator is being trained. However, with progressive growing GANs this is not needed because of a smooth transition from already trained (lower-resolution) layers to higher resolution layers. This helps prevent a sudden shock to the already well-trained weights of previous layers.

4.1 The Building Blocks

Listed below are the basic building blocks used by the model. During the course of experimenting, it was identified that the training could be sped up by starting with the 16×16 block instead of 4×4 block used in [3]. Furthermore, we decided to halt the training when the output size reached 256×256 as the training speed significantly slows down at higher resolutions.

1. The generator and discriminator are mirrored with each other. They are built using basic building blocks that start with an output size of 16×16 and go on till 256×256.

2. The generator blocks starting from 16×16 to 256×256 employ the generator initialization block and up sampling blocks that are called through the fade-in block to generate a model with a particular size.

3. Generator Initialization block (see gen_init_block): This block is called at every progressive layer to create a multidimensional artifact of size $4 \times 4 \times 512$ using a noise input of dimension 512. Additionally, it initializes the weight factor α that decides the contribution of the generated artifact to the output at the current stage.

4. Upsampling block (see upsample_block): This block up samples the current input by a factor of two. This block employs learning rate equalization and pixel wise normalization.

5. Fade-in blocks: The fade-in block employs upsampling block, learning rate equalization, and pixel normalization to fade in the newly added layer gradually with the previously existing layer. Mode collapse happens when the generated images by the generator lack variety. Although this issue is still inherent to GANs, PGGANs tend to do reasonably well, thanks to pixel normalization layers [3].

6. Model Generation: This block accepts the RGB output of the previous layer, RGB output of the newly created upsampled layer, and the weight factor α and combines them to provide the output of a particular size (which is the size of the upsampled output.

7. The structure of the discriminator and of the generator is shown in 2 for a 16x16 model.

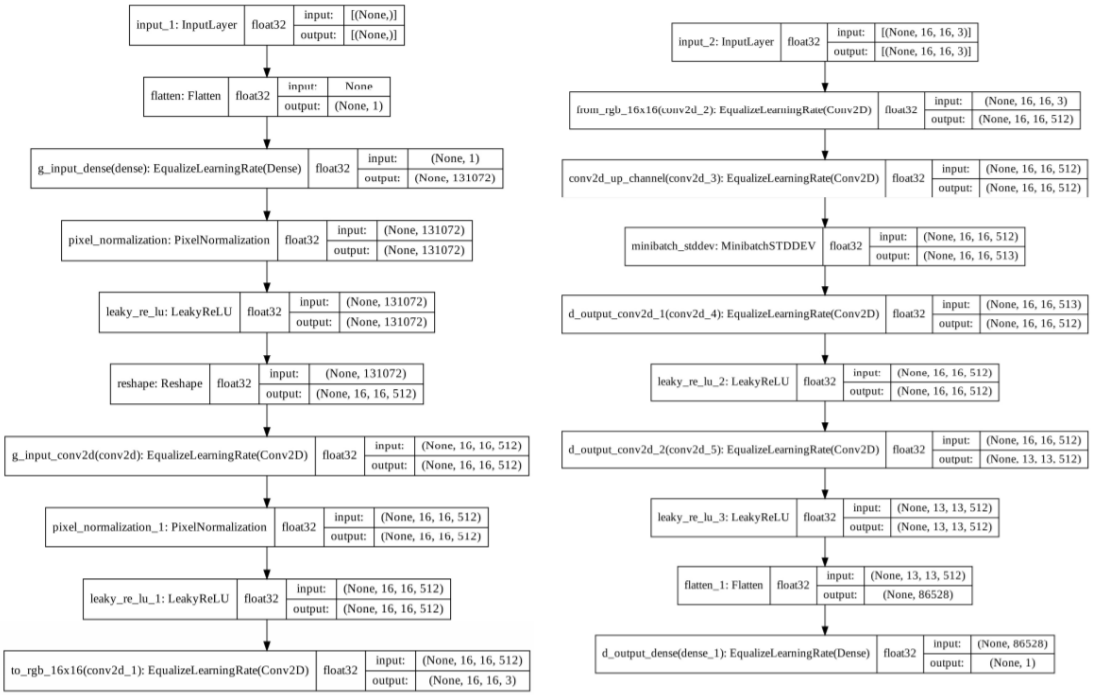

Figure 2: The generator (left) has almost a reversed stack of layers compared to the discriminator (right), based on their structure. They also have an antagonistic relationship: the generator generates a fake sample which serves as input to the discriminator. The discriminator's task is to call out fake from real. But the generator keeps trying to generate samples that would lead the discriminator to mistake fake for real.

4.2 The Setup

1. The training process starts by building a generator-discriminator pair of output size 16×16.

2. The image generation process is kick started by generating a Gaussian noise of size 512, fixing the number of images to generate (this is fixed at 4 images in our case), and generating the corresponding values of weight factor α. The generator built in the previous step is used to generate images (see function generate_and_save_images).

3. Training step for generator: The generator is trained by first making the current version of discriminator classify fake images that is generated by the generator. This prediction is used to calculate the loss and gradients to be applied to the optimizer.

4. Training step for discriminator: The discriminator is trained by feeding the model a batch of real and fake images. the ratio of real and fake images are determined a an ϵ factor generated randomly. Normalized gradients are calculated on the misclassification error on a batch of mixed images to calculate a gradient penalty. This gradient penalty is further used to calculate a discriminator loss and gradients to be applied to the optimizer.

5. Training: If the image size is > 4, it means that the training needs to be resumed from the previous model checkpoint. The previous model is loaded from the location pointed by the model path global variable. For every epoch, the discriminator is trained first followed by the generator according to the aforementioned process. At the end of an epoch, the discriminator and generator losses are collected and the images generated by the generator are saved. Model versions are upgraded every 5 epochs. Therefore, the current version of the models are saved after every 5 epochs. Besides saving the model, the weight factor, image resolution, previous image resolution variables are updated according to the progress of the training.

6. Training for the next version of Generator-discriminator begins after every 5 epochs. As the output resolution of generated images increases, the batch size of the image is reduced to avoid out of memory issues.

7. Testing: Testing begins with loading a trained model of the appropriate target size (256 at the time of this experiment) from the model path. The prediction batch size is fixed at 4. Since the model requires two inputs while training; i.e. the input noise, and weight factor α, Gaussian noise of dimension 4×512 and alpha vector of dimension 4 is generated. The alpha is fixed at 1 during testing.

8. Hyperparameters used for this experiment are summarized in Table 1.

Table 1: Hyperparameters.

Name	**Value**
Number of models	5
Target resolution	16,32,64,128,256
Number of epochs per model	5
Initial target resolution	16×16
Batch Size (initial)	32
GAN loss	Binary Cross entropy
Optimizer	RMSProp
ϵ to avoid divide by zero during pixel normalization	1e-8
Output Activation	tanh
Kernel Initializer	He Normal
Noise Dimension	512
Activation for upsample block	Leaky ReLU

Table 2: Model Results

Model	Training Time per Epoch	Discriminator Loss	Generator Loss
16×16	3.3 min	0.6113	1.0183
32×32	14.6 min	0.6202	1.1007
64×64	47.9 min	0.4259	2.7586
128×128	201 min	0.3141	5.5596

5 Experimental Results

The PGGAN is trained with models 16×16, 32×32, 64×64, and 128×128 for 5 epochs each. Due to time constraints, the 256×256 was only trained for 1 epoch. The binary cross entropy loss of the generator and discriminator were calculated after each model was trained. The results for the models are shown in 2. The outputs of the 32×32, 64×64, and 128×128 models were up scaled to a $218 \times 178 \times 3$ output. The outputs of the models are shown in Figures 5-8. The losses for the discriminator are shown in **??** and the losses for the generator are shown in **??**. The discriminator losses generally decrease with more training and model complexity, while the generator losses increased. This negative correlation is discussed further in Section 6. Time constraints prevented the 256×256 model for being completed in time for the report.

Figure 3: Discriminator's loss per thousand training steps for the entire training run of all the five progressive growing GANs (from 16x16 to 256x256). Overall, the loss value is between zero and one. The cropping of the y-axis is due to TensorBoard default web visualization. At this point in time the 256x256 model has just started its first epoch.

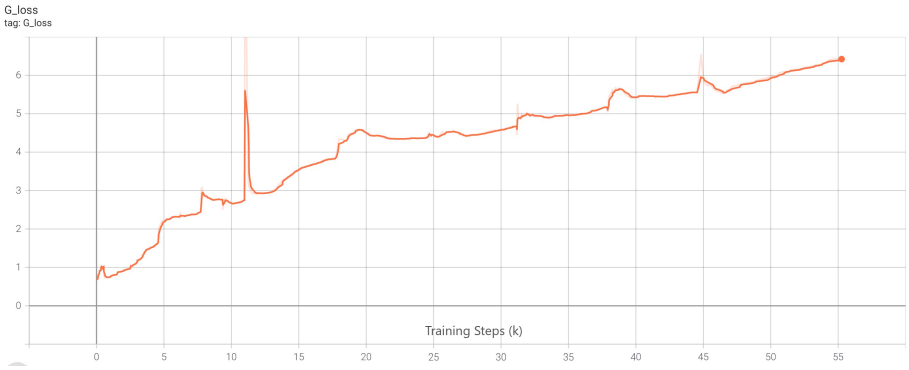

Figure 4: Generator's loss per thousand training steps for the entire training run of all the five progressive growing GANs (from 16x16 to 256x256). Overall, the loss value is between zero and 6.5. At this point in time the 256x256 model has just started its first epoch.

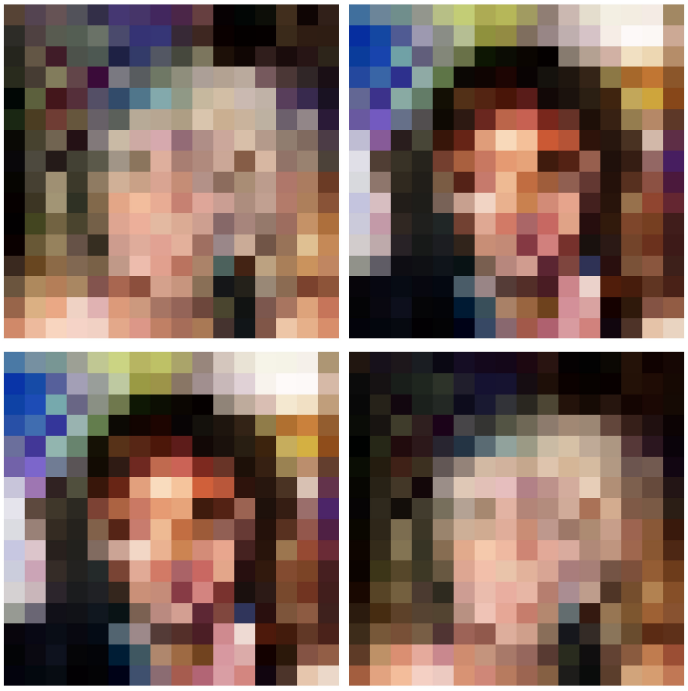

Figure 5: 16x16 model with 16x16 output

Figure 6: 32x32 model with 218x178 output

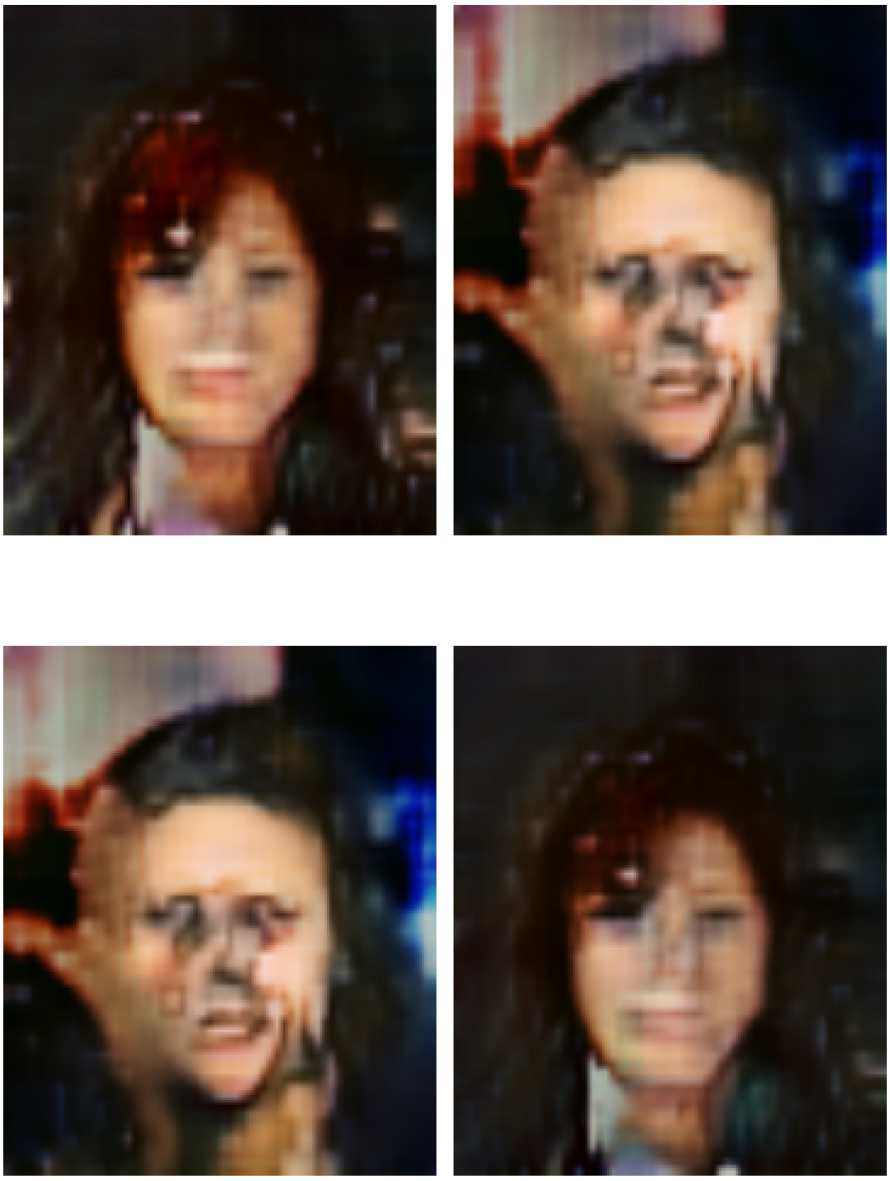

Figure 7: 64x64 model with 218x178 output

Figure 8: 128x128 model with 218x178 output

6 Discussion

As shown in Figures 5-8 and Table 2 the image quality greatly increases with the larger dimensions, the discriminator loss decreases, and the generator loss increases. At first glance, the increase in generator loss appears to do the opposite of what is expected. But the loss shown is what the discriminator's loss is when dealing with generated images, meaning that when the generator error goes up the generator is making more images that are fooling the discriminator. The group would have liked to complete the 256×256 model, but it is taking near 800 minutes per epoch for training. This long training time highlights the importance of having long training times and proper hardware.

Ideally, each model's output in the Progressive Growing of GANs paradigm should output the resolution of the said intermediate model. In our case, we wanted to have results that suit the assignment's description early during training. Hence, we resized the output of the generator to the size of the original dataset images starting with the 32x32 model.

Furthermore, the NVIDIA team experimented with sliced Wasserstein and least-squared loss, but in our training technique, we adapted the loop from the textbook, sticking with the binary cross-entropy or log loss. This choice of the loss function might have adversely impacted the generated images' quality.

7 Conclusions

This assignment shows how increasing the complexity and size of a model increases the training time and the generated image quality. With more time or computation power, it is reasonable to assume that photo-realistic images as generated in [3] can be produced on the mega-pixel scale.

The original paper on progressive growing of GANs starts with a 4x4 model up to a 1024x1024 model on high-resolution datasets of images. In our case, however, our dataset does not contain very high-resolution images, moreover given the time constraint and computational resources, we went from 16x16 to 256x256, and still ran out of time. So, we might have missed finer-grained features that would have been captured by lower resolution models and very high-level features. It would be interesting to explore how the performance can compare to when the growth is implemented, starting from 4x4 and going past 256x256.

Moreover, given the task's constraint of providing no inputs aside from a batch size of images to generate, we could not pass in a varied smoothing term alpha. Indeed, alpha allows to smoothly control the fading-in and fading-out of additional layers, and it has to be fed at each epoch, going from a value of zero to a value of one at the end of the training steps. In the original implementation, alpha is part of the inputs. In our adaptation, we constrained alpha to be random generated when a model is initialized and remains constant through the training steps. It will thus be interesting to see how the performance would compare to that of when alpha is properly fed in.

Table 3: Contributions by team member for this assignment.

Team Member	Contribution
Yves Shamavu	Worked on PGGAN and Report
Mrinal Rawool	Worked on VAE, PGGAN, and Report
Mason Lien	Worked on PGGAN and Report
Matthew Penne	Worked on InfoGAN and Report

References

[1] Xi Chen, Yan Duan, Rein Houthooft, John Schulman, Ilya Sutskever, and Pieter Abbeel. Infogan: Interpretable representation learning by information maximizing generative adversarial nets, 2016. http://arxiv.org/abs/1606.03657 arXiv:1606.03657.

[2] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems.* O'Reilly Media, 2019.

[3] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196*, 2017.

[4] Pascal Lamblin, Hugo Larochelle, and Dan Popovici. Greedy layer-wise training of deep networks. *Advances in Neural Information Processing Systems 19*, 2007. https://doi.org/10.7551/mitpress/7503.003.0024 doi:10.7551/mitpress/7503.003.0024.

[5] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015.