Programming Assignment 4: DCGAN, CycleGAN and BigGAN

Due Date: Tuesday, Mar. 31st, at 11:59pm
Based on an assignment by Paul Vicol

Submission: You must submit 3 files through MarkUs[1]: a PDF file containing your writeup, titled `a4-writeup.pdf`, and your code files `dcgan.ipynb` and `biggan.ipynb`. Your writeup must be typed.

The programming assignments are individual work. See the Course Information handout[2] for detailed policies.

You should attempt all questions for this assignment. Most of them can be answered at least partially even if you were unable to finish earlier questions. If you think your computational results are incorrect, please say so; that may help you get partial credit.

[1] `https://markus.teach.cs.toronto.edu/csc413-2020-01`
[2] `https://csc413-2020.github.io/assets/misc/syllabus.pdf`

Introduction

In this assignment, you'll get hands-on experience coding and training GANs. This assignment is divided into three parts: in the first part, we will implement a specific type of GAN designed to process images, called a Deep Convolutional GAN (DCGAN). We'll train the DCGAN to generate emojis from samples of random noise. In the second part, we will look at a more complex GAN architecture called CycleGAN, which was designed for the task of *image-to-image translation* (described in more detail in Part 2). We'll train the CycleGAN to convert between Apple-style and Windows-style emojis. In the third part, we will turn to a large-scale GAN model – BigGAN – that generates high resolution and high fidelity natural images. We will visualize the class imbeddings learned by BigGAN, and interpolate between them.

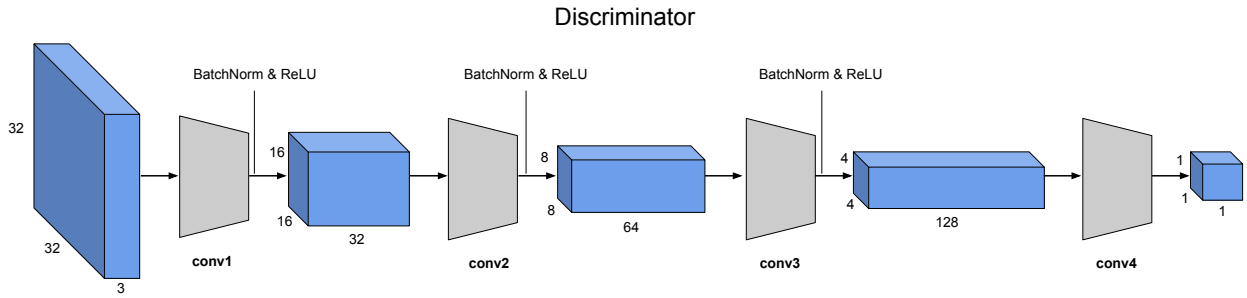
Part 1: Deep Convolutional GAN (DCGAN) [4pt]

For the first part of this assignment, we will implement a *Deep Convolutional GAN (DCGAN)*. A DCGAN is simply a GAN that uses a convolutional neural network as the discriminator, and a network composed of *transposed convolutions* as the generator. To implement the DCGAN, we need to specify three things: 1) the generator, 2) the discriminator, and 3) the training procedure. We will go over each of these three components in the following subsections. Open `https://colab.research.google.com/drive/1Jm2csPnWoKZ2DicPDF48Bq_xiukamPif` on Colab and answer the following questions.

DCGAN

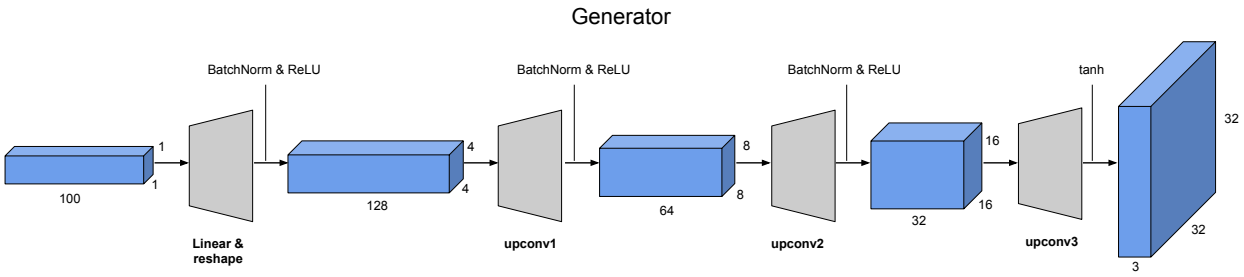
The discriminator in this DCGAN is a convolutional neural network that has the following architecture:

The `DCDiscriminator` class is implemented for you. We strongly recommend you to carefully read the code, in particular the `__init__` method. The three stages of the generator architectures are implemented using `conv` and `upconv` functions respectively, all of which provided in `Helper Modules`.



Generator

Now, we will implement the generator of the DCGAN, which consists of a sequence of transpose convolutional layers that progressively upsample the input noise sample to generate a fake image. The generator has the following architecture:



1. [1pt] **Implementation:** Implement this architecture by filling in the `__init__` method of the `DCGenerator` class, shown below. Note that the forward pass of `DCGenerator` is already provided for you.

   (Hint: You may find the provided `DCDiscriminator` useful.)

   Note: The original DCGAN generator uses `deconv` function to expand the spatial dimension. Odena et al. later found the `deconv` creates checker board artifacts in the generated samples. In this assignment, we will use `upconv` that consists of an upsampling layer followed by conv2D to replace the deconv module (analogous to the `conv` function used for the discriminator above) in your generator implementation.

Training Loop

Next, you will implement the training loop for the DCGAN. A DCGAN is simply a GAN with a specific type of generator and discriminator; thus, we train it in exactly the same way as a standard GAN. The pseudo-code for the training procedure is shown below. The actual implementation is simpler than it may seem from the pseudo-code: this will give you practice in translating math to code.

Algorithm 1 GAN Training Loop Pseudocode

procedure TRAINGAN
  Draw m training examples $\{x^{(1)}, \ldots, x^{(m)}\}$ from the data distribution p_{data}
  Draw m noise samples $\{z^{(1)}, \ldots, z^{(m)}\}$ from the noise distribution p_z
  Generate fake images from the noise: $G(z^{(i)})$ for $i \in \{1, \ldots m\}$
  Compute the (least-squares) discriminator loss:

$$J^{(D)} = \frac{1}{2m}\sum_{i=1}^{m}\left[\Big(D(x^{(i)}) - 1\Big)^2\right] + \frac{1}{2m}\sum_{i=1}^{m}\left[\Big(D(G(z^{(i)}))\Big)^2\right]$$

  Update the parameters of the discriminator
  Draw m new noise samples $\{z^{(1)}, \ldots, z^{(m)}\}$ from the noise distribution p_z
  Generate fake images from the noise: $G(z^{(i)})$ for $i \in \{1, \ldots m\}$
  Compute the (least-squares) generator loss:

$$J^{(G)} = \frac{1}{m}\sum_{i=1}^{m}\left[\Big(D(G(z^{(i)})) - 1\Big)^2\right]$$

  Update the parameters of the generator

1. [1pt] **Implementation:** Fill in the `gan_training_loop` function in the GAN section of the notebook.

   There are 5 numbered bullets in the code to fill in for the discriminator and 3 bullets for the generator. Each of these can be done in a single line of code, although you will not lose marks for using multiple lines.

   Note that in the discriminator update, we have provided you with the implementation of gradient penalty. Gradient penalty adds a term in the discriminator loss, and it is another popular technique for stabilizing GAN training. Gradient penalty can take different forms, and it is an active research area to study its effect on GAN training [Gulrajani et al., 2017] [Kodali et al., 2017] [Mescheder et al., 2018].

Experiment

1. [1pt] We will train a DCGAN to generate Windows (or Apple) emojis in the Training - GAN section of the notebook. By default, the script runs for 20000 iterations, and should take approximately half an hour on Colab. The script saves the output of the generator for a fixed noise sample every 200 iterations throughout training; this allows you to see how the generator improves over time. How does the generator performance evolve over time? **Include in your write-up some representative samples (e.g. one early in the training, one with satisfactory image quality, and one towards the end of training, and give the iteration number for those samples. Briefly comment on the quality of the samples.**

2. [1pt] Multiple techniques can be used to stabilize GAN training. We have provided code for `gradient_penalty` [Thanh-Tung et al., 2019].

   Try turn on the `gradient_penalty` flag in the `args_dict` and train the model again. Are you able to stabilize the training? Briefly explain why the gradient penalty can help. You are welcome to check out the related literature above for gradient penalty.

   (Hint: Consider relationship between the Jacobian norm and its singular values.)

3. [0pt] Playing with some other hyperparameters such as `spectral_norm`. You can also try lowering `lr` (learning rate), and increasing `d_train_iters` (number of discriminator updates per generator update). Are you able to stabilize the training? Can you explain why the above measures help?

Part 2: CycleGAN [3pt]

Now we are going to implement the CycleGAN architecture.

Motivation: Image-to-Image Translation

Say you have a picture of a sunny landscape, and you wonder what it would look like in the rain. Or perhaps you wonder what a painter like Monet or van Gogh would see in it? These questions can be addressed through *image-to-image translation* wherein an input image is automatically converted into a new image with some desired appearance.

Recently, Generative Adversarial Networks have been successfully applied to image translation, and have sparked a resurgence of interest in the topic. The basic idea behind the GAN-based approaches is to use a conditional GAN to learn a mapping from input to output images. The loss functions of these approaches generally include extra terms (in addition to the standard GAN loss), to express constraints on the types of images that are generated.

A recently-introduced method for image-to-image translation called CycleGAN is particularly interesting because it allows us to use *un-paired* training data. This means that in order to train it to translate images from domain X to domain Y, we do not have to have exact correspondences between individual images in those domains. For example, in the paper that introduced CycleGANs, the authors are able to translate between images of horses and zebras, even though there are no images of a zebra in exactly the same position as a horse, and with exactly the same background, etc.

Thus, CycleGANs enable learning a mapping from one domain X (say, images of horses) to another domain Y (images of zebras) *without* having to find perfectly matched training pairs.

To summarize the differences between paired and un-paired data, we have:

- Paired training data: $\{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$
- Un-paired training data:
  - Source set: $\{x^{(i)}\}_{i=1}^{N}$ with each $x^{(i)} \in X$
  - Target set: $\{y^{(j)}\}_{j=1}^{M}$ with each $y^{(j)} \in Y$
  - For example, X is the set of horse pictures, and Y is the set of zebra pictures, where there are no direct correspondences between images in X and Y

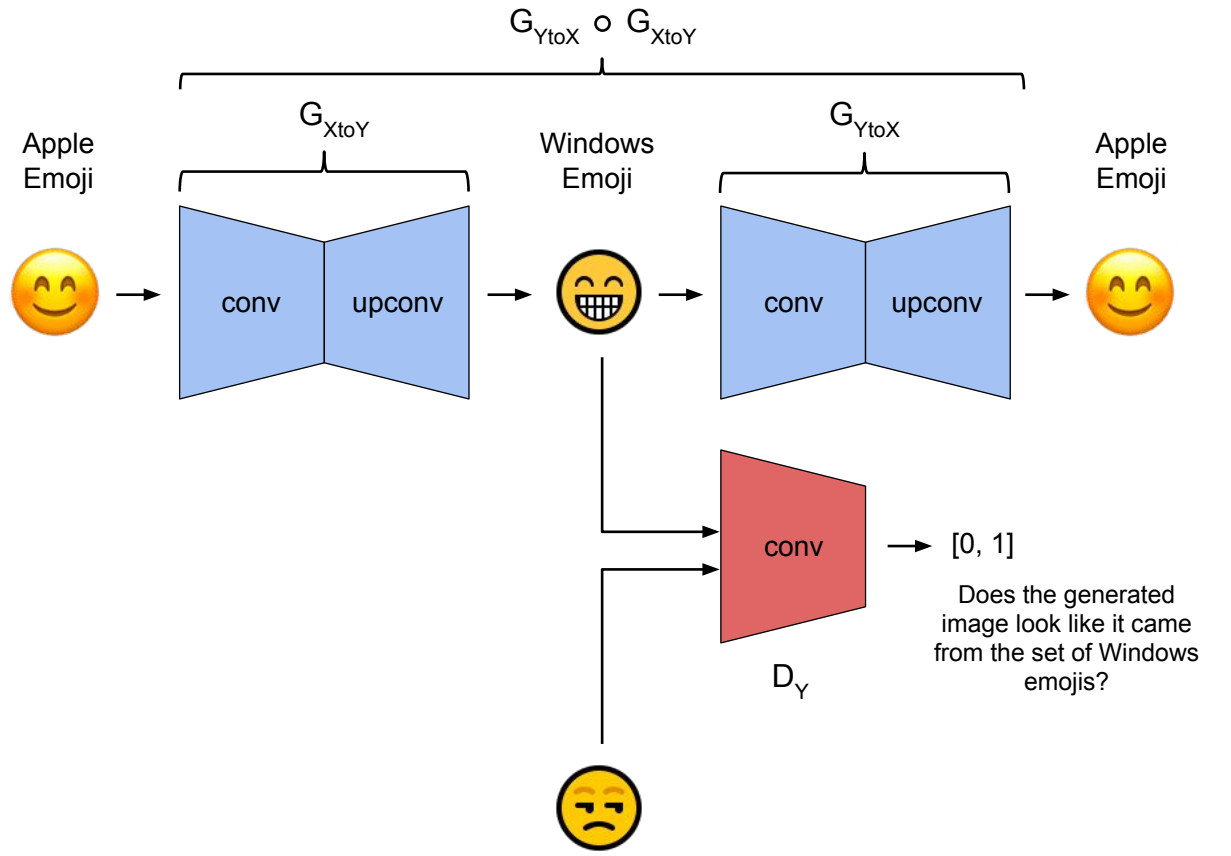
Emoji CycleGAN

Now we'll build a CycleGAN and use it to translate emojis between two different styles, in particular, Windows $\leftrightarrow$ Apple emojis.

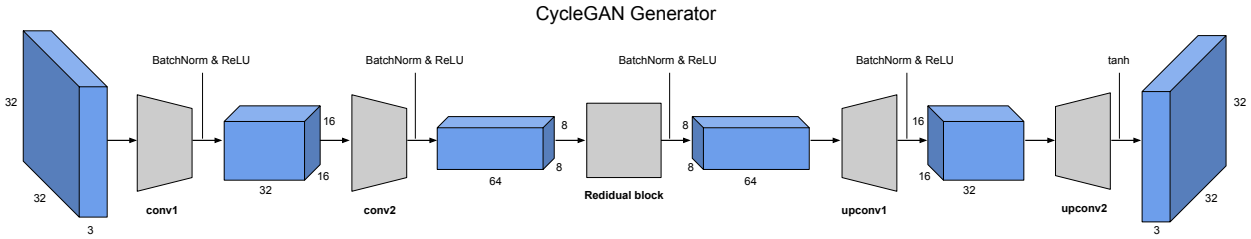
Generator

The generator in the CycleGAN has layers that implement three stages of computation: 1) the first stage *encodes* the input via a series of convolutional layers that extract the image features; 2) the second stage then *transforms* the features by passing them through one or more *residual blocks*; and 3) the third stage *decodes* the transformed features using a series of transpose convolutional layers, to build an output image of the same size as the input.

The residual block used in the transformation stage consists of a convolutional layer, where the input is added to the output of the convolution. This is done so that the characteristics of the output image (e.g., the shapes of objects) do not differ too much from the input.



The `CycleGenerator` class is implemented for you. We strongly recommend you to carefully read the code, in particular the `__init__` method. The three stages of the generator architectures are implemented using `conv`, `ResnetBlock` and `upconv` functions respectively, all of which provided in `Helper Modules`.



Note: There are two generators in the CycleGAN model, $G_{X \to Y}$ and $G_{Y \to X}$, but their implementations are identical. Thus, in the code, $G_{X \to Y}$ and $G_{Y \to X}$ are simply different instantiations of the same class.

CycleGAN Training Loop

Finally, we will take a look at the CycleGAN training procedure. The training objective is more involved than the procedure in Part 1, but an implementation is provided.

Algorithm 2 CycleGAN Training Loop Pseudocode

procedure TRAINCYCLEGAN
  Draw a minibatch of samples $\{x^{(1)}, \ldots, x^{(m)}\}$ from domain X
  Draw a minibatch of samples $\{y^{(1)}, \ldots, y^{(m)}\}$ from domain Y
  Compute the discriminator loss on real images:

$$\mathcal{J}_{real}^{(D)} = \frac{1}{m}\sum_{i=1}^{m}(D_X(x^{(i)}) - 1)^2 + \frac{1}{n}\sum_{j=1}^{n}(D_Y(y^{(j)} - 1)^2$$

  Compute the discriminator loss on fake images:

$$\mathcal{J}_{fake}^{(D)} = \frac{1}{m}\sum_{i=1}^{m}(D_Y(G_{X\to Y}(x^{(i)})))^2 + \frac{1}{n}\sum_{j=1}^{n}(D_X(G_{Y\to X}(y^{(j)})))^2$$

  Update the discriminators
  Compute the $Y \to X$ generator loss:

$$\mathcal{J}^{(G_{Y\to X})} = \frac{1}{n}\sum_{j=1}^{n}(D_X(G_{Y\to X}(y^{(j)})) - 1)^2 + \lambda_{cycle}\mathcal{J}_{cycle}^{(Y\to X\to Y)}$$

  Compute the $X \to Y$ generator loss:

$$\mathcal{J}^{(G_{X\to Y})} = \frac{1}{m}\sum_{i=1}^{m}(D_Y(G_{X\to Y}(x^{(i)})) - 1)^2 + \lambda_{cycle}\mathcal{J}_{cycle}^{(X\to Y\to X)}$$

  Update the generators

Cycle Consistency

The most interesting idea behind CycleGANs (and the one from which they get their name) is the idea of introducing a *cycle consistency loss* to constrain the model. The idea is that when we translate an image from domain X to domain Y, and then translate the generated image *back* to domain X, the result should look like the original image that we started with.

The cycle consistency component of the loss is the L1 distance between the input images and their *reconstructions* obtained by passing through both generators in sequence (i.e., from domain X to Y via the $X \to Y$ generator, and then from domain Y back to X via the $Y \to X$ generator). The cycle consistency loss for the $Y \to X \to Y$ cycle is expressed as follows:

$$\lambda_{cycle}\mathcal{J}_{cycle}^{(Y\to X\to Y)} = \lambda_{cycle}\frac{1}{m}\sum_{i=1}^{m}\|y^{(i)} - G_{X\to Y}(G_{Y\to X}(y^{(i)}))\|_1,$$

where λ_{cycle} is a scalar hyper-parameter balancing the two loss terms: the cycle consistant loss and the GAN loss. The loss for the $X \to Y \to X$ cycle is analogous.

CycleGAN Experiments

1. [1pt] Train the CycleGAN to translate Apple emojis to Windows emojis in the Training - CycleGAN section of the notebook. The script will train for 5000 iterations, and save generated samples in the `samples_cyclegan` folder. In each sample, images from the source domain are shown with their translations to the right.

Include in your writeup the samples from both generators at either iteration 200 and samples from a later iteration.

2. [1pt] Change the random seed and train the CycleGAN again. What are the most noticible difference between the *similar* quality samples from the different random seeds? Explain why there is such a difference?

3. [1pt] Changing the default `lambda_cycle` hyperparameters and train the CycleGAN again. Try a couple of different values including *without* the cycle-consistency loss. (i.e. `lambda_cycle = 0`)

   For different values of `lambda_cycle`, include in your writeup some samples from both generators at either iteration 200 and samples from a later iteration. Do you notice a difference between the results with and without the cycle consistency loss? Write down your observations (positive or negative) in your writeup. Can you explain these results, i.e., why there is or isn't a difference among the experiments?

Part 3: BigGAN [2pt]

For this part, we will implement the interpolation function that you see in many GAN papers to show that the GAN model can generate novel images between classes. We will use BigGAN, Brock et al. [2018], as our learned model to do the interpolation.

BigGAN is a state-of-the-art model that pulls together a suite of recent best practices in training GANs. The authors scaled up the model to large-scale datasets like Imagenet. The result is the routine generation of both high-resolution (large) and high-quality (high-fidelity) images.

Figure 1: "Cherry-picked" high-resolution samples from BigGAN

BigGAN benefits from larger batch size and more expressive generator and discriminator. A varied range of techniques has been proposed to improve GANs in terms of sample quality, diversity and training stability. BigGAN utilized self-attention Zhang et al. [2018], moving average of generator weights Karras et al. [2017] , orthogonal initialization Saxe et al. [2013], and different residual block structures. See figure 2 for the illustration of the residual block for both generator and discriminator. They used conditional GAN where generator receives both a learnable class embedding and a noise vector. In the generator architecture, conditional batch normalization Huang and Belongie [2017] conditioned on class embedding is used. They also proposed truncation of noise distribution to control fidelity vs quality of samples. The paper shows that applying Spectral NormalizationMiyato et al. [2018] in generator improves stability, allowing for fewer discriminator steps per iteration.

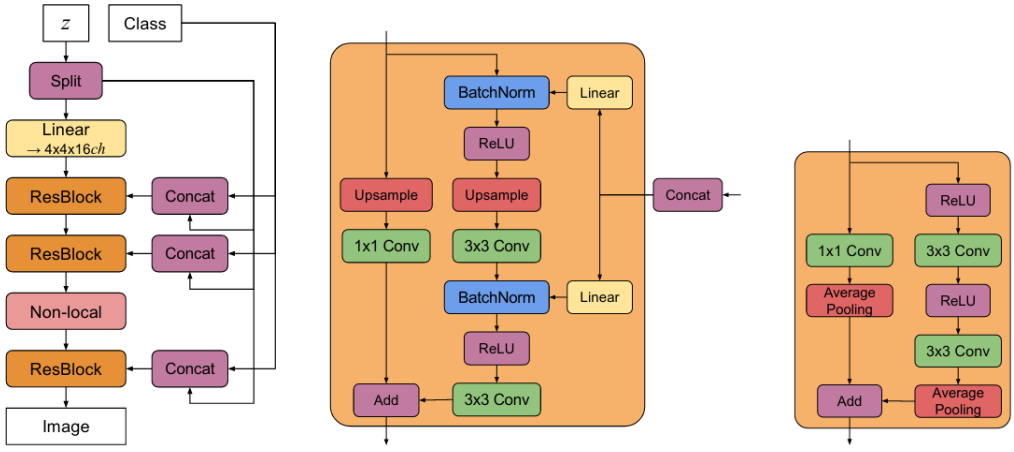


Figure 2: **Left:** Architecture of generator. **Middle:** Residual module of generator. **Right:** Residual block of discriminator (taken from original paper)

BigGAN Experiments

In this part we would our pre-trained BigGAN to "fantasize" a new class category and generate samples from that class. We will create a new class embedding, $\mathbf{r}_{new\_class}$, using a linear interpolation between the two existing class embeddings, $\mathbf{r}_{class1}$ and $\mathbf{r}_{class2}$:

$$\mathbf{r}_{new\_class} = \alpha \mathbf{r}_{class1} + (1 - \alpha)\mathbf{r}_{class2}, \forall \alpha \in [0, 1].$$

Open `https://colab.research.google.com/drive/1mKk7isQthab5k68FqqcdSJTT4P-7DV-s` on Colab and answer the following questions.

1. [1pt] Based on T-SNE visualization of class embeddings, which two classes are good candidates for interpolation? Which two classes might not be a good match for interpolation? In each case, give 2 examples. Briefly explain your choice.

2. [1pt] Complete `generate_linear_interpolate_sample` function. Verify the examples you gave in the previous question by generating samples. Include the four sets of generated images in your report.

What you need to submit

- Your code file: `dcgan.ipynb` and `biggan.ipynb`.
- A PDF document titled `a4-writeup.pdf`containing **code screenshots, any experiment results or visualizations, as well as your answers to the written questions**.

Further Resources

For further reading on GANs in general, and CycleGANs in particular, the following links may be useful:

1. Deconvolution and Checkerboard Artifacts (Odena et al., 2016)
2. Unpaired image-to-image translation using cycle-consistent adversarial networks (Zhu et al., 2017)
3. Generative Adversarial Nets (Goodfellow et al., 2014)
4. An Introduction to GANs in Tensorflow
5. Generative Models Blog Post from OpenAI
6. Official PyTorch Implementations of Pix2Pix and CycleGAN

References

Andrew Brock, Jeff Donahue, and Karen Simonyan. Large scale gan training for high fidelity natural image synthesis. *arXiv preprint arXiv:1809.11096*, 2018.

Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. Improved training of wasserstein gans. In *Advances in neural information processing systems*, pages 5767–5777, 2017.

Xun Huang and Serge Belongie. Arbitrary style transfer in real-time with adaptive instance normalization. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1501–1510, 2017.

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.

Naveen Kodali, Jacob Abernethy, James Hays, and Zsolt Kira. On convergence and stability of gans. *arXiv preprint arXiv:1705.07215*, 2017.

Lars Mescheder, Andreas Geiger, and Sebastian Nowozin. Which training methods for gans do actually converge? *arXiv preprint arXiv:1801.04406*, 2018.

Takeru Miyato, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. Spectral normalization for generative adversarial networks. *arXiv preprint arXiv:1802.05957*, 2018.

Andrew M Saxe, James L McClelland, and Surya Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv preprint arXiv:1312.6120*, 2013.

Hoang Thanh-Tung, Truyen Tran, and Svetha Venkatesh. Improving generalization and stability of generative adversarial networks. *arXiv preprint arXiv:1902.03984*, 2019.

Han Zhang, Ian Goodfellow, Dimitris Metaxas, and Augustus Odena. Self-attention generative adversarial networks. *arXiv preprint arXiv:1805.08318*, 2018.