Programming Assignment 4: DCGAN, StyleGAN, and DQN

Version: 1.0
Version Release Date: 2021-03-19
Due Date: Thursday, April 1st, at 11:59pm

Submission: You must submit 4 files through MarkUs[1]: a PDF file containing your writeup, titled `a4-writeup.pdf`, and your code files `a4-dcgan.ipynb`, `a4-stylegan.ipynb`, `a4-dqn.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.

The teaching assistants for this assignment are Bolin Gao and Rex Ma. Send your email with subject "*[CSC413] PA4 ...*" to csc413-2021-01-tas@cs.toronto.edu or post on Piazza with the tag `pa4`.

[1] `https://markus.teach.cs.toronto.edu/csc413-2021-01`
[2] `https://csc413-uoft.github.io/2021/assets/misc/syllabus.pdf`

Introduction

In this assignment, you'll get hands-on experience coding and training GANs, as well as DQN (Deep Q-learning Network), one of Reinforcement Learning methods. 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, you will get to play with a state-of-the-art GAN called StyleGAN2-Ada. In the third part, we will implement and train a DQN agent to learn how to play the CartPole balancing game. It will be fun to see your model performs much better than you on the simple game :).

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

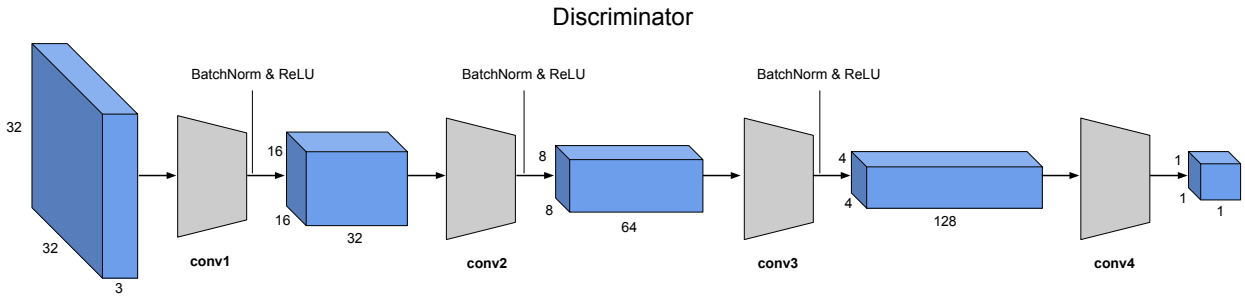
For the first part of this assignment, we will implement a *Deep Convolutional GAN (DCGAN)*. A DCGAN is simply a GAN that uses a convolutional neural network as the discriminator, and a network composed of *transposed convolutions* as the generator. To implement the DCGAN, we need to specify three things: 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 [DCGAN notebook link] on Colab and answer the following questions.

DCGAN

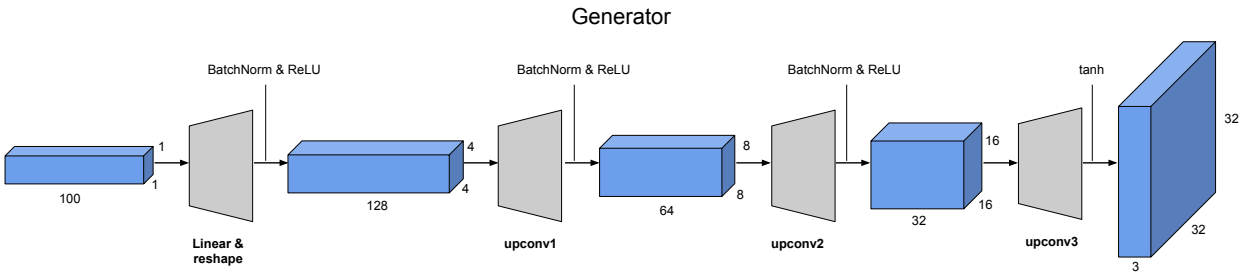
The discriminator in this DCGAN is a convolutional neural network that has the following architecture:

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: StyleGAN2-Ada [4pt]

For this part of the assignment, you will get to to play with a state-of-the-art GAN called StyleGAN2-Ada (Karras et al. [2020a]). Several samples from the authors' paper are shown in Figure 1.

Figure 1: *Images sampled from the StyleGAN2-Ada Generator. From Left: "METFACES", Dog/Cat/Wild Animal, CIFAR10. Karras et al. [2020a]*

To understand how StyleGAN2-Ada works, let's first review the basics of StyleGAN (Karras et al. [2019]).

StyleGAN: A StyleGAN differentiates itself from a regular GAN in that its generator has been heavily modified for generating images with multiple layers of details, such as hair strand, freckles (fine detail), eye open/closed, hairstyle (mid detail) and pose, glasses, face shape (coarse detail). This amazing level of control is achieved through the architecture shown in Figure 2.

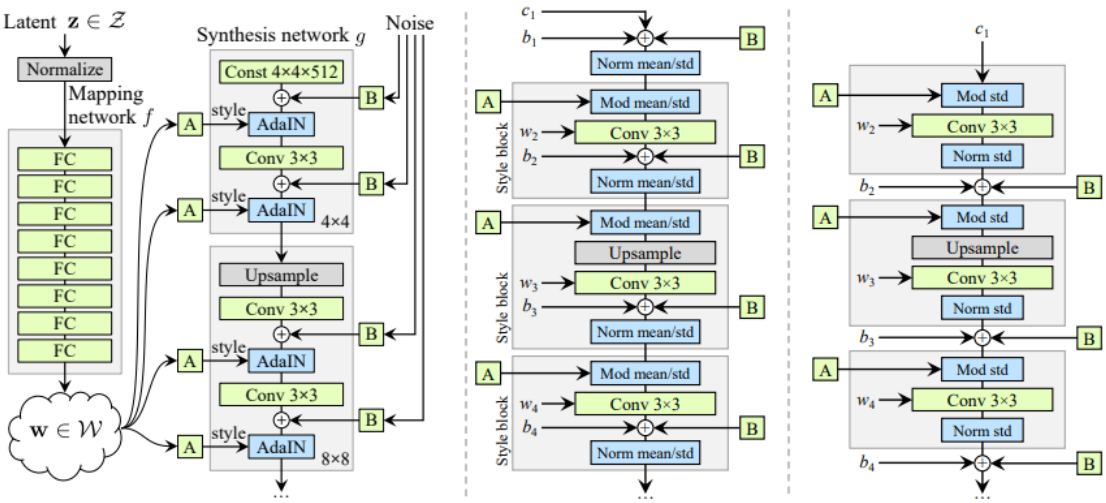


Figure 2: *Generator Architecture for StyleGAN and StyleGAN2. Left: StyleGAN, Center: StyleGAN2, Right: Expanded View of Demodulation operation. From Karras et al. [2019, 2020b]*

In Figure 2(left), StyleGAN first passes a *latent code* $z \in \mathcal{Z} = \mathbb{R}^{512}$ into a so-called *intermediate latent variable* $w \in \mathcal{W} = \mathbb{R}^{512}$. Next, w is fed through multiple blocks which modifies the image at different resolutions (from 4^2 (coarse) to 1024^2 (fine)) within a so-called *Synthesis Network*, starting from a constant $4 \times 4 \times 512$ tensor. The output of each block is progressively up-sampled into higher resolution, an idea from Karras et al. [2017], which greatly helps with learning and allows for the fine, medium and coarse details of an image to be controlled in a localized fashion.

Some other techniques used to improve the quality of the image includes the usage of a WGAN-GP objective (Arjovsky et al. [2017]), adaptive instance normalization (AdaIN) and a modified truncation trick (Brock et al. [2018]) that works in $\mathcal{W}$. The truncation trick essentially shrinks the latent distribution to avoid sampling from regions where the generator was not trained on, thus forming a trade-off between image quality and variety.

StyleGAN2: One major problem that StyleGAN had is that it generates several undesirable distortions such as an water-droplet like "blob artifact", which was found to be caused by the AdaIN operation and the progressive up-sampling architecture. StyleGAN2 addresses this problem by replacing the AdaIN operation with a so-called *demodulation* operation Figure 2(middle, right), which removes the said artifact. Another crucial modification is the replacement of the progressive up-sampling architecture with skip connections, which allows for the same level of control but without artifacts.

StyleGAN2-Ada: Training a StyleGAN is expensive. It can take up to 41 days on a single Tesla V100 GPU using on a dataset with $\mathcal{O}(10^5) - \mathcal{O}(10^6)$ high-quality examples. Training on smaller dataset on the other hand will lead to overfitting of the discriminator. The usual data augmentation will lead to undesirable distortions. StyleGAN2-Ada mitigates these issues through a so-called *adaptive discriminator augmentation*, which allows it to be trained on a few thousand images while preserving the same quality of the images as generated by StyleGAN2.

In the following experiments, we generate images and manipulate these images with a pre-trained StyleGAN2-Ada generator. Since StyleGAN2-Ada is written in Tensorflow, therefore we will have to run in a Tensorflow environment. Don't worry, you do not need to write any TensorFlow code.

Experiments

Open the Colab notebook link to begin: [StyleGAN notebook link].

1. [1pt] **Sampling and Identifying Fakes** To begin, unlock one of the pre-trained generators (which ever you prefer). Complete `generate_latent_code` and `generate_images` functions to generate a small row of $3-5$ images. This is done by following the instructions in the notebook as well as StyleGAN's official documentation `https://github.com/NVlabs/stylegan` starting from "*There are three ways to use the pre-trained generator....*". If you wish, you can try to use `https://www.whichfaceisreal.com/learn.html` as a guideline to spot any imperfections that you detect in these images. Do not include these images in your submission.

2. [1pt] **Interpolation**

   Complete the `interpolate_images` function using linear interpolation between two latent codes z_1, z_2 sampled from your `generate_latent_code` function created previously,

   $$z = rz_1 + (1 - r)z_2, r \in [0, 1] \tag{1}$$

   and feed this interpolation through the StyleGAN2-Ada generator `Gs`. **Include a small row of interpolation in your PDF submission.**

3. [2pt] **Style Mixing and Fine Control**

   In the final part, you will reproduce the famous style mixing example from the original StyleGAN paper Figure 3. There are two main steps:

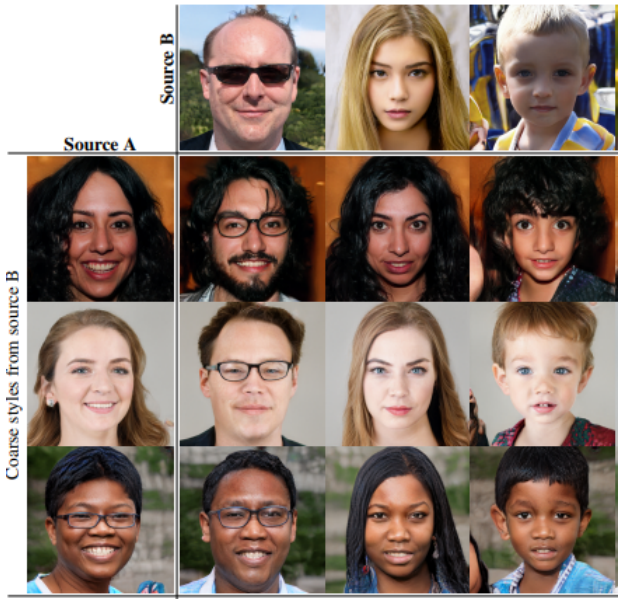


Figure 3: *Style Mixing figure from Karras et al.*

Step 1: Follow the StyleGAN documentation `https://github.com/NVlabs/stylegan` starting from *Look up Gs.components.mapping and Gs.components.synthesis to access individual sub-networks of the generator...* and the instructions in the notebook to complete generate_from_subnetwork. Note: before putting src_dlatents into Gs.components.synthesis.run() we will need to perform the truncation trick as described from the StyleGAN paper (which you were using it implicitly in the previous parts), which involves the following:

(a) sample an average latent vector w_{avg} using Gs.get_var (see documentation `https://github.com/NVlabs/stylegan`)

(b) set src_dlatents as,

$$\text{src\_dlatents} = w_{avg} + (\text{src\_dlatents} - w_{avg}) \times \psi \tag{2}$$

where $\psi \in [-1, 1]$ is your truncation constant and src_dlatents on the right-hand side is the output from Gs.components.mapping.run.

For more details on how this works, see Appendix B of Karras et al. [2019] or Brock et al. [2018], where it was first introduced. Sample a few images to ensure that your function works. Do not include these images in your submission.

Step 2: Copy the code you used to generate images from sub-network into the indicated location in the final cell of the notebook. Initialize the col_seeds, row_seeds and col_styles and generate a grid of image. Run the final cell to generate a grid of images. Now, experiment with the col_styles variable. **In a few sentences, Explain what col_styles does, for instance, roughly describe what these numbers corresponds to. Create a simple experiment to backup your argument. Include at maximum two sets of images that illustrates the effect of changing col_styles along with your explanation.** Include them as screenshots to minimize the size of your submission. Make reference to the StyleGAN papers by Karras et al. [2019, 2020a,b] if you wish. (Hint: coarse versus fine details.)

Part 3: Deep Q-Learning Network (DQN) [4pt]

In this part of the assignment, we will apply Reinforcement Learning (DQN) to tackle the CartPole Balancing game, the game that seems easy but actually quite hard. If you haven't tried it yet, I recommend you try it first [the link]. However, the difficult game for human may be very simple to a computer.

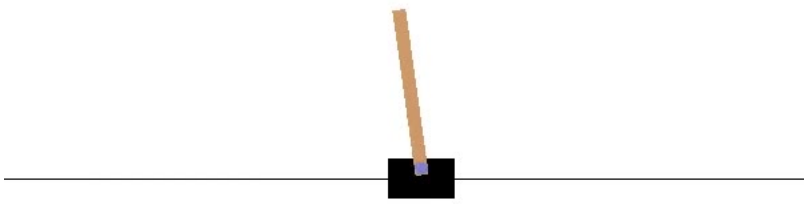


Figure 4: *Image of the CartPole Balancing game from OpenAI Gym.Brockman et al. [2016]*

DQN Overview

Reinforcement learning defines an environment for the agent to perform certain actions (according to the policy) that maximize the reward at every time stamp. Essentially, our aim is to train a agent that tries to maximize the discounted, cumulative reward $R_{t_0} = \sum_{t=t_0}^{\infty} \gamma^{t-t_0} r_t$. Because we assume there can be infinite time stamps, the discount factor, γ, is a constant between 0 and 1 that ensures the sum converges. It makes rewards from the uncertain far future less important for our agent than the ones in the near future.

The idea of Q-learning is that if we have a function $Q^*(state, action)$ that outputs the maximum expected cumulative reward achievable from a given state-action pair, we could easily construct a policy (action selection rule) that maximizes the reward:

$$\pi^*(s) = \operatorname*{argmax}_a \ Q^*(s, a) \tag{3}$$

However, we don't know everything about the world, so we don't have access to Q^*. But, since neural networks are universal function approximators, we can simply create one and train it to resemble Q^*. For our training update rule, we will use a fact that every Q function for some policies obeys the Bellman equation:

$$Q^\pi(s, a) = r + \gamma Q^\pi(s', \pi(s')) \tag{4}$$

An intuitive explanation of the structure of the Bellman equation is as follows. Suppose that the agent has received reward r_t at the current state, then the maximum discounted reward from this point onward is equal to the current reward plus the maximum expected discounted reward $\gamma Q^*(s_{t+1}, a_{t+1})$ from the next stage onward. The difference between the two sides of the equality is known as the temporal difference error, δ:

$$\delta = Q(s, a) - (r + \gamma \max_a Q(s', a)) \tag{5}$$

Our goal is the minimise this error, so that we can have a good Q function to estimate the rewards given any state-action pair.

Experiments

Open the Colab notebook link to begin: [DQN notebook link]. Read through the notebook and play around with it. More detailed instructions are given in the notebook. Have fun!

1. [1pt] **Implementation of $\epsilon-$greedy**

   Complete the function `get_action` for the agent to select an action based on current state. We want to balance exploitation and exploration through $\epsilon-$**greedy**, which is explained in the notebook.

2. [2pt] **Implementation of DQN training step**

   Complete the function `train` for the model to perform a single step of optimization. This is basically to construct the the temporal difference error δ and perform a standard optimizer update. Notice that there are two networks in the DQN_network, policy_net and target_net, think about how to use these two networks to construct the loss.

3. [1pt] **Train your DQN Agent**

   After implementing the required functions, now you can train your DQN Agent, and you are suggested to tune the hyperparameters listed in the notebook. Hyperparameters are important to train a good agent.

   After all of these, now you can validate your model by playing the CartPole Balance game!

What you need to submit

- Your code files: `a4-dcgan.ipynb`, `a4-stylegan.ipynb`, `a4-dqn.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, DCGAN, StyleGAN and DQN, the following links may be useful:

1. Generative Adversarial Nets (Goodfellow et al., 2014)
2. Deconvolution and Checkerboard Artifacts (Odena et al., 2016)
3. Progressive Growing of GANs (Karras et al. [2017]
4. Analyzing and Improving the Image Quality of StyleGAN (Karras et al. [2020b])
5. An Introduction to GANs in Tensorflow
6. Generative Models Blog Post from OpenAI
7. Playing Atari with Deep Reinforcement Learning (Mnih et al., 2013)
8. Deep Reinforcement Learning: A Brief Survey (Arulkumaran et al., 2017)

References

Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. In *International conference on machine learning*, pages 214–223. PMLR, 2017.

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

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

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.

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.

Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4401–4410, 2019.

Tero Karras, Miika Aittala, Janne Hellsten, Samuli Laine, Jaakko Lehtinen, and Timo Aila. Training generative adversarial networks with limited data. *arXiv preprint arXiv:2006.06676*, 2020a.

Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Analyzing and improving the image quality of stylegan. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8110–8119, 2020b.

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.

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