Programming Assignment 4: DCGAN, GCN, and DQN

Version: 1.0
Version Release Date: 2022-03-26
Due Date: Thursday, April 8th, 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-gcn.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 Rex Ma and Emmy Fang. Send your email with subject "*[CSC413] PA4 ...*" to csc413-2022-01-tas@cs.toronto.edu or post on Piazza with the tag `pa4`.

[1]`https://markus.teach.cs.toronto.edu/2022-01/courses/16`
[2]`https://uoft-csc413.github.io/2022/assets/misc/syllabus.pdf`

Introduction

In this assignment, you'll get hands-on experience coding and training GANs, GCN (Graph Convolution Network) 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 learn how to implement the vanilla version of GCN and GAT. 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]

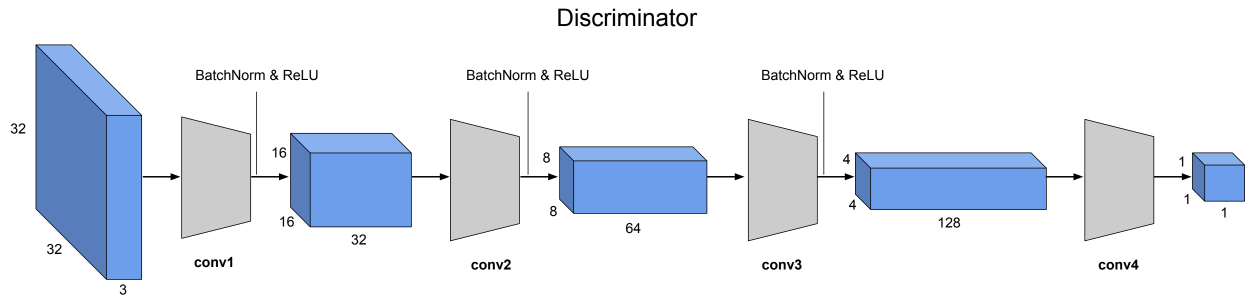
For the first part of this assignment, we will implement a *Deep Convolutional GAN (DCGAN)*. A DCGAN is simply a GAN that uses a convolutional neural network as the discriminator, and a network composed of *transposed convolutions* as the generator. To implement the DCGAN, we need to specify three things: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

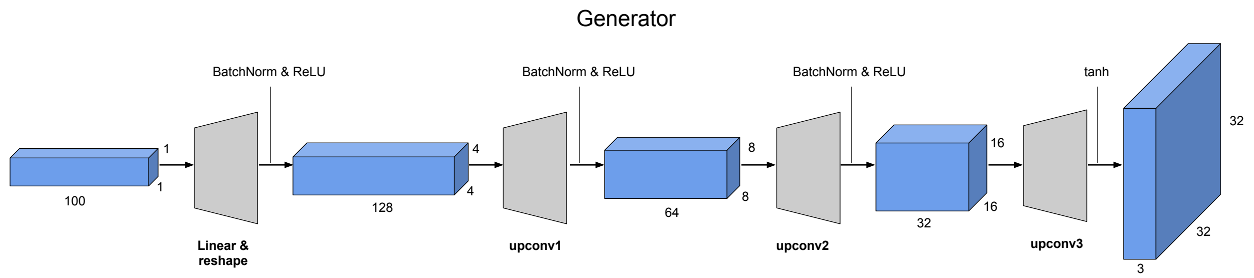
The discriminator in this DCGAN is a convolutional neural network that has the following architecture:

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 Regular GAN Training Loop Pseudocode

```
1: procedure TrainGAN
2:     Draw $m$ training examples $\{x^{(1)}, \dots, x^{(m)}\}$ from the data distribution $p_{data}$
3:     Draw $m$ noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from the noise distribution $p_z$
4:     Generate fake images from the noise: $G(z^{(i)})$ for $i \in \{1, \dots .m\}$
5:     Compute the discriminator loss (negative log likelihood):
           $$L^{(D)} = -\frac{1}{m}\sum_{i=1}^{m}\left[\log D\left(x^{(i)}\right) + \log\left(1 - D\left(G(z^{(i)})\right)\right)\right]$$
6:     Update the parameters of the discriminator
7:     Draw $m$ new noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from the noise distribution $p_z$
8:     Generate fake images from the noise: $G(z^{(i)})$ for $i \in \{1, \dots .m\}$
9:     Compute the generator loss (negative log likelihood):
           $$L^{(G)} = -\frac{1}{m}\sum_{i=1}^{m}\log\left(D(G(z^{(i)}))\right)$$
10:    Update the parameters of the generator
```

1. [1pt] **Implementation:** Fill in the `gan_training_loop_regular` 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. Note that in the original GAN paper, we want to train the Generator by minimizing $\log\left(1 - D(G(z^{(i)}))\right)$ in an effort to generate better fakes. However, this was shown by Goodfellow to not provide sufficient gradients, especially early in the learning process. As a fix, we instead wish to maximize $\log\left(D(G(z^{(i)}))\right)$, which is also refered as the non-saturating GAN Loss.

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 20 minutes 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. Least-squares GAN is one of the many techniques [Mao et al., 2017]. Fill in the `gan_training_loop_leastsquares` function in the GAN section of the notebook. Compared to the regular GAN training loop you have implemented, you will only need to modify the discriminator loss and the generator loss to the following:

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

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

   After the implementation, try turn on the `least_squares_gan` flag in the `args_dict` and train the model again. Are you able to stabilize the training? Briefly explain in 1∼2 sentences why the least squares GAN can help. You are welcome to check out the related blog posts for LSGAN.

Part 2: Graph Convolution Networks

For this part of the assignment, you will implement the vanilla version of Graph Convolution Networks (GCN) Kipf and Welling [2016] and Graph Attention Networks (GAT) Veličković et al. [2018].

Basics of GCN:

Recall from the lecture, the goal of a GCN is to learn a function of signals/features on a graph $G = (V, E)$, which takes as inputs:

1. the input features of each node, $x_i \in \mathcal{R}^F$ (in matrix form: $X \in \mathcal{R}^{|V| \times F}$)
2. some information about the graph structure, typically the adjacency matrix A

Each convolutional layer can be written as $H^{(l+1)} = f(H^{(l)}, A)$, for some function $f()$. The $f()$ we are using for this assignment is in the form of $f(H^{(l)}, A) = \sigma(\hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2}H^{(l)}W^{(l)})$, where $\hat{A} = A + Identity$ and $\hat{D}$ is diagonal node degree matrix ($\hat{D}^{-1}\hat{A}$ normalizes $\hat{A}$ such that all rows sum to one). Let $\tilde{A} = \hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2}$. The GCN we will implement takes two convolution layers, $Z = f(X, A) = softmax(\tilde{A} \cdot Dropout(ReLU(\tilde{A}XW^{(0)})) \cdot W^{(1)})$

Basics of GAT:

Graph Attention Network (GAT) is a novel convolution-style neural network. It operates on graph-structured data and leverages masked self-attentional layers. In this assignment, we will implement the graph attention layer.

Dataset:

The dataset we used for this assignment is Cora Sen et al. [2008]. Cora is one of standard citation network benchmark dataset (just like MNIST dataset for computer vision tasks). It that consists of 2708 scientific publications and 5429 links. Each publication is classified into one of 7 classes. Each publication is described by a word vector (length 1433) that indicates the absence/presence of the corresponding word. This is used as the features of each node for our experiment. The task is to perform node classification (predict which class each node belongs to).

Experiments:

Open [GCN notebook link] on Colab and answer the following questions.

1. [1pt] **Implementation of Graph Convolution Layer**

   Complete the code for `GraphConvolution()` Class

2. [1pt] **Implementation of Graph Convolution Network**

   Complete the code for `GCN()` Class

3. [0.5pt] **Train your Graph Convolution Network**

   After implementing the required classes, now you can train your GCN. You can play with the hyperparameters in `args`.

4. [2pt] **Implementation of Graph Attention Layer**

   Complete the code for GraphAttentionLayer() Class

5. [0.5pt] **Train your Graph Convolution Network**

   After implementing the required classes, now you can train your GAT. You can play with the hyperparameters in args.

6. [0.5pt] **Compare your models**

   Compare the evaluation results for Vanilla GCN and GAT. Comment on the discrepancy in their performance (if any) and briefly explain why you think it's the case (in 1-2 sentences).

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.

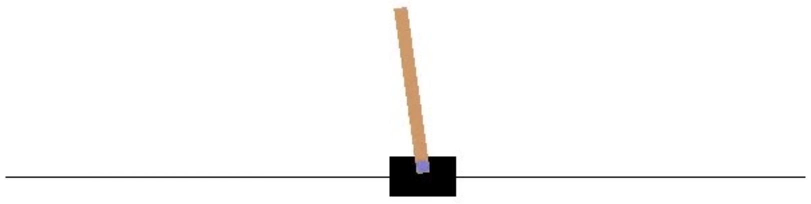


Figure 1: *Image of the CartPole Balancing game from OpenAI Gym.Brockman et al. [2016]*

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:

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

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. **Include your code snippet in your write-up**.

2. [1pt] **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. **Include your code snippet in your write-up**.

3. [2pt] **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! **List the hyperparameters' value you choose, your epsilon decay rule and summarize your final results from the visualizations in a few sentences in your write-up**.

What you need to submit

- Your code files: `a4-dcgan.ipynb`, `a4-gcn.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, GCN 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. [2020])
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

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

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, 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, 2020.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016. URL `http://arxiv.org/abs/1609.02907`.

Xudong Mao, Qing Li, Haoran Xie, Raymond YK Lau, Zhen Wang, and Stephen Paul Smolley. Least squares generative adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2794–2802, 2017.

Prithviraj Sen, Galileo Mark Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. Collective classification in network data. *AI Magazine*, 29(3):93–106, 2008. URL `http://www.cs.iit.edu/~ml/pdfs/sen-aimag08.pdf`.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. *International Conference on Learning Representations*, 2018. URL `https://openreview.net/forum?id=rJXMpikCZ`. accepted as poster.