ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

ỨNG DỤNG XỬ LÝ ẢNH SỐ VÀ VIDEO SỐ

Practice 03
Generative Adversarial Networks

Giảng viên hướng dẫn Thầy Lý Quốc Ngọc
Thầy Nguyễn Mạnh Hùng
Thầy Phạm Minh Hoàng

Sinh viên thực hiện Võ Nguyễn Hoàng Kim
21127090

PHỤ LỤC

A. BẢNG TỰ ĐÁNH GIÁ

Yêu cầu	Mức độ hoàn thành
Mô hình: GAN	100%
Tập dữ liệu: MNIST	100%
Thực nghiệm và báo cáo	100%

B. TỔNG QUAN

I. Mô hình GAN

1. GAN là gì

- GAN là lớp mô hình có mục tiêu là tạo ra dữ liệu giả giống với thật. GAN được viết tắt từ cụm từ Generative Adversarial Networks tức là một mạng sinh đối nghịch (Generative tương ứng với sinh và Adversarial là đối nghịch). Sở dĩ GAN có tên gọi như vậy là vì kiến trúc của nó bao gồm hai mạng có mục tiêu đối nghịch nhau đó là Generator và Discriminator [1].

2. Cấu trúc GAN

- Cấu trúc mạng GAN được hình thành từ hai mạng là Generator và Discriminator. Trong khi Generator luôn học cách sinh ra mô hình giả làm sao để gần như giống hệt mô hình thật thì Discriminator lại cố gắng học cách phân biệt giữa dữ liệu giả được sinh ra từ Generator và dữ liệu thật. Thông qua quá trình huấn luyện, cả Generator và Discriminator cùng cải thiện được khả năng của mình.
    - Generator là mạng sinh ra dữ liệu, có input là noise (random vector) có z chiều và output là ảnh được sinh ra.
    - Discriminator là mạng để phân biệt xem dữ liệu là thật (dữ liệu từ dataset) hay giả (dữ liệu sinh ra từ Generator), đây là bài toán binary classification.
- Generator và Discriminator tương tự như hai người chơi trong bài toán zero-sum game trong lý thuyết trò chơi. Ở trò chơi này thì hai người chơi xung đột lợi ích, thiệt hại của người này chính là lợi ích của người kia. Mô hình Generator tạo ra dữ liệu giả tốt hơn sẽ làm cho Discriminator phân biệt khó hơn và khi Discriminator phân biệt tốt hơn thì Generator cần phải tạo ra ảnh giống thật hơn để qua mặt Discriminator. Trong zero-sum game, người chơi sẽ có chiến lược riêng của mình, đối với Generator thì đó là sinh ra ảnh giống thật và Discriminator là phân loại ảnh thật/giả. Sau các bước ra quyết định, cả Generator và Discriminator sẽ đạt được trạng thái cân bằng (tức là đi tiếp cũng sẽ không làm tăng cơ hội thắng) [1].

II. Tập dữ liệu

1. MNIST

- MNIST là bộ dữ liệu được sử dụng rộng rãi cho nhiệm vụ phân loại chữ số viết tay. Nó bao gồm 70.000 hình ảnh thang độ xám được dán nhãn gồm các chữ số viết tay, mỗi hình có kích thước 28×28 pixel. Tập dữ liệu được chia thành 60.000 hình ảnh huấn luyện và 10.000 hình ảnh thử nghiệm. Có 10 lớp (một lớp cho mỗi chữ số trong số 10 chữ số) [2]. Dưới đây là một số mẫu trong bộ dữ liệu

C. THỰC NGHIỆM

I. Thiết lập Google Colab và môi trường

- Trong bài này, ta sử dụng tài nguyên là CPU để huấn luyện mô hình và chạy chương trình.
- Cấp quyền truy cập vào Google Drive để truy xuất, lưu trữ mã nguồn và tập dữ liệu.
- Tải và chuẩn hóa tập dữ liệu MNIST.
    - Sử dụng hàm **torchvision.dataset.MNIST()** để tải bộ dữ liệu MNIST, thực hiện chuẩn hóa nhờ vào tham số *transform* trong hàm, cụ thể:
        - Sử dụng hàm **transforms.ToTensor()** chuyển đổi hình ảnh thành tensor và **transforms.Normalize()** chuẩn hóa dữ liệu về trong khoảng $[-1, 1]$ thông qua độ lệch chuẩn và giá trị trung bình là 0.5.
- Sau khi tải và chuẩn hóa tập dữ liệu, tạo một đối tượng *dataloader* trong PyTorch (sử dùn hàm **torch.utils.data.DataLoader()**), cho phép chúng ta duyệt qua dữ liệu theo các batc, với kích thước của mỗi batch là 64 và được xáo trộn dữ liệu trước khi chia thành các batch.

II. Xây dựng mô hình GAN

1. Discriminator

- Trong phạm vi mã nguồn này, chúng ta chỉ xây dựng mạng Discriminator đơn giản với hai lớp liên kết đầy đủ (fully connected). Như vậy, mạng này chỉ có một lớp ẩn.
- Viết lại hàm khởi tạo **__init__** cho mạng Discriminator với số chiều đầu vào là *input_dim* được đặt mặc định là 784, điều này tương ứng với số pixel trong ảnh MNIST ($28 \times 28 = 784$)
    - Trong hàm này, nó thực hiện gọi phương thức khởi tạo của lớp cha **nn.Module** để thiết lập các thuộc tính cơ bản của lớp Discriminator.
    - Tiếp tục định nghĩa một lớp liên kết đầy đủ (FC1) với kích thước đầu vào là *input_dim* và kích thước đầu ra là 128.
    - Khởi tạo hàm kích hoạt LeakyReLU với hệ số là 0.2, hàm này sẽ được sử dụng sau lớp FC1 để giúp mô hình học được các đặc trưng phi tuyến tính.
    - Định nghĩa một lớp liên kết đầy đủ khác (FC2) với kích thước đầu vào là 128 và chỉ có 1 đầu ra. Đầu ra của lớp này sẽ được coi là điểm số của ảnh, dùng để đánh giá xem ảnh đó có phải là ảnh thật (real) hay ảnh giả (fake).
- Ngoài hàm khởi tạo, ta còn viết lại phương thức **forward** cho lớp, xác định phương thức truyền xuôi (truyền tiến) của mô hình Discriminator. Phương thức này có tham số đầu vào là *x* đại diện cho đầu vào của mạng, là tensor chứa các ảnh đầu vào.
    - Nó thực hiện chuyển đổi tensor *x* để phù hợp với kích thước đầu vào của lớp FC1, cụ thể: chuyển từ dạng $batch\_size \times 1 \times 28 \times 28$ thành dạng $batch\_size \times 784$.
    - Tiếp tục truyền xuôi qua lớp FC1 và áp dụng hàm kích hoạt LeakyReLU để tính toán các đặc trưng trong ảnh.
    - Truyền xuôi tiếp qua lớp FC2 để tính toán điểm số.
    - Cuối cùng, sử dụng hàm kích hoạt Sigmoid để đưa điểm số có được ở bước trên về lại trong khoảng $[0, 1]$, điều này sẽ tương ứng với xác suất của ảnh là thật (real) hay giả (fake).
    - Giá trị có được từ hàm Sigmoid trên cũng chính là kết quả trả về của phương thức **forward.**

2. Generator

- Tương tự như Discriminator, trong phạm vi bài này, ta chỉ xây dựng mạng Generator đơn giản với hai lớp liên kết đầy đủ (fully connected). Như vậy, mạng này cũng chỉ có một lớp ẩn.
- Viết lại hàm khởi tạo **__init__** cho mạng Generator với số chiều của vector nhiễu ngẫu nhiên *z_dim* được đặt mặc định là 100.

    - Trong hàm này, nó thực hiện gọi phương thức khởi tạo của lớp cha **nn.Module** để thiết lập các thuộc tính cơ bản của lớp Generator.
    - Tiếp tục định nghĩa một lớp liên kết đầy đủ (FC1) với kích thước đầu vào là *z_dim* và kích thước đầu ra là 128.
    - Khởi tạo hàm kích hoạt LeakyReLU với hệ số là 0.2, hàm này sẽ được sử dụng sau lớp FC1 để giúp mô hình học được các đặc trưng phi tuyến tính.
    - Định nghĩa một lớp liên kết đầy đủ khác (FC2) với kích thước đầu vào là 128 và có 784 đầu ra. Kích thước đầu ra này tương ứng với số pixel trong mỗi ảnh MNIST (28×28).
- Ngoài hàm khởi tạo, ta còn viết lại phương thức **forward** cho lớp, xác định phương thức truyền xuôi (truyền tiến) của mô hình Generator. Phương thức này có tham số đầu vào là *x* đại diện cho đầu vào của mạng, trong lớp này, *x* thường là vector nhiễu ngẫu nhiên.
    - Nó thực hiện truyền xuôi qua lớp FC1 và áp dụng hàm kích hoạt LeakyReLU để tính toán các đặc trưng trong ảnh.
    - Truyền xuôi tiếp qua lớp FC2 để tính toán các giá trị đầu ra.
    - Sử dụng hàm kích hoạt Tanh để đưa giá trị đầu ra có được ở bước trên về lại trong khoảng $[-1, 1]$, điều này giống như phân phối của ảnh MNIST.
    - Cuối cùng, nó thực hiện chuyển đổi tensor đầu ra thành dạng hình ảnh có kích thước $batch\_size \times 1 \times 28 \times 28$ để phù hợp với định dạng ảnh của MNIST.
    - Tensor biểu diễn cho ảnh được tạo ra (ở bước trên) cũng chính là kết quả trả về của phương thức **forward.**

3. Hàm mất mát

- Thiết lập hàm tối ưu cho cả Discriminator và Generator, ở đây chúng ta sử dụng thuật toán Stochastic Gradient Descent (SGD) để cập nhật các tham số cho Discriminator với tốc độ học (learning rate – lr) là 0.3.
- Thiết lập hàm mất mát, ở đây, chúng ta sử dụng hàm mất mát Binary Cross-Entropy (BCE) cho mô hình GAN.
- Đặt kích thước cố định cho *batch_size*, là kích thước của mỗi batch dữ liệu được sử dụng trong quá trình huấn luyện, nó được đặt là 64 trong mã nguồn này.
- Ngoài ra, ta còn khởi tạo hai biến là *lab_real* và *lab_fake* lần lượt đại diện cho nhãn các ảnh thật (được thiết lập là 1) và ảnh giả (được thiết lập là 0). Kích thước của chúng đều có dạng là $batch\_size \times 1$

4. Huấn luyện mô hình

- Mô hình được huấn luyện với số epochs là 100 (*epoch_num = 100).* Thực hiện vòng lặp **for** để huấn luyện mô hình qua các epoch.
- Đối với mỗi epoch, tiếp tục sử dụng vòng lặp **for** để duyệt qua từng batch dữ liệu trong *dataloader*, ở đây ta sử dụng hai biến là *i* cho chỉ số của batch và *data* cho dữ liệu tương ứng. Việc huấn luyện này được chia làm hai bước:
    - Bước 1: huấn luyện Discriminator:
        - Trước hết lấy một batch dữ liệu ảnh thật từ tập dữ liệu (*x_real*) và đặt lại toàn bộ gradient của tất cả tham số trong *optimizerD* về lại 0 (vì chúng ta không muốn gradient tích lũy qua các batch trước đó).
        - Đưa dữ liệu liệu thật vào mạng Discriminator và tính toán kết quả dự đoán của nó, lưu trữ vào *D_x.* Tính toán độ mất mát cho dữ liệu thật *lossD_real* bằng cách so sánh kết quả dự đoán ở bước trên với nhãn thật.
        - Tạo dữ liệu giả bằng cách sinh ra các vector nhiễu ngẫu nhiên *z* và truyền vào mạng Generator để tạo ra ảnh giả *x_gen*. Đưa dữ liệu giả vào mạng Discriminator và tính

toán kết quả dự đoán của nó, lưu trữ vào D_G_z. Tính toán độ mất mát cho dữ liệu giả *lossD_fake* bằng cách so sánh kết quả dự doán với nhãn giả *lab_fake.*

- Tổng hợp độ mất mát *lossD* của mạng Discriminator bằng tổng hai giá trị của *lossD_real* và *lossD_fake*.
- Gọi hàm backward thực hiện lan truyền ngược để tính toán gradient của hàm mất mát *lossD* theo các tham số của Discriminator. Cập nhật lại các tham số của mạng Discriminator dựa trên gradient tính toán được ở bước trên (thuật toán SGD sẽ được áp dụng để điều chỉnh các tham số sao cho hàm mất mát được giảm đáng kể).

- Bước 2: Huấn luyện Generator:
  - Tương tự như Discriminator, ta đặt lại toàn bộ gradient của tất cả tham số trong *optimizerG* về lại 0 (vì chúng ta không muốn gradient tích lũy qua các batch trước đó).
  - Sinh ra các vector nhiễu ngẫu nhiên *z* và truyền vào mạng Generator để tạo ra các ảnh giả *x_gen*. Đưa dữ liệu giả vào mạng Discriminator và tính toán kết quả dự đoán của nó, lưu trữ vào *D_G_z*.
  - Tính toán độ mất mát cho Generator bằng cách so sánh kết quả dự đoán của Discriminator ở bước trên *D_G_z* với nhãn thật *lab_real* (Vì mục tiêu của nó là đánh lừa Discriminator để nghĩ rằng các ảnh giả là ảnh thật).
  - Tương tự như Discriminator, ta cũng gọi hàm backward thực hiện lan truyền ngược để tính toán gradient của hàm mất mát *lossG* theo các tham số của Generator. Cập nhật lại các tham số của mạng Generator dựa trên gradient tính toán được ở bước trên (thuật toán SGD sẽ được áp dụng để điều chỉnh các tham số sao cho hàm mất mát được giảm đáng kể).

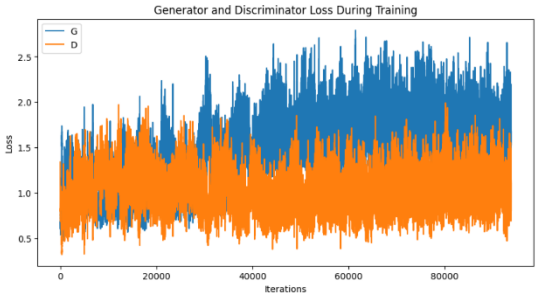
III. Kết quả – Nhận xét

1. Thay đổi hàm tối ưu

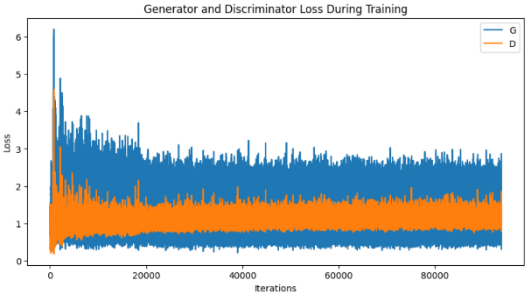
Khi thực nghiệm mã nguồn xây dựng mô hình GAN, ta có hai thuật toán dành cho hàm tối ưu là SGD và Adam. Do đó, để so sánh kết quả này, ta thực nghiệm xây dựng mô hình với *epoch_num* là 100, kích thước của vector nhiễu ngẫu nhiên *z_dim* là 100, *batch_size* là 64, ta thu được các kết quả sau:

a. Độ lỗi

Adam



SGD



Đối với Generator:

- Từ biểu đồ trên, có thể thấy, tuy rằng độ lỗi mà mô hình đạt khi sử dụng Adam thấp hơn so với SGD ở những bước đầu (khi Adam chỉ khoảng gần 1.8 thì SGD lại đạt đến mức 6). Tuy nhiên, độ lỗi của SGD lại được cải thiện, giảm dần qua các lần lặp sau, thì Adam lại tăng lên. Dù độ lỗi mà Adam tăng không cao (so với lần lặp tương ứng của SGD thì giá trị độ lỗi gần như bằng nhau), song nó lại cho thấy rằng mô hình đang học không hiệu quả.

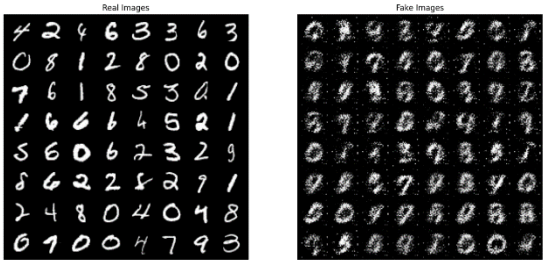
- Tuy nhiên, có thể nhận thấy ở đây, dù sử dụng thuật toán nào thì ở mô hình GAN này độ lỗi đều xảy ra biến động liên tục, nhưng đối với Adam, sự biến động này trở nên mạnh mẽ và độ chênh lệch của mức biến động lớn hơn rất nhiều so với SGD.

Đối với Discriminator

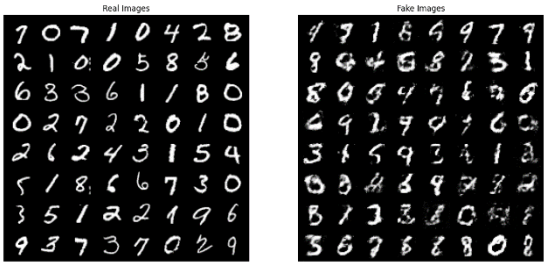
- Tương tự như Generator, độ mất mát mà Discriminator gặp phải cũng có vấn đề tương tự ở Adam và SGD. Dù ở những bước đầu, độ mất mát mà Adam có được thấp hơn rất nhiều so với SGD, song nó vẫn giữ nguyên giá trị mất mát đó cho ngừng lần lặp sau mà không có chút cải thiện nào. Ngược lại, SGD dù cho độ mất mát ban đầu rất cao những trong quá trình huấn luyện, độ mất mát đã được giảm đi rất nhiều, từ khoảng 4.5 đã giảm xuống còn gần 1.5.
- Cũng như Generator, độ mất mát của Discriminator khi sử dụng Adam cho ra biến động rất lớn đối với mỗi lần lặp. Độ biến động này lớn hơn rất nhiều so với SGD khi độ mất mát của nó gần như đều nhau qua mỗi lần lặp.

b. Ảnh đầu ra từ Generator

Adam



SGD



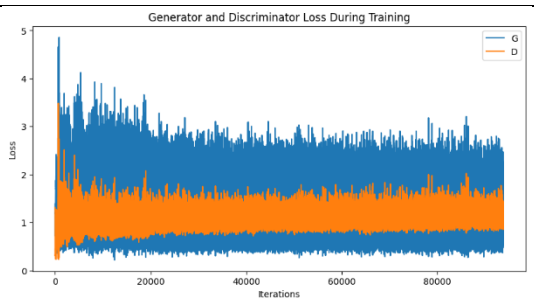
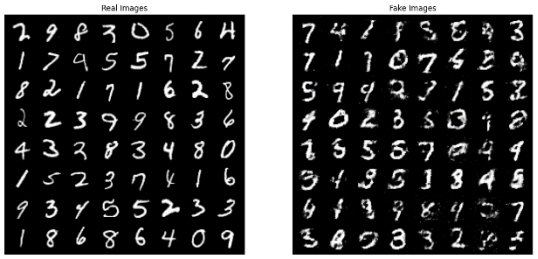
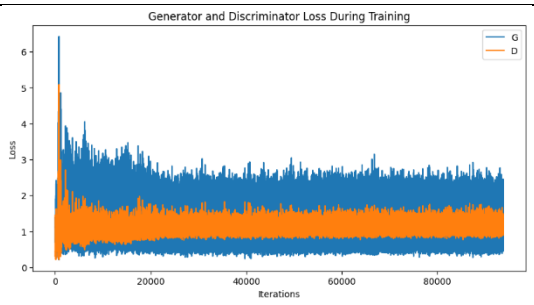
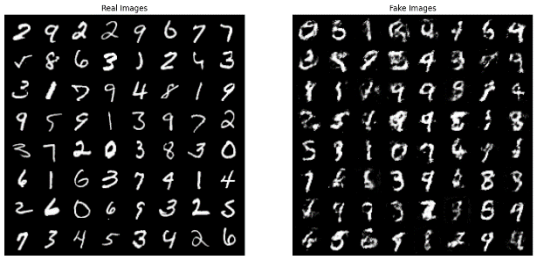
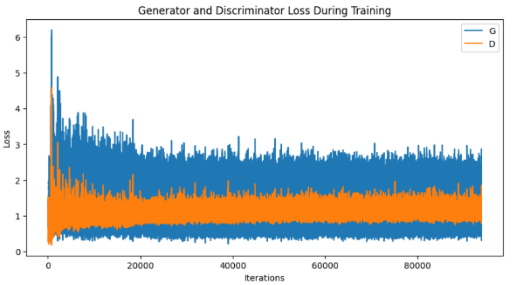
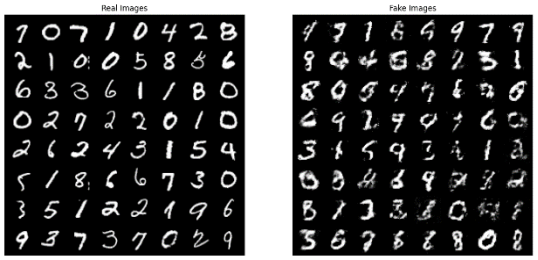
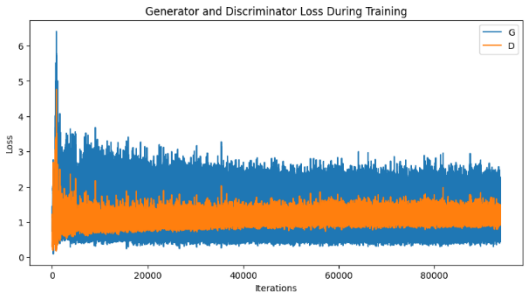
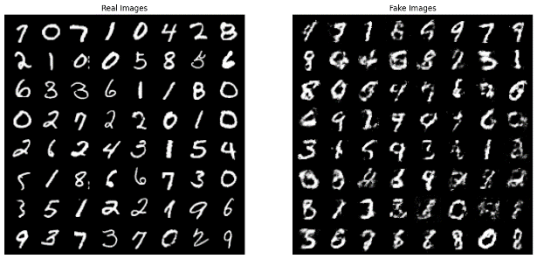
- Từ độ lỗi trên, có thể thấy rằng mô hình vẫn chưa được tối ưu (cả về Discriminator hay Generator), nên vì thế mà kết quả của Generator cho ra được không thể chính xác được so với ảnh thật. Như vậy, ở mục này, ta chỉ so sánh ảnh giả mà Generator cho được dựa vào hình dạng của các chữ số mà nó tạo được trong ảnh kết quả. Ảnh này được ghi lại vào lần lặp cuối cùng của quá trình huấn luyện mô hình GAN.
- Dễ dàng quan sát thấy, nếu sử dụng Adam, kết quả mà Generator cho ra được vẫn còn tồn tại điểm nhiễu rất nhiều, các ký số vẫn chưa được hình thành (dù là hình dạng đơn giản).
- Tuy nhiên, đối với SGD, với các tham số được truyền và sử dụng tương tự, nhưng nó đã cho ra được hình dáng của các chữ số (như các số 3, 7, 8, 9).

→ Từ thực nghiệm có được và so sánh trên, có thể thấy, trong mã nguồn này, SGD đã làm tốt hơn so với Adam. Do đó, để tiến hành thay đổi chiều vector nhiễu ngẫu nhiên ở mục dưới, ta sử dụng SGD để có thể so sánh đầu ra của Generator dễ hơn cũng như độ lỗi của cả Discriminator và Generator.

2. Thay đổi kích thước của vector nhiễu ngẫu nhiên

Để dễ dàng quan sát, chúng ta sử dụng cùng một loại thuật toán SGD cho hàm tối ưu cả Generator và Discriminator, xây dựng mô hình với *epoch_num* là 100, *batch_size* là 64. Ta lần lượt thay đổi kích thước của vector nhiễu ngẫu nhiên (random vector noise) *z_dim* với các giá trị là 20, 50, 100, 200. Kết quả thu được như sau:

z_dim	Đồ thị mất mát	Kết quả từ Generator

20		
50		
100		
200		

a. Độ lỗi

- Việc đánh giá độ lỗi của các khảo sát này dựa vào mắt thường là một việc khó khăn và ít có tính khách quan. Chúng ta chỉ có thể thấy được một số yếu tố cơ bản, cụ thể như sau:
    - Đối *z_dim* là 20, độ lỗi ban đầu mà mô hình đạt được (cả Generator và Discriminator) thấp hơn so với các kích thước khác, chỉ dừng ở mức khoảng 4.8 cho Generator và 3.5 cho Discriminator.
    - Tuy nhiên, ở những lần lặp sau, giá trị độ lỗi (Generator) mà *z_dim = 20* lại không được cải thiện quá nhiều, vẫn quanh quẩn ở mức 3 đến 4, trong khi các *z_dim* khác lại được cải thiện khá tốt trong những lần lặp sau (khoảng mức 3 đổ xuống).

b. Kết quả từ Generator

- Dựa vào độ lỗi trên có thể nhận thấy rằng mô hình chưa đạt điểm đến cân bằng để có thể tối ưu, do đó, kết quả mà ta nhận được từ Generator không thể so sánh về độ chính xác so với ảnh thật. Vì thế, ở đây, ta chỉ có thể nhận xét dựa trên hình dạng của các ký số được hình thành trong ảnh kết quả cuối cùng.

- Nhìn vào kết quả thực nghiệm, có thể thấy rằng kích thước của vector nhiễu ngẫu nhiên có ảnh hưởng lớn đến kết quả mà Generator đem lại. Đối với *z_dim=20*, hình dạng của các chữ số được thể hiện gần như hoàn chỉnh và khá rõ. Trái với *z_dim=200*, dù các vệt đã hiện lên, song hình dáng của các chữ số vẫn còn khá mơ hồ, một số vẫn còn những điểm nhiễu chưa được thành hình. Cách so sánh này gần như đúng với cả trường hợp *z_dim = 50* và *z_dim = 100* khi đem so sánh với nhau.
- Có thể kết luận rằng, *z_dim* càng lớn, thì mạng Generator càng có khả năng tạo ra mô hình phức tạp và đa dạng hơn, và ngược lại cho *z_dim* bé. Đối với tập dữ liệu MNIST, độ phức tạp của nó không quá cao nên việc đòi hỏi một kích thước lớn cho vector nhiễu ngẫu nhiên là không cần thiết. Do đó, giá trị mà *z_dim* sử dụng ở đây chỉ nên nằm trong khoảng 50 đến 100 là hợp lý.

D. TÀI LIỆU THAM KHẢO

[1] P. D. Khanh, "Bài 43 - Model GAN," Khanh's blog, 13 July 2020. [Online]. Available: https://phamdinhkhanh.github.io/2020/07/13/GAN.html.

[2] Apache MXNet (Incubating), "Hand-written Digit Recognition," Apache MXNet (Incubating), [Online]. Available: https://mxnet.apache.org/versions/1.5.0/tutorials/gluon/mnist.html.