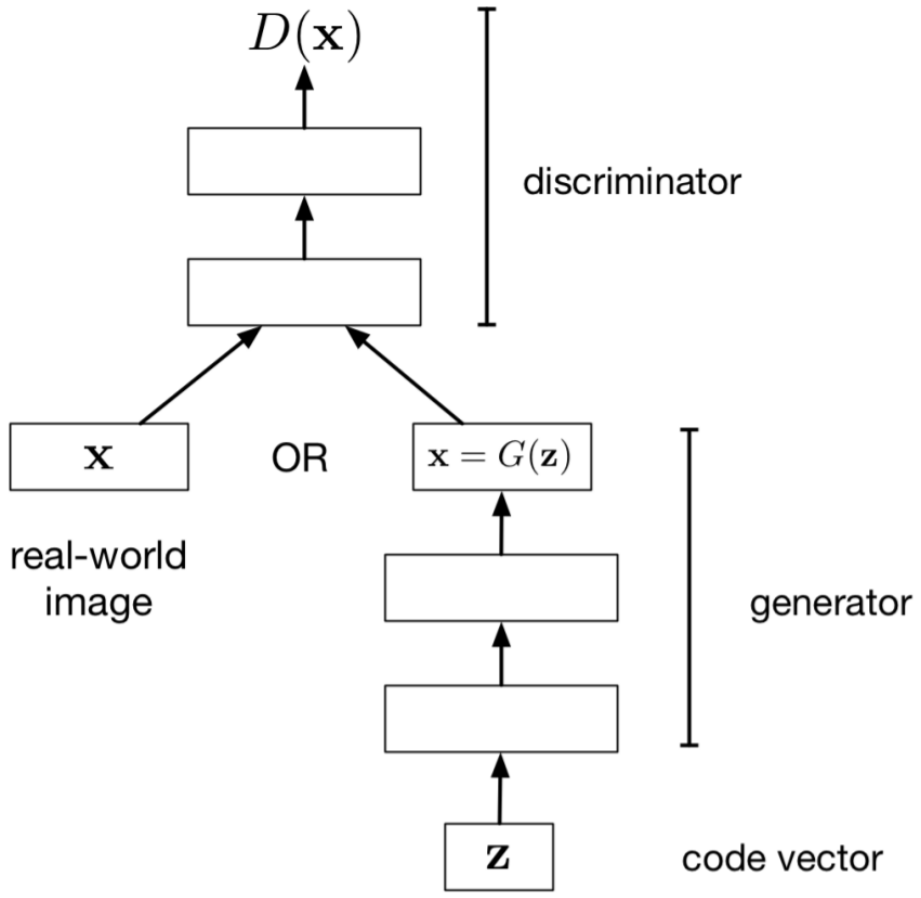
Generative Adversarial Net

Tóm tắt

Nhóm tác giả đề xuất một 'framework' mới để ước tính **các mô hình sinh (generative models)** thông qua một quá trình **đối nghịch (adversarial)**, trong đó chúng ta huấn luyện đồng thời 2 mô hình: **một mô hình sinh** G nắm bắt phân phối dữ liệu và một **mô hình phân biệt** D xác định một mẫu đến từ dữ liệu thật hay được sinh từ mô hình G. Việc huấn luyện mô hình G là tối đa hóa xác suất mô hình D mắc lỗi. **Mô hình sinh** có thể được coi là tương tự như một nhóm làm giả, cố gắng sản xuất tiền giả và sử dụng nó mà không bị phát hiện, trong khi **mô hình phân biệt** tương tự như cảnh sát, cố gắng phát hiện tiền giả. Sự cạnh tranh này thúc đẩy cả hai đội phải cải tiến phương pháp của mình cho đến khi không thể phân biệt được hàng giả với hàng thật.

'Framework' này tương ứng với **minimax** cho trò chơi hai người chơi. Trong trường hợp mô hình G và D được xây dựng bởi các **perceptron nhiều lớp**, thì toàn bộ mô hình có thể được huấn luyện với **lan truyền ngược (backprop)**. Không cần bất kỳ chuỗi Markov nào trong quá trình huấn luyện hoặc tạo mẫu. Các thử nghiệm chứng

minh tiềm năng của khuôn khổ thông qua đánh giá định tính (qualitative) và định lượng (quantitative) các mẫu được tạo.



Nguồn: Bài 18 - Khóa "Neural Networks and Deep Learning" - Đại học Toronto

Cho đến nay, những thành công nổi bật nhất trong học sâu đều liên quan đến các **mô hình phân biệt**, thường là những mô hình ánh xạ đầu vào lớn, có nhãn. Những thành công nổi bật này chủ yếu dựa trên các thuật toán lan truyền ngược (backprop) và 'dropout', sử dụng các đơn vị tuyến tính (linear units). Các **mô hình sinh sâu** chịu ít tác động hơn, do khó khăn trong việc tính toán xác suất xảy và các chiến lược liên quan, và do khó tận dụng lợi ích của các đơn vị tuyến tính. Do đó, nhóm tác giả đề xuất một quy trình ước tính các **mô hình sinh (generative model)** mới như đã đề cập đến bên trên để khắc phục những khó khăn này.

Những nghiên cứu trước đó có liên quan

Deep Boltzmann machines: [26] Salakhutdinov, R. and Hinton, G. E. (2009). Deep Boltzmann machines. In AISTATS'2009, pages 448 – 455.

Prominent recent work in this area:

- Generalized denoising auto-encoders: [4] Bengio, Y., Yao, L., Alain, G., and Vincent, P. (2013b). Generalized denoising auto-encoders as generative models. In NIPS26. Nips Foundation.
  Generative stochastic network (GSN) framework: [5] Bengio, Y., Thibodeau-Laufer, E., and Yosinski, J. (2014a). Deep generative stochastic networks trainable by backprop. In ICML'14.

Deep belief networks (DBNs): [16] Hinton, G. E., Osindero, S., and Teh, Y. (2006). A fast learning algorithm for deep belief nets. Neural Computation, 18, 1527–1554.

Alternative criteria:

- Score matching: [18] Hyvarinen, A. (2005). Estimation of non-normalized statistical models using score matching. J. Machine Learning Res., 6.
- Noise-contrastive estimation (NCE): [13] Gutmann, M. and Hyvarinen, A. (2010). Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In AISTATS'2010.

Some models have learning rules very similar to score matching applied to RBMs: [30] Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P.-A. (2008). Extracting and composing robust features with denoising autoencoders. In ICML 2008.

Examples of training a generative machine by back-propagating:

- Auto-encoding variational Bayes: [20] Kingma, D. P. and Welling, M. (2014). Auto-encoding variational bayes. In Proceedings of the International Conference on Learning Representations (ICLR).
- Stochastic backpropagation: [24] Rezende, D. J., Mohamed, S., and Wierstra, D. (2014). Stochastic backpropagation and approximate inference in deep generative models. Technical report, arXiv:1401.4082.

Tác động đến các nghiên cứu sau

Tác động của bài báo đến những nghiên cứu sau này chủ yếu liên quan đến các biến thể của GAN:

DCGAN: **Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks -** https://arxiv.org/pdf/1511.06434.pdf

Conditional GAN: **Conditional Generative Adversarial Nets -** https://arxiv.org/pdf/1411.1784.pdf

BigGAN: **Large Scale GAN Training for High Fidelity Natural Image Synthesis -** https://arxiv.org/pdf/1809.11096.pdf

StyleGAN: **A Style-Based Generator Architecture for Generative Adversarial Networks -** https://arxiv.org/pdf/1812.04948.pdf

CycleGAN: **Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks -** https://arxiv.org/pdf/1703.10593.pdf

Pix2Pix: **Image-to-Image Translation with Conditional Adversarial Networks -** https://arxiv.org/pdf/1611.07004.pd

... Còn nữa

Tổng hợp các bài báo có liên quan đến GAN và theo em nghĩ các bài báo này đều nhận được sự ảnh hưởng từ bài báo chúng ta đang tìm hiểu:
https://github.com/hindupuravinash/the-gan-zoo

Hàm mục tiêu và ý tưởng

In other words, D and G play the following two-player minimax game with value function $V(G, D)$:

$$\min_G \max_D V(D, G) = \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]. \quad (1)$$

Bộ phân biệt D sẽ cố gắng tối đa hóa biểu thức bên phải dấu '=', tương đương với việc gán đúng nhãn, tức là: Nó sẽ cố gắng tối đa hóa xác suất của dữ liệu thật $D(x)$ và tối thiểu hóa xác suất của dữ liệu giả $D(G(\boldsymbol{z}))$. Trong đó:

- Tối thiểu hóa $D(G(\boldsymbol{z}))$ tương tự tối đa hóa $1 - D(G(\boldsymbol{z}))$. Vậy, hàm mất mát (loss function) của Bộ phân biệt D được viết lại thành:

$$\max_D V(D) = \text{"} + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log 1 - D(G(\boldsymbol{z}))]$$

Trong khi đó thì bộ sinh G vì chỉ cố khiến cho D nghĩ rằng dữ liệu nó sinh ra (giả) là thật nên sẽ tối đa hóa $D(G(\boldsymbol{z}))$ tương đương với tối thiểu hóa $1 - D(G(\boldsymbol{z}))$. Vậy, hàm mất mát của Bộ sinh G là:

$$\min_G V(G) = \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log 1 - D(G(\boldsymbol{z}))]$$

Do đó, ta có thể viết lại hàm mất mát của mô hình GAN:

$$\min_G \max_D V(D, G) = \mathbb{E}_{\boldsymbol{x} \sim p_{data(\boldsymbol{x})}}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log 1 - D(G(\boldsymbol{z}))]$$

- Có 1 vấn đề với công thức này chính là độ bão hòa (saturation). Trong phân loại, khi dự đoán (prediction) thực sự sai:
  - Dùng "Logistic + Squared error" sẽ nhận được tín hiệu đạo hàm (gradient signal) yếu.
  - Dùng "Logistic + Cross-entropy" sẽ nhận được tín hiệu đạo hàm mạnh.
- Khi mới học, mẫu (sample) được sinh ra thực sự tệ, D có khả năng chối các mẫu này với độ tin cậy cao vì chúng khác biệt rõ ràng với dữ liệu huấn luyện. Trong trường hợp này, $\log(1 - D(G(\boldsymbol{z})))$ bão hòa. Thay vì huấn luyện G để cực tiểu hóa $\log(1 - D(G(\boldsymbol{z})))$, chúng ta có thể huấn luyện G để cực đại hóa $\log D(G(\boldsymbol{z}))$.

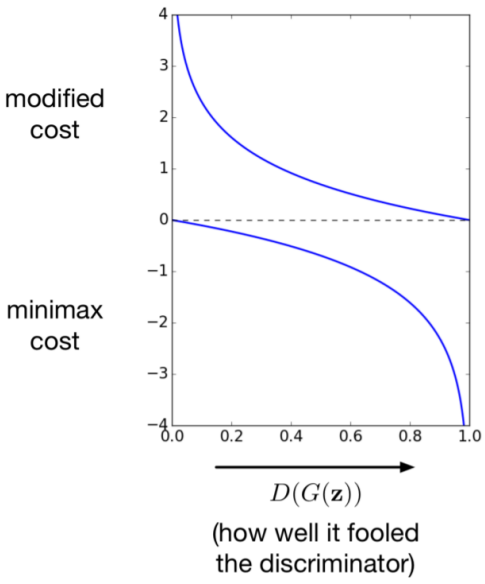
- Original minimax cost:

$$\mathcal{J}_G = \mathbb{E}_{\mathbf{z}}[\log(1 - D(G(\mathbf{z})))]$$

- Modified generator cost:

$$\mathcal{J}_G = \mathbb{E}_{\mathbf{z}}[-\log D(G(\mathbf{z}))]$$

- This fixes the saturation problem.

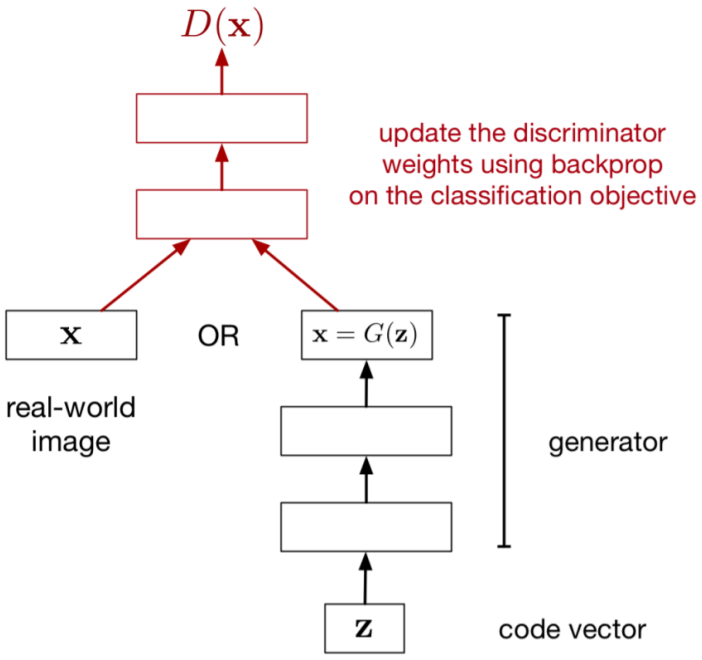


Nguồn: Bài 18 - Khóa "Neural Networks and Deep Learning" - Đại học Toronto

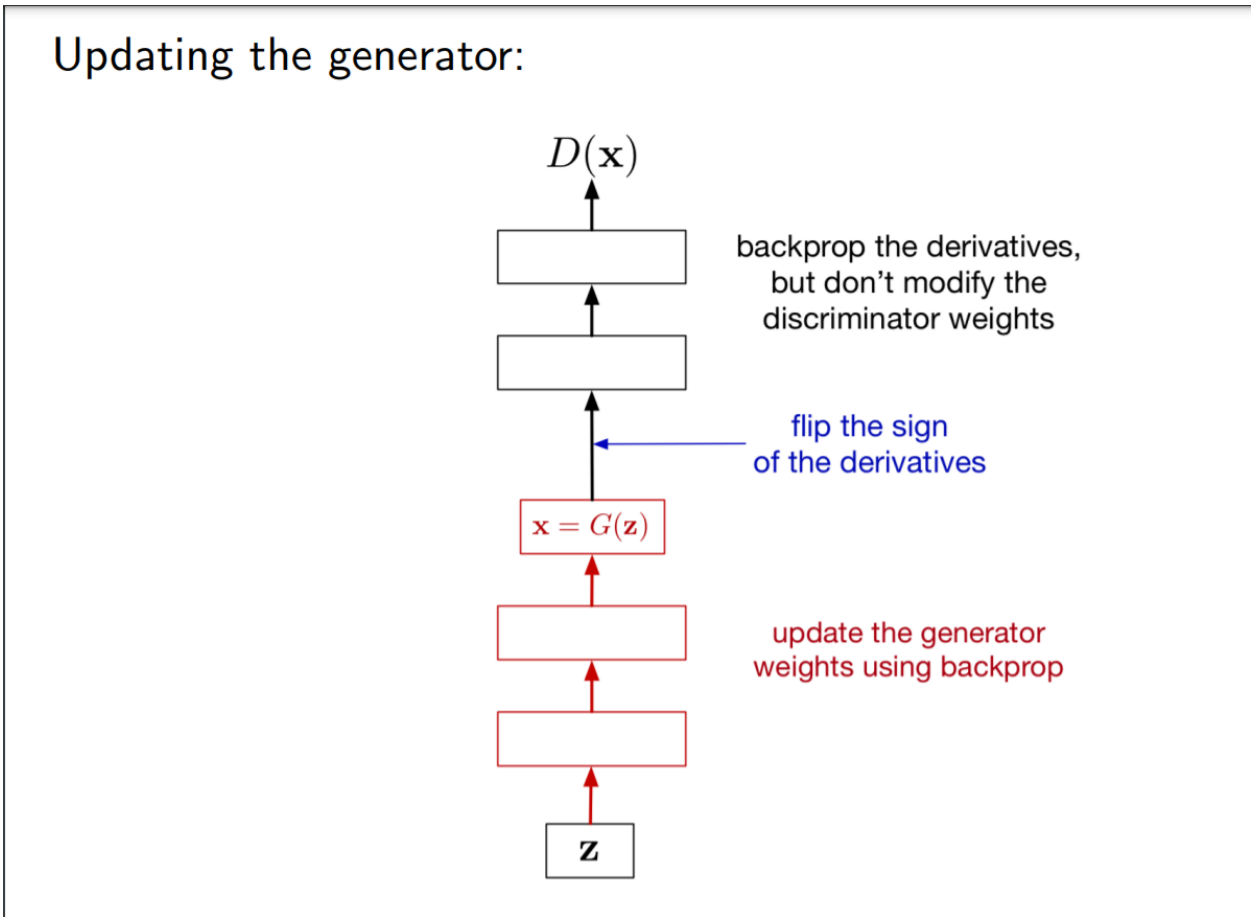
Thuật toán huấn luyện

Chúng ta sẽ huấn luyện, cập nhật các trọng số của bộ phân biệt (discriminator) bằng cách sử dụng lan truyền ngược (backpropagation) trên mục tiêu phân lớp

Updating the discriminator:



Trong khi đó, vì không có sự kết nối trực tiếp giữa dữ liệu và bộ sinh, bộ sinh không được huấn luyện từ dữ liệu nên chúng ta tận dụng bộ phân biệt để huấn luyện bộ sinh bằng cách sử dụng lan truyền ngược từ bộ phân biệt về bộ sinh:



Như chúng ta đã thấy từ công thức hàm mất mát của mô hình, $\min_G$ đứng trước $\max_D$. Vậy chúng ta sẽ cập nhật D ở vòng lặp bên trong và sau đó sẽ cập nhật G ở vòng lặp ngoài. Thuật toán được dùng sẽ như sau:

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

  for k steps **do**

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
  - Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for

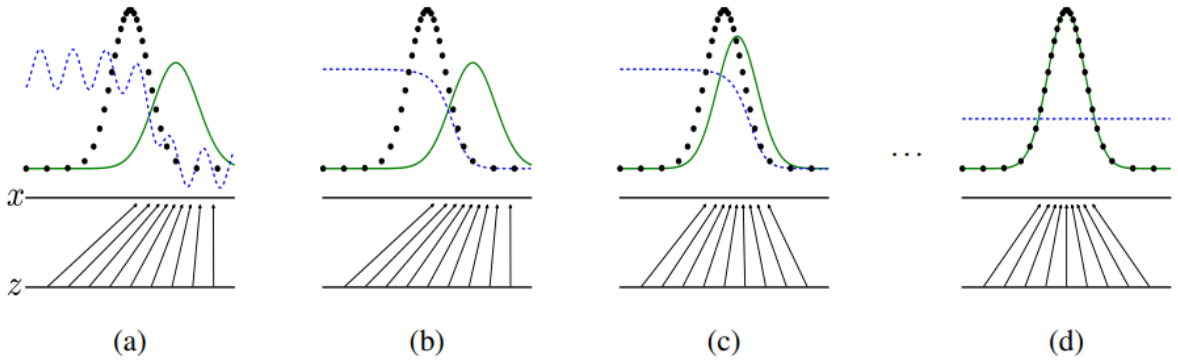
  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

- Ở vòng lặp trong, trong k bước, chúng ta sẽ lấy m mẫu noise từ phân phối (thường là phân phối chuẩn hoặc phân phối đều) làm đầu vào cho bộ sinh và m mẫu từ phân phối dữ liệu $p_{data}(\boldsymbol{x})$ rồi cập nhật bộ phân biệt bằng cách tăng *stochastic gradient.*
- Sau k vòng lặp cập nhật trọng số bộ phân biệt, chúng ta lấy m mẫu noise làm đầu vào và cập nhật bộ sinh bằng *stochastic gradient descent* theo công thức như trên hình.
    - Chúng ta có thể dùng momentum để tối ưu thuật toán



Ban đầu, chúng ta có z được lấy mẫu từ phân phối (thường là phân phối đều hoặc phân phối chuẩn) và được ánh xạ lên x thông qua bộ sinh G ($x = G(z)$). Chúng ta có G (dữ liệu được làm giả) được biểu diễn bởi đường liền màu xanh lá cây, trong khi dữ liệu từ tập dữ liệu - dữ liệu thật được biểu diễn bởi đường chấm đen và D được biểu diễn bởi đường gạch ngang xanh nước biển.

- Chúng ta sẽ huấn luyện (tối đa hóa bộ phân biệt) và nhận được D có dạng sigmoid như ở hình (b). Nó sẽ nói cho chúng ta về tỉ lệ của đường chấm đen và đường liền xanh ở mỗi điểm.
- Tiếp tục, chúng ta huấn luyện G theo D. Ta thấy đạo hàm của D sẽ theo hướng lên đồi vậy chúng ta sẽ huấn luyện G sao cho đường xanh liền sẽ đi lên theo hướng đạo hàm D. Điều quan trọng là G không biết trực tiếp dữ liệu thật mà chỉ biết thông qua D
- Sau nhiều lần cập nhật, chúng ta sẽ kết thúc với $D = \frac{1}{2}$ ở mọi nơi bởi vì đường G đã trùng với đường data ($p_G = p_{data}$)

Đảm bảo lý thuyết

Tối ưu toàn cục tại $p_g = p_{data}$

Ta có:

$$V(D,G) = \int_x p_{data}(\boldsymbol{x}) \log(D(\boldsymbol{x}))dx + \int_z p_{\boldsymbol{z}}(\boldsymbol{z}) \log(1 - D(g(\boldsymbol{z})))dz$$
$$= \int_x p_{data}(\boldsymbol{x}) \log(D(\boldsymbol{x}))dx + p_g(\boldsymbol{x}) \log(1 - D(x))dx$$

Do $\forall(a,b) \in R^2 \setminus \{0, 0\}$, hàm $y \to a\log(y) + b\log(1-y)$ đạt cực đại trong khoảng $[0,1]$ tại $\frac{a}{a+b}$.

Do đó, với G cố đinh, bộ phân biệt D tối ưu khi: $D_G^\star(\boldsymbol{x}) = \frac{p_{\text{data}}(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_g(\boldsymbol{x})}$

Hàm mục tiêu được viết lại theo $D_G^\star(\boldsymbol{x})$:

Note that the training objective for D can be interpreted as maximizing the log-likelihood for estimating the conditional probability $P(Y = y|\boldsymbol{x})$, where Y indicates whether $\boldsymbol{x}$ comes from p_{data} (with $y = 1$) or from p_g (with $y = 0$). The minimax game in Eq. 1 can now be reformulated as:

$$\begin{aligned} C(G) =& \max_D V(G,D) \\ =& \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}}[\log D_G^*(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}}[\log(1 - D_G^*(G(\boldsymbol{z})))] \\ =& \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}}[\log D_G^*(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{x}\sim p_g}[\log(1 - D_G^*(\boldsymbol{x}))] \\ =& \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}}\left[\log \frac{p_{\text{data}}(\boldsymbol{x})}{P_{\text{data}}(\boldsymbol{x}) + p_g(\boldsymbol{x})}\right] + \mathbb{E}_{\boldsymbol{x}\sim p_g}\left[\log \frac{p_g(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_g(\boldsymbol{x})}\right] \end{aligned} \tag{4}$$

Từ đó, họ chứng minh được điểm cực tiểu toàn cục mà $C(G)$ đạt được khi và chỉ khi $p_{\text{g}} = p_{\text{data}}$:

Theorem 1. *The global minimum of the virtual training criterion $C(G)$ is achieved if and only if $p_g = p_{data}$. At that point, $C(G)$ achieves the value $-\log 4$.*

Proof. For $p_g = p_{\text{data}}$, $D_G^*(\boldsymbol{x}) = \frac{1}{2}$, (consider Eq. 2). Hence, by inspecting Eq. 4 at $D_G^*(\boldsymbol{x}) = \frac{1}{2}$, we find $C(G) = \log \frac{1}{2} + \log \frac{1}{2} = -\log 4$. To see that this is the best possible value of $C(G)$, reached only for $p_g = p_{\text{data}}$, observe that

$$\mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}} [-\log 2] + \mathbb{E}_{\boldsymbol{x} \sim p_g} [-\log 2] = -\log 4$$

and that by subtracting this expression from $C(G) = V(D_G^*, G)$, we obtain:

$$C(G) = -\log(4) + KL\left(p_{\text{data}} \middle\| \frac{p_{\text{data}} + p_g}{2}\right) + KL\left(p_g \middle\| \frac{p_{\text{data}} + p_g}{2}\right) \tag{5}$$

where KL is the Kullback–Leibler divergence. We recognize in the previous expression the Jensen–Shannon divergence between the model's distribution and the data generating process:

$$C(G) = -\log(4) + 2 \cdot JSD\left(p_{\text{data}} \| p_g\right) \tag{6}$$

Since the Jensen–Shannon divergence between two distributions is always non-negative and zero only when they are equal, we have shown that $C^* = -\log(4)$ is the global minimum of $C(G)$ and that the only solution is $p_g = p_{\text{data}}$, i.e., the generative model perfectly replicating the data generating process. □

Sau đó, các tác giả đã chứng minh được thuật toán huấn luyện phía trên luôn tìm được nghiệm:

4.2 Convergence of Algorithm 1

Proposition 2. *If G and D have enough capacity, and at each step of Algorithm 1, the discriminator is allowed to reach its optimum given G, and p_g is updated so as to improve the criterion*

$$\mathbb{E}_{\boldsymbol{x}\sim p_{data}}[\log D^*_G(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{x}\sim p_g}[\log(1 - D^*_G(\boldsymbol{x}))]$$

then p_g converges to p_{data}

Proof. Consider $V(G, D) = U(p_g, D)$ as a function of p_g as done in the above criterion. Note that $U(p_g, D)$ is convex in p_g. The subderivatives of a supremum of convex functions include the derivative of the function at the point where the maximum is attained. In other words, if $f(x) = \sup_{\alpha\in\mathcal{A}} f_\alpha(x)$ and $f_\alpha(x)$ is convex in x for every α, then $\partial f_\beta(x) \in \partial f$ if $\beta = \arg\sup_{\alpha\in\mathcal{A}} f_\alpha(x)$. This is equivalent to computing a gradient descent update for p_g at the optimal D given the corresponding G. $\sup_D U(p_g, D)$ is convex in p_g with a unique global optima as proven in Thm 1, therefore with sufficiently small updates of p_g, p_g converges to p_x, concluding the proof. □

In practice, adversarial nets represent a limited family of p_g distributions via the function $G(\boldsymbol{z}; \theta_g)$, and we optimize θ_g rather than p_g itself. Using a multilayer perceptron to define G introduces multiple critical points in parameter space. However, the excellent performance of multilayer perceptrons in practice suggests that they are a reasonable model to use despite their lack of theoretical guarantees.

Điểm mạnh

Các lợi thế chủ yếu là về việc tính toán. Các mô hình đối nghịch cũng có thể đạt được một số lợi thế thống kê do mạng sinh không được cập nhật trực tiếp từ các mẫu dữ liệu, mà chỉ với lan truyền ngược từ bộ phân biệt. Điều này có nghĩa là các thành phần của đầu vào không được sao chép trực tiếp vào các tham số của bộ sinh.

Một ưu điểm khác của mạng đối nghịch là chúng có thể cho ra các phân phối sắc nét hơn các phương pháp dựa trên chuỗi Markov.

Ưu điểm chính là không bao giờ cần đến chuỗi Markov, chỉ sử dụng lan truyền ngược để tính đạo hàm, không cần suy luận trong quá trình học và có thể kết hợp nhiều chức năng vào mô hình.

Điểm yếu

Những nhược điểm chủ yếu là không có biểu diễn rõ ràng của $p_g(x)$ và D phải được đồng bộ hóa tốt với G trong quá trình đào tạo (đặc biệt, G không được huấn luyện quá nhiều mà không cập nhật D)

Cài đặt

GAN

Trong GAN, chúng ta huấn luyện đồng thời 2 mô hình:

Đầu tiên là mô hình phân biệt - Discriminator D:

```
class Discriminator(nn.Module):

    def __init__(self, input_size, hidden_dim, output_size):
        super(Discriminator, self).__init__()
        # define hidden linear layers
        self.fc1 = nn.Linear(input_size, hidden_dim*4)
        self.fc2 = nn.Linear(hidden_dim*4, hidden_dim*2)
        self.fc3 = nn.Linear(hidden_dim*2, hidden_dim)
        #final fully-connected layer
        self.fc4 = nn.Linear(hidden_dim, output_size)
        #dropout layer
        self.dropout = nn.Dropout(0.3)


    def forward(self, x):
        # flatten image
        x = x.view(-1, 28*28)

        # pass x through all layers
        # apply leaky relu activation to all hidden layers
        x = F.leaky_relu(self.fc1(x), 0.2)  #(input, negative_slope = 0.2)
        x = self.dropout(x)
        x = F.leaky_relu(self.fc2(x), 0.2)
        x = self.dropout(x)
        x = F.leaky_relu(self.fc3(x), 0.2)
        x = self.dropout(x)
        # final layer
        out = self.fc4(x)
        return out
```

Tác giả đã sử dụng mạng MLP để xây dựng bộ phân biệt. Các lớp được dùng là các lớp tuyến tính. Các lớp ẩn sẽ được áp dụng hàm kích hoạt Leaky ReLU ở đầu ra với mục đích khi dùng thuật toán lan truyền ngược để tính đạo hàm sẽ không bị cản trở. Do bộ phân biệt là 1 mô hình phân lớp (cho biết ảnh là thật hay giả) nên ở lớp cuối cùng chúng ta sẽ không áp dụng hàm kích hoạt nào; thay vào đó, chúng ta sẽ sử dụng kết hợp giữa hàm kích hoạt sigmoid và hàm loss Binary Cross Entropy.

Tiếp đến là mô hình sinh - Generator G:

```python
class Generator(nn.Module):

    def __init__(self, input_size, hidden_dim, output_size):
        super(Generator, self).__init__()

        # define hidden linear layers
        self.fc1 = nn.Linear(input_size, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim*2)
        self.fc3 = nn.Linear(hidden_dim*2, hidden_dim*4)

        # final fully-connected layer
        self.fc4 = nn.Linear(hidden_dim*4, output_size)

        # dropout layer
        self.dropout = nn.Dropout(0.3)

    def forward(self, x):
        # pass x through all layers
        x = F.leaky_relu(self.fc1(x), 0.2)
        x = self.dropout(x)
        x = F.leaky_relu(self.fc2(x), 0.2)
        x = self.dropout(x)
        x = F.leaky_relu(self.fc3(x), 0.2)
        x = self.dropout(x)

        # final layer should have tanh applied
        out = F.tanh(self.fc4(x))

        return out
```

Cũng giống như bộ phân biệt, ở bộ sinh, chúng ta cũng xây dựng mô hình bằng mạng MLP. Tuy nhiên, ở lớp đầu ra cuối cùng, chúng ta sẽ áp dụng hàm kích hoạt tanh(). Một lưu ý nhỏ là hàm tanh() ở bộ sinh sẽ cho đầu ra trong khoảng [-1, 1], trong khi đó đầu ra của bộ phân biệt nằm trong khoảng [0, 1] nên chúng ta sẽ phải "scale" lại ảnh đầu vào về khoảng [-1, 1] khi huấn luyện bộ phân biệt.

Sau khi xây dựng xong bộ sinh và bộ phân biệt, chúng ta cần xây dựng hàm mất mát cho mô hình.

```
# Calculate losses
def real_loss(D_out, smooth=False):
    batch_size = D_out.size(0)

    # smooth labels if smooth=True
    if smooth:
        labels = torch.ones(batch_size) * 0.9
    else:
        labels = torch.ones(batch_size)

    # numerically stable loss
    criterion = nn.BCEWithLogitsLoss()
    # calculate loss
    loss = criterion(D_out.squeeze(), labels)
    return loss

def fake_loss(D_out):
    batch_size = D_out.size(0)
    labels = torch.zeros(batch_size)
    criterion = nn.BCEWithLogitsLoss()
    # compare logits to fake labels
    loss = criterion(D_out.squeeze(), labels)
    return loss
```

Hàm real_loss() sẽ tính loss đối với ảnh thật và hàm fake_loss() sẽ tính loss đối với ảnh giả.

Đối với bộ phân biệt D, hàm mất mát của D sẽ là tổng mất mát đối với ảnh thật và ảnh giả. Chúng ta sẽ muốn D phân các ảnh thật với nhãn 1 và ảnh giả được sinh từ G với nhãn 0. Để D hoạt động tốt hơn, tác giả đã giảm nhãn ảnh thật từ 1 xuống 0.9.

Còn với bộ sinh G, nó sẽ cố khiến cho D gán nhãn ảnh do nó sinh ra là 1 (ảnh thật).

Bên cạnh đó, chúng ta cũng có thuật toán tối ưu hóa Adam với hệ số học (learning rate) lr = 0.002 được áp dùng cho cả D và G

```
# learning rate for optimizers
lr = 0.002

# Create optimizers for the discriminator and generator
d_optimizer = optim.Adam(D.parameters(), lr)
g_optimizer = optim.Adam(G.parameters(), lr)
```

Và cuối cùng, chúng ta sẽ huấn luyện mô hình.

```
# training hyperparams
num_epochs = 100

# keep track of loss and generated, "fake" samples
samples = []
losses = []

print_every = 400

# Get some fixed data for sampling. These are images that are held
# constant throughout training, and allow us to inspect the model's performance
sample_size=16
fixed_z = np.random.uniform(-1, 1, size=(sample_size, z_size))
fixed_z = torch.from_numpy(fixed_z).float()

# train the network
D.train()
G.train()
for epoch in range(num_epochs):

    for batch_i, (real_images, _) in enumerate(train_loader):

        batch_size = real_images.size(0)

        ## Important rescaling step ##
        real_images = real_images*2 - 1  # rescale input images from [0,1) to [-1, 1)

        # =============================================
        #            TRAIN THE DISCRIMINATOR
        # =============================================
```

```
# =========================================
#            TRAIN THE DISCRIMINATOR
# =========================================

d_optimizer.zero_grad()

# 1. Train with real images

# Compute the discriminator losses on real images
# use smoothed labels
D_real = D(real_images)
d_real_loss = real_loss(D_real, smooth=True)

# 2. Train with fake images

# Generate fake images
with torch.no_grad():
    z = np.random.uniform(-1, 1, size=(batch_size, z_size))
    z = torch.from_numpy(z).float()
    fake_images = G(z)

# Compute the discriminator losses on fake images
D_fake = D(fake_images)
d_fake_loss = fake_loss(D_fake)

# add up real and fake losses and perform backprop
d_loss = d_real_loss + d_fake_loss
d_loss.backward()
d_optimizer.step()
```

```
# =========================================
#            TRAIN THE GENERATOR
# =========================================

g_optimizer.zero_grad()

# 1. Train with fake images and flipped labels

# Generate fake images
z = np.random.uniform(-1, 1, size=(batch_size, z_size))
z = torch.from_numpy(z).float()
fake_images = G(z)

# Compute the discriminator losses on fake images
# using flipped labels!
D_fake = D(fake_images)
g_loss = real_loss(D_fake)

# perform backprop
g_loss.backward()
g_optimizer.step()


# Print some loss stats
if batch_i % print_every == 0:
    # print discriminator and generator loss
    print('Epoch [{:5d}/{:5d}] | d_loss: {:6.4f} | g_loss: {:6.4f}'.format(
            epoch+1, num_epochs, d_loss.item(), g_loss.item()))
```

Chúng ta sẽ huấn luyện D và G thay phiên nhau như đã nêu trên thuật toán huấn luyện.

Huấn luyện D:

- Đầu tiên, chúng ta sẽ tính loss của D trên ảnh thật từ tập dữ liệu
- Sau đó, chúng ta sẽ lấy noise z từ phân phối đều, sau đó đưa qua bộ sinh G để sinh ra ảnh giả.
- Từ ảnh giả được sinh, chúng ta tính loss của D trên ảnh giả rồi cộng với loss của D trên ảnh thật để được loss D

- Cuối cùng, chúng ta sử dụng thuật toán lan truyền ngược và áp dụng thuật toán tối ưu để cập nhật trọng số của bộ phân biệt D

Huấn luyện G:

- Sau khi huấn luyện D, chúng ta sẽ huấn luyện G. Chúng ta cũng sẽ lấy noise z từ phân phối đều rồi đưa qua bộ sinh G để sinh ảnh giả.
- Sau đó, vì muốn D nhầm ảnh giả thành ảnh thật, ta sẽ cho đống ảnh giả đó qua D rồi tính real_loss như ảnh thật.
- Và cuối cùng vẫn là sử dụng lan truyền ngược và thuật toán tối ưu để cập nhật trọng số bộ sinh G.

Và đó là quá trình xây dựng GAN.

DCGAN

DCGAN là một mô hình được mở rộng từ GAN. Trong DCGAN, chúng ta vẫn sẽ huấn luyện 2 mô hình đồng thời: Mô hình sinh/ Bộ sinh G và mô hình phân biệt/ Bộ phân biệt D. Tuy nhiên, thay vì D và G được xây dựng bằng mạng neural network thông thường, 2 mô hình này sẽ được xây dựng bằng mạng CNN (mạng nơ-ron tích chập) với các lớp tích chập (convolutional layer) và các lớp tích chập chuyển vị (transposed convolutional layer).

(Mô hình này em tham khảo từ Tutorial của Pytorch, từ Tutorial trong link hướng dẫn và từ 1 blog tiếng Việt)

Đầu tiên, chúng ta xây dựng mô hình phân biệt D:

```python
class Discriminator(nn.Module):

    def __init__(self, conv_dim=32):
        super(Discriminator, self).__init__()

        # complete init function
        self.conv_dim = conv_dim

        # transpose conv layers
        self.conv1 = nn.Sequential(
            nn.Conv2d(1, conv_dim, kernel_size=4, stride=2, padding=1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(conv_dim, conv_dim * 2, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(conv_dim * 2),
            nn.LeakyReLU(0.2, inplace=True),
        )
        self.conv3 = nn.Sequential(
            nn.Conv2d(conv_dim * 2, conv_dim * 4, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(conv_dim * 4),
            nn.LeakyReLU(0.2, inplace=True),
        )
        self.fc = nn.Linear(conv_dim*4*4*4, 1)

    def forward(self, x):
        output = self.conv1(x)
        output = self.conv2(output)
        output = self.conv3(output)
        output = output.view(-1, self.conv_dim*4*4*4)
        output = self.fc(output)
        return output
```

Chúng ta sẽ sử dụng mạng nơ-ron tích chập để xây dựng bộ phân biệt. Các lớp được dùng là các lớp tích chập (Conv2d). Sau đó, chúng ta sẽ áp dụng "batch normalization" bằng hàm BatchNorm2d trên mỗi lớp trừ lớp tích chập đầu tiên và lớp tuyến tính cuối cùng. Các lớp ẩn sẽ được áp dụng hàm kích hoạt Leaky ReLU ở đầu ra. Và cũng như bộ phân biệt ở GAN, chúng ta cũng sẽ sử dụng kết hợp giữa hàm kích hoạt sigmoid và hàm loss Binary Cross Entropy.

Ảnh đầu vào sẽ được đi qua các lớp tích chập với stride = 2 để giảm dần kích thước. Sau đó tensor sẽ được đưa về dạng véc-tơ và dùng 1 lớp tuyến tính để đưa về dạng 1d.

Tiếp đến chúng ta xây dựng mô hình sinh - G:

```python
class Generator(nn.Module):

    def __init__(self, z_size, conv_dim=32):
        super(Generator, self).__init__()


        self.conv_dim = conv_dim

        self.fc = nn.Linear(z_size, conv_dim*4*4*4)

        self.t_conv1 = nn.Sequential(
            nn.ConvTranspose2d(conv_dim * 4, conv_dim * 2, kernel_size=3, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(conv_dim * 2),
            nn.ReLU(True)
        )
        self.t_conv2 = nn.Sequential(
            nn.ConvTranspose2d(conv_dim * 2, conv_dim , kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(conv_dim),
            nn.ReLU(True)
        )
        self.t_conv3 = nn.Sequential(
            nn.ConvTranspose2d(conv_dim, 1 , kernel_size=4, stride=2, padding=1, bias=False),
            nn.Tanh()
        )


    def forward(self, x):
        out = self.fc(x)
        out = out.view(-1, self.conv_dim*4, 4, 4) # (batch_size, depth, 4, 4)
        out = self.t_conv1(out)
        out = self.t_conv2(out)
        out = self.t_conv3(out)
        return out
```

Đầu tiên, input noise được đưa qua lớp fully-connected sau đó được reshape về dạng tensor 3d. Sau đó, chúng ta dùng lớp Tranposed Convolution đi cùng với "batch normalization" và hàm kích hoạt ReLU ở các lớp ẩn tiếp theo. Và ở lớp cuối cùng, chúng ta chỉ áp dụng hàm kích hoạt Tanh() sau khi cho qua lớp Transposed Convolution.

Việc tính loss và huấn luyện mô hình ở DCGAN cũng tương tự như ở GAN đã nêu bên trên.

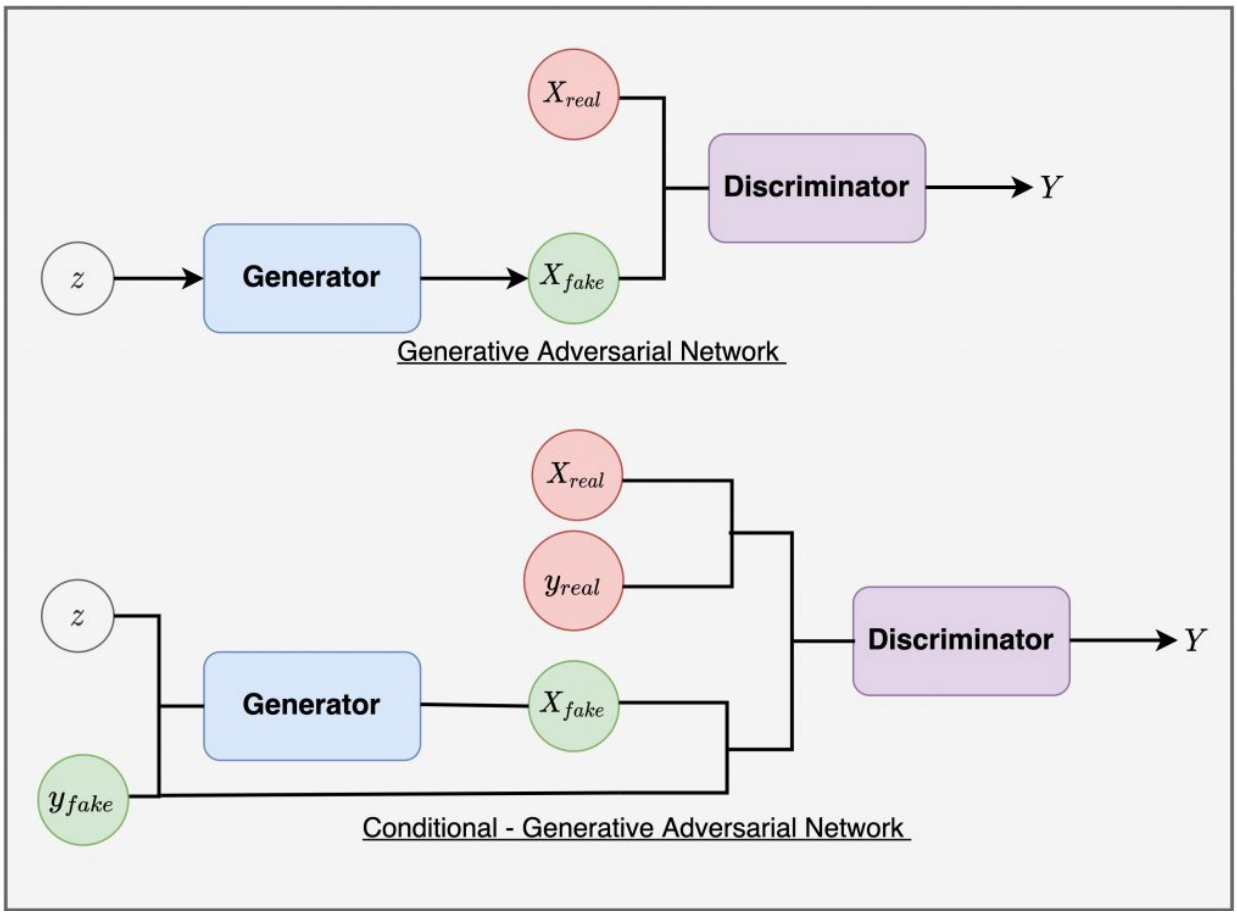
CGAN

Mô hình này em tham khảo ở bài báo về mô hình và 1 số bài viết trên mạng.

Ở GAN và DCGAN, sau khi huấn luyện xong mô hình, ta có thể dùng bộ sinh G để sinh ra ảnh mới giống ảnh trong tập dữ liệu, tuy nhiên, chúng ta không thể kiểm soát được

ảnh sinh ra giống loại nào trong tập dữ liệu. Mô hình CGAN (Conditional GAN) có thể khắc phục được nhược điểm đó.

Cấu trúc mạng ở CGAN cũng có 1 số thay đổi nhỏ.



Ở GAN và DCGAN, bộ sinh G sinh ra ảnh giả từ noise vector. Còn ở CGAN, bên cạnh noise vector, chúng ta sẽ thêm vào nhãn y với mong muốn sinh ra ảnh giả có nhãn y đó.

Cũng tương tự với bộ sinh G, ở bộ phân biệt D, bên cạnh ảnh đầu vào, chúng ta cũng có nhãn y đi kèm. Từ đó dẫn đến hàm loss của mô hình cũng có thay đổi (trở thành loss với điều kiện đã biết y):

The objective function of a two-player minimax game would be as Eq 2

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x}|\boldsymbol{y})] + \mathbb{E}_{\boldsymbol{z}\sim p_z(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z}|\boldsymbol{y})))]. \qquad (2)$$

Về phần cài đặt, em có tham khảo cách xây dựng mạng D và G tại đây

Đầu tiên là xây dựng bộ sinh G:

```
class Generator(nn.Module):
    def __init__(self):
        super().__init__()

        self.label_emb = nn.Embedding(10, 10)

        self.model = nn.Sequential(
            nn.Linear(110, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(256, 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 1024),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(1024, 784),
            nn.Tanh()
        )

    def forward(self, z, labels):
        z = z.view(z.size(0), 100)
        c = self.label_emb(labels)
        x = torch.cat([z, c], 1)
        out = self.model(x)
        return out.view(x.size(0), 28, 28)
```

Nhãn đầu vào được đi qua Embedding layer sau đó được xếp chồng lên z là noise vector. Sau đó chúng được đi qua mô hình với các lớp ẩn là lớp tuyến tính được áp dụng hàm kích hoạt LeakyReLU ở đầu ra trừ lớp cuối cùng. Ở lớp cuối, chúng ta áp dụng hàm Tanh() như bộ sinh G ở GAN và DCGAN. Cuối cùng, G cho ra ảnh 1*28*28.

Tiếp đến là xây dựng D:

```python
class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.label_emb = nn.Embedding(10, 10)

        self.model = nn.Sequential(
            nn.Linear(794, 1024),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),
            nn.Linear(1024, 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

    def forward(self, x, labels):
        x = x.view(x.size(0), 784)
        c = self.label_emb(labels)
        x = torch.cat([x, c], 1)
        out = self.model(x)
        return out.squeeze()
```

Nhãn đầu vào ở D cũng giống như ở G, nó được đi qua Embedding layer rồi được xếp chồng lên ảnh đầu vào đã được reshape. Sau đó chúng được đi qua các lớp ẩn là các lớp tuyến tính cũng được áp dụng hàm kích hoạt LeakyReLU và có thêm cả Dropout layer. Ở lớp đầu ra cuối cùng, sau khi về dạng 1d, đầu ra sẽ được áp dụng hàm Sigmoid().

Việc huấn luyện CGAN cũng khá giống với GAN và DCGAN.

```python
def generator_train_step(batch_size, discriminator, generator, g_optimizer, criterion):
    g_optimizer.zero_grad()

    Z = Variable(torch.randn(batch_size, 100)).cuda()
    fake_labels = Variable(torch.LongTensor(np.random.randint(0, 10, batch_size))).cuda()
    fake_images = generator(Z, fake_labels)

    validity = discriminator(fake_images, fake_labels)
    g_loss = criterion(validity, Variable(torch.ones(batch_size)).cuda())
    g_loss.backward()
    g_optimizer.step()
    return g_loss.item()
```

```python
def discriminator_train_step(batch_size, discriminator, generator, d_optimizer, criterion, real_images, labels):
    d_optimizer.zero_grad()

    #real loss
    real_validity = discriminator(real_images, labels)
    real_loss = criterion(real_validity, Variable(torch.ones(batch_size)).cuda())

    #fake loss
    Z = Variable(torch.randn(batch_size, 100)).cuda()
    fake_labels = Variable(torch.LongTensor(np.random.randint(0, 10, batch_size))).cuda()
    fake_images = generator(Z, fake_labels)
    fake_validity = discriminator(fake_images, fake_labels)
    fake_loss = criterion(fake_validity, Variable(torch.zeros(batch_size)).cuda())

    d_loss = real_loss + fake_loss
    d_loss.backward()
    d_optimizer.step()
    return d_loss.item()
```

Chúng ta vẫn có noise z được đưa vào G để sinh ảnh giả, sau đó ảnh giả được đưa qua D để phân biệt và tính được loss.

Ở D chúng ta cũng tính real loss và fake loss như ở mô hình GAN và DCGAN.

```python
def train(discriminator, generator, data_loader, num_epochs, lr, device = d2l.try_gpu()):
    #set parameter
    for w in discriminator.parameters():
        nn.init.normal_(w, 0, 0.01)
    for w in generator.parameters():
        nn.init.normal_(w, 0, 0.01)
    discriminator, generator = discriminator.to(device), generator.to(device)
    trainer_hp = {'lr': lr, 'betas': [0.9, 0.999]}
    # trainer_hp = {'lr': lr}
    d_optimizer = torch.optim.Adam(discriminator.parameters(), **trainer_hp)
    g_optimizer = torch.optim.Adam(generator.parameters(), **trainer_hp)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss', xlim=[1, num_epochs], legend=['discriminator', 'generator'])

    for epoch in range(1, num_epochs + 1):
        timer = d2l.Timer()
        metric = d2l.Accumulator(3)     #d_loss, g_loss, num_examples
        for i, (images, labels) in enumerate(data_loader):
            real_images = Variable(images).cuda()
            labels = Variable(labels).cuda()
            generator.train()
            d_loss = discriminator_train_step(len(real_images), discriminator, generator,
                                              d_optimizer, criterion, real_images, labels)
            g_loss = generator_train_step(batch_size, discriminator, generator, g_optimizer, criterion)
            metric.add(d_loss, g_loss, batch_size)
```

Ở đây, em có tham khảo thư viện của d2l để vẽ biểu đồ loss ngay trong lúc huấn luyện.

```python
def generate_digit(generator, digit):
    z = Variable(torch.randn(1, 100)).cuda()
    label = torch.LongTensor([digit]).cuda()
    img = generator(z, label).data.cpu()
    img = 0.5 * img + 0.5
    return transforms.ToPILImage()(img)

generate_digit(generator, 8)
```

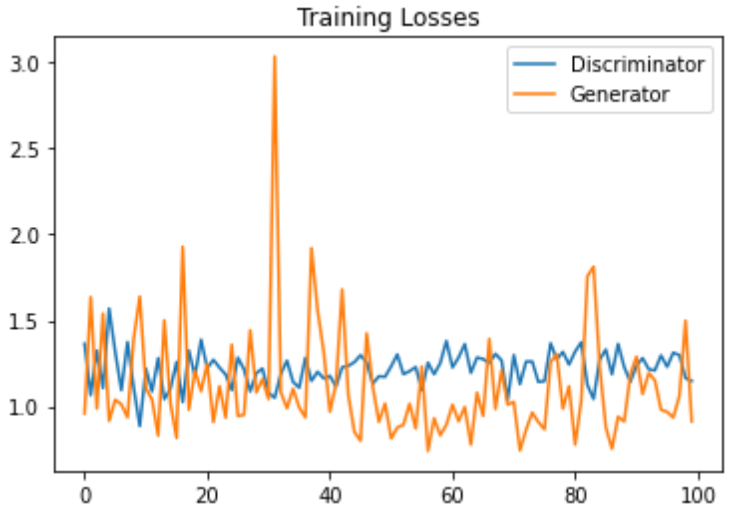
Cuối cùng, ta cũng có thể sinh ra ảnh giả với noise z và nhãn đầu vào mong muốn.

Pytorch Lightning

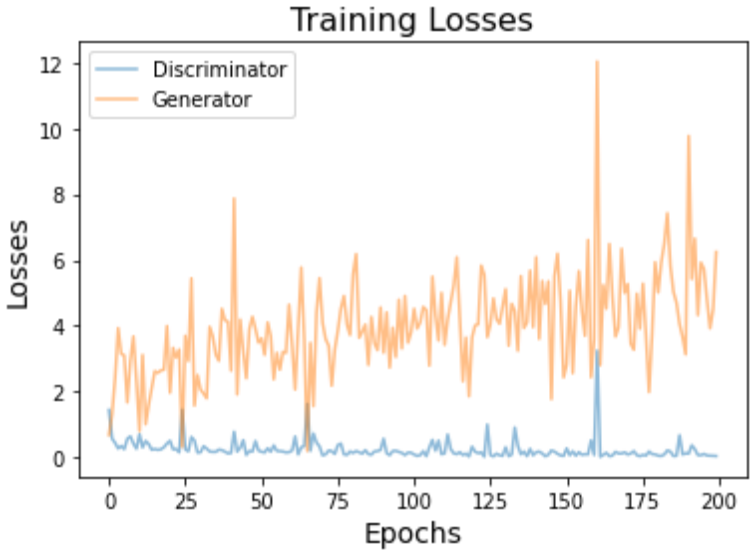
Phần này em tham khảo và chạy lại code tại Tutorial

Losses

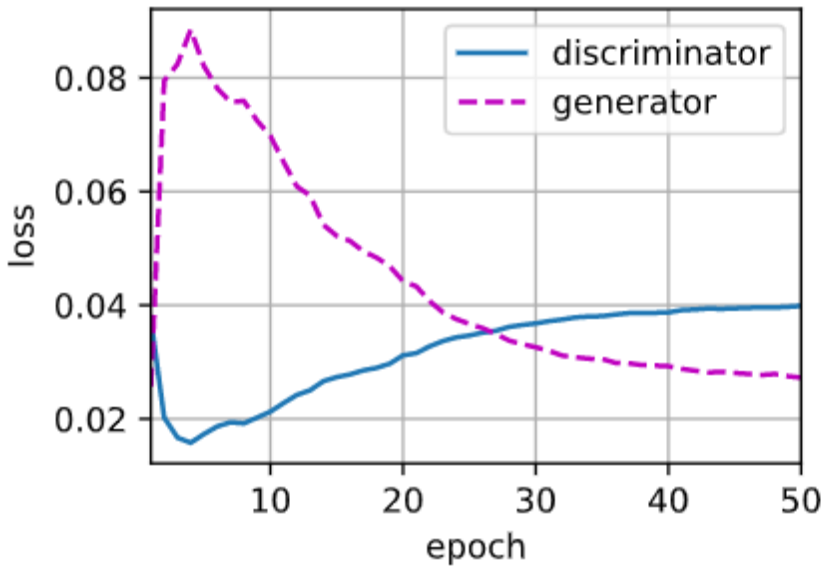
- Biểu đồ mô tả loss của D và G ở GAN:



- Biểu đồ mô tả loss của D và G ở DCGAN:

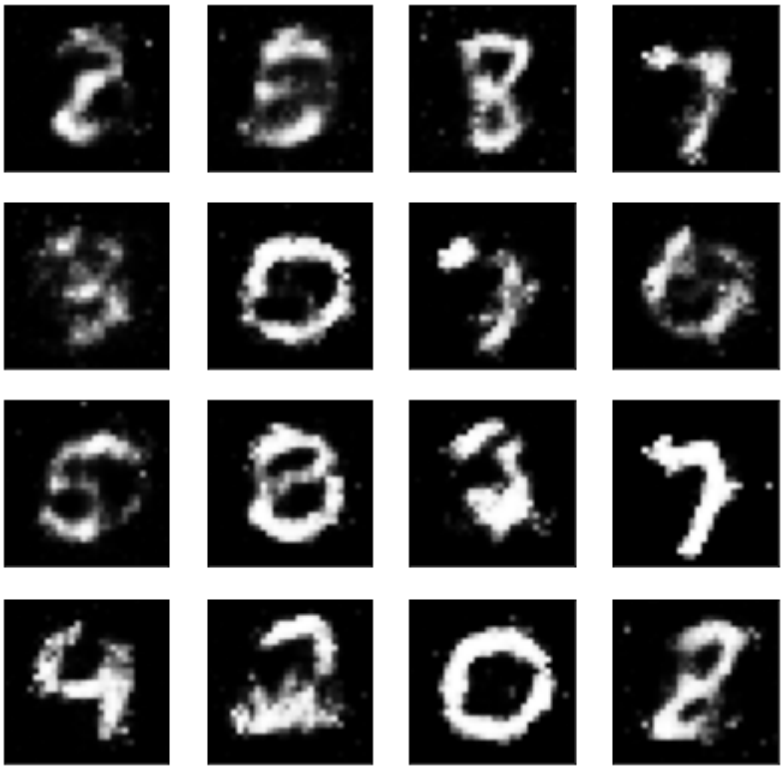


- Biểu đồ mô tả loss của D và G ở CGAN:

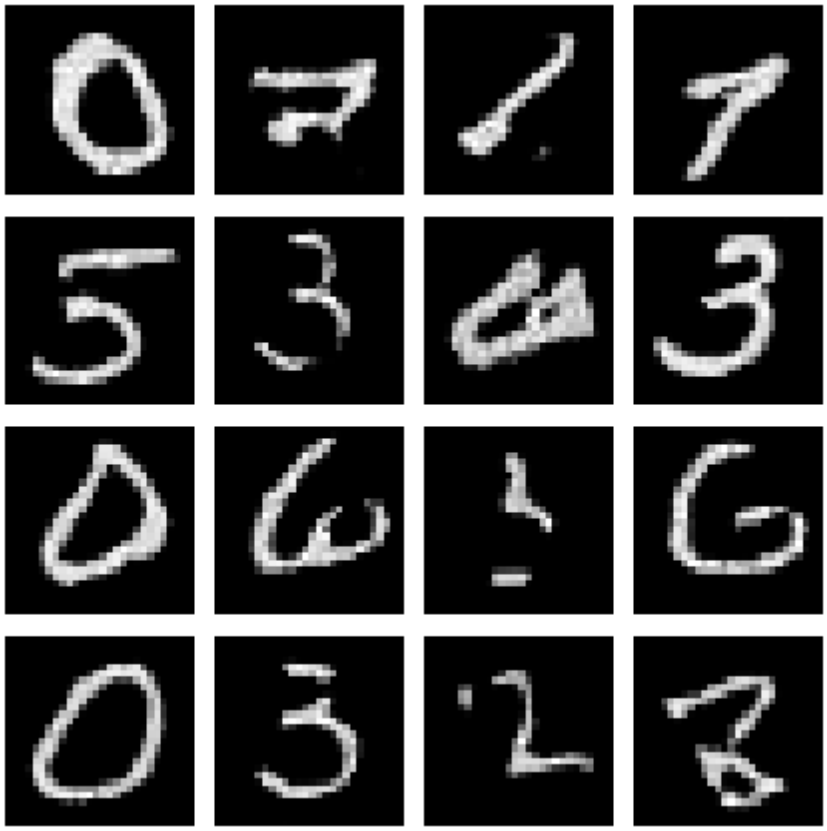


Kết quả

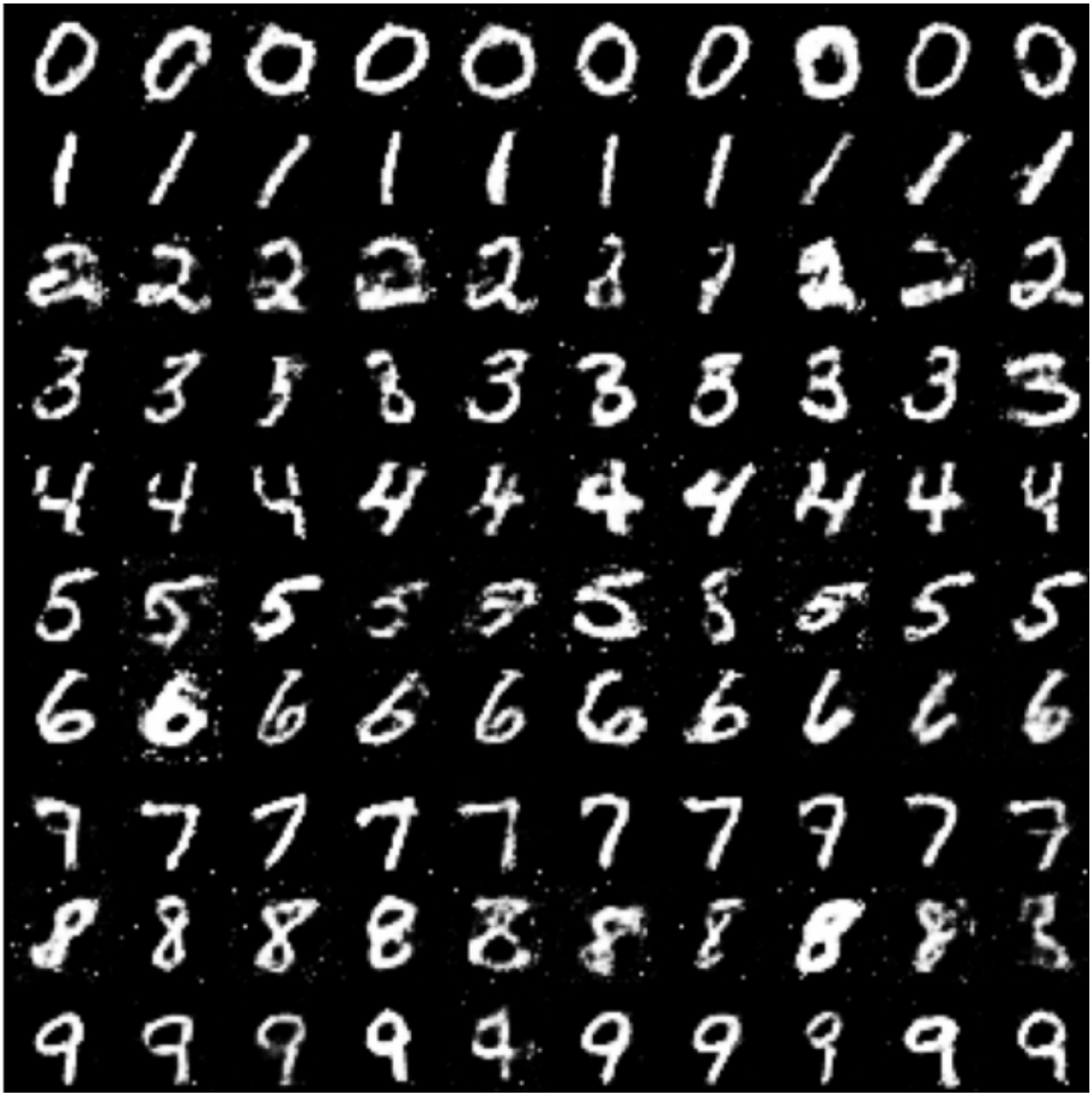
Kết quả khi sampling từ mô hình GAN sau khi được huấn luyện



Kết quả khi sampling từ mô hình DCGAN:



Kết quả khi sampling từ CGAN:



Kết luận

Qua quá trình đọc bài báo và triển khai mô hình GAN, em đã nhận được nhiều kinh nghiệm:

- Hiểu được về ý tưởng chung của các mô hình GAN, cách 1 mạng đối nghịch hoạt động và cách tính loss của mô hình.

- Cho dù mới đầu vẫn còn bỡ ngỡ nhưng qua các bài hướng dẫn và tự tìm hiểu, em đã triển khai được GAN, DCGAN và CGAN. Thông qua DCGAN, em cũng hiểu được cơ bản về cách xây dựng 1 mạng tích chập.
    - Về kết quả, ảnh được sinh ra bằng DCGAN có chất lượng tốt nhất mặc dù thời gian huấn luyện khá lâu. Có lẽ do các mạng nơ ron tích chập có khả năng trích xuất đặc trưng ảnh cao. Mạng CGAN cũng sẽ có chất lượng tương tự nếu D và G được triển khai bằng lớp tích chập. CGAN có ưu điểm là mình sẽ kiểm soát được mẫu mình muốn sinh ra bằng nhãn đầu vào cho mỗi mạng. Và mô hình GAN sinh ra ảnh có chất lượng kém nhất, do D và G được xây dựng từ mạng MLP.
    - Thông qua triển khai các mô hình, em cũng có thêm kiến thức về lớp tích chập và tích chập chuyển vị, các hàm kích hoạt được áp dụng vào mô hình cũng như cách reshape các đầu ra để phù hợp làm đầu vào ở các lớp kế tiếp.