

Usando GANs para geração automática de imagens de rostos humanos

Matheus O. Silva¹ e Rodrigo Pita¹

¹Instituto de Computação – Universidade Federal do Rio de Janeiro (UFRJ)
Rio de Janeiro – Brazil

matheusflups8@gmail.com , rodrigopita@dcc.ufrj.br



UFRJ
UNIVERSIDADE FEDERAL
DO RIO DE JANEIRO

Resumo. *Esse artigo tem como finalidade representar um resumo final de como nós desenvolvemos nosso código relacionado ao trabalho final da disciplina Introdução ao Aprendizado de Máquina. No nosso trabalho usamos redes generativas adversariais (GAN) conhecidas já por seu excelente desempenho em problemas com imagem, e nossa ideia inicial era justamente escolher um banco de dados qualquer com vários rostos e modelar uma rede neural para aprender como é um rosto humano e reproduzir imagens de novos rostos humanos.*

1. Introdução

A nossa motivação inicial para abordarmos esse tópico em específico foi porque eu sou muito interessado em sistemas de geração de imagem text-to-image como midjourney, dall-e e outros sistemas baseado em técnicas de aprendizado de máquina com finalidade de gerar imagens. Aproveitamos a experiência prévia do Rodrigo que já havia estudado um pouco sobre o assunto e já conhecia o funcionamento teórico de uma Rede Generativa Adversarial (GAN).

O objetivo no começo era conseguir gerar algo parecido com um rosto [2, 3, 4], porque não conhecíamos a complexidade do problema. Depois que vimos que seria completamente possível gerar imagens de rostos, nosso objetivo com esse trabalho foi apenas de pegar experiência de intuição no contexto de aprendizado de máquina envolvendo imagens. Aprender quantas épocas de treinamento devemos fazer e que tipo de resultados podemos esperar, quantas imagens deve-se ter no mínimo em um conjunto de dados para que consigamos resultados confiáveis, quais tipos de funções de ativação para cada camada de neurônios, diferenças de aplicar ou não convolução nas imagens e entre outras coisas.

2. Descrição da base

A base de dados veio do Kaggle [7] e contém 202.599 imagens de várias celebridades, em primeiro momento fizemos um arquivo preprocessor.py que redimensiona as imagens uma por uma e que está presente no nosso repositório do GitHub, onde disponibilizamos o trabalho completo apenas para registro de atividade. Porém, quando começamos a usar o PyTorch descobrimos que não seria necessário o arquivo de pré processamento, pois podemos aplicar um conjunto de transformações aos dados com uma função Resize e Normalize, que no caso é o que fazemos logo nas primeiras linhas do nosso notebook, o pré processamento acontece portanto logo no início do código nas células iniciais.

3. Métodos utilizados no trabalho

3.1. Como funcionam as GANs em teoria

Para explicarmos como nosso código funciona e como é composta a arquitetura da nossa rede neural, devemos primeiramente explicar como funcionam as GANs [1, 6] e

depois disso podemos de fato explicar a escolha das funções de ativação, otimizadores e outros parâmetros .

Na estrutura de redes adversárias proposta, o modelo generativo é confrontado com um adversário: um modelo discriminativo que aprende a determinar se uma amostra é da distribuição do modelo ou da distribuição de dados. O modelo generativo pode ser pensado como análogo a uma equipe de falsificadores, tentando produzir moeda falsa e usá-la sem detecção, enquanto o modelo discriminativo é análogo à polícia, tentando detectar a moeda falsa. A competição neste jogo leva ambas as equipes a melhorar seus métodos até que as falsificações sejam indistinguíveis dos dados genuínos.

No artigo original das GANs [1], proposto por Ian Goodfellow, explorou-se o caso especial em que o modelo generativo gera amostras passando ruído aleatório (ruído que vamos chamar futuramente de amostra do espaço latente) por um perceptron multicamadas que tem como output algum item que queremos gerar. O modelo discriminativo também é um perceptron multicamadas cujo retorno é um escalar para a probabilidade de dado item ser da base real, quanto menor essa probabilidade, mais provável que tenha vindo do gerador. Nesse caso, podemos treinar ambos os modelos usando apenas os algoritmos altamente bem-sucedidos de backpropagation e dropout.

Vimos também no artigo [1], que a estrutura de modelagem adversária é mais simples de aplicar quando os modelos são perceptrons multicamadas. Para aprender a distribuição do gerador p_g sobre os dados x , definimos um prior nas variáveis de ruído de entrada $p_z(z)$ no começo de cada loop de treinamento de cada batch (lote), e isso é feito no código gerando um tensor de tamanho 100 (arbitrário) que vai conter números aleatórios e vai servir de entrada para nossa rede generativa. E então representamos um mapeamento para o espaço de dados como $G(z; \theta_g)$ onde G é uma equação diferenciável, representada pelo nosso perceptron multicamadas com parâmetros θ_g que no código representamos pela classe Generator. Também definimos o outro perceptron multicamadas $D(x; \theta_d)$, representado no código pela classe Discriminator, que vai se modificando ao longo do treinamento. Lembrando que x é o item gerado pela rede generativa, o retorno desse perceptron é um escalar representando a probabilidade de x ter vindo dos dados em vez de p_g (distribuição do gerador).

3.2. Matemática prática por trás das GANs

Então treinamos D para maximizar a probabilidade de atribuir o rótulo correto aos exemplos de treinamento e amostras de G , ou seja, queremos que ele venha a ser enganado, e isso implica que estamos treinando G para melhorar cada vez mais, e minimizar a parte da equação da entropia cruzada $\log(1 - D(G(z)))$.¹

Em outras palavras nossas GANs estão jogando o seguinte jogo minimax com a função $V(D, G)$:

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

Para explicar melhor a nossa função de perda:

A parte $\min_G \max_D$ indica que estamos realizando uma otimização em duas etapas.

Primeiro, maximizamos em relação aos parâmetros da rede discriminativa (D) e, em seguida, minimizamos em relação aos parâmetros da rede generativa (G).

A parte $E_{x \sim p_{data}(x)}[\log D(x)]$ calcula a esperança dos valores logarítmicos da saída da rede discriminativa (D) para amostras reais x , onde x é amostrado da distribuição de dados reais $p_{data}(x)$. Em outras palavras, mede quão bem a rede discriminativa consegue distinguir as amostras reais.

A parte $E_{z \sim p_z(z)}[\log(1 - D(G(z)))]$ calcula a esperança dos valores logarítmicos da saída da rede discriminativa (D) para amostras geradas pela rede generativa (G), onde z é amostrado da distribuição de ruído $p_z(z)$. Então, em outras palavras, essa parte mede quão bem a rede discriminativa consegue distinguir as amostras geradas daquelas provenientes da distribuição real.

Durante o treinamento, as duas redes são atualizadas em direções opostas para encontrar um equilíbrio, resultando em uma rede generativa capaz de gerar amostras realistas e uma rede discriminativa capaz de distinguir corretamente as amostras.

E enquanto ao explicar tudo acima, descrevemos nosso algoritmo de treinamento, também chamado de loop de treinamento, que está no nosso código. Fazemos isso para todos os lotes de treinamento e para cada época, criando assim um processo de treino robusto.

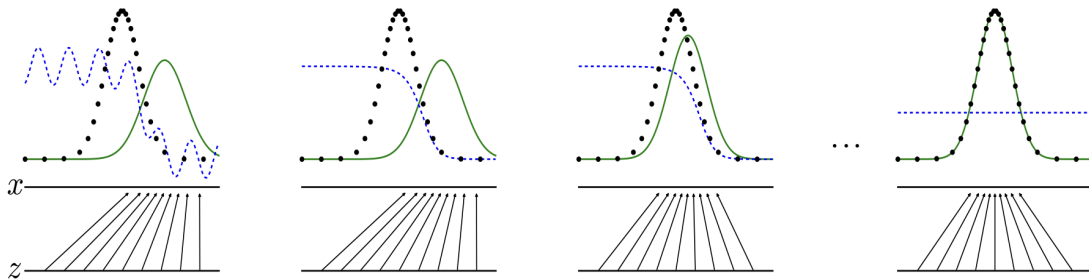


Figura 1. Essa figura acima mostra que as redes adversárias generativas são treinadas atualizando simultaneamente a distribuição discriminativa (D, azul, linha tracejada) para que ela discrimine entre amostras da distribuição geradora de dados (linha preta, pontilhada) p_x daquelas da distribuição generativa p_g (G) (verde, linha sólida). A linha horizontal inferior é o domínio do qual z é amostrado, neste caso uniformemente. As setas para cima mostram como o mapeamento $x = G(z)$ impõe a distribuição não uniforme p_g nas amostras transformadas e como as coisas melhoram conforme se passam as épocas de treinamento. Fonte: artigo original das GANs.

4. Arquitetura das redes adversárias

4.1.1. Evolução da rede discriminativa sem convolução

Todas as referências básicas sobre como desenvolver os modelos discriminativos falam sobre criar uma camada linear de entrada de tamanho $3 \times \text{dimensão} \times \text{dimensão}$ em casos de imagens coloridas, ou $\text{dimensão} \times \text{dimensão}$ em caso de imagens cinzas. Isso se dá pelo conceito de como as cores das imagens são tratadas no computador, imagens cinzas costumam ser composta por apenas 1 canal de coloração. Já para imagens coloridas usamos o RGB que possui 3 canais de coloração, vermelho, verde e azul. Outra convenção na construção de modelos de aprendizado de máquina que vimos é utilizar camadas com potências de dois como o tamanho. Por conta disso, nossa segunda camada tem tamanho 1024. Para a função de ativação entre as camadas usamos ReLU (Rectified Linear Unit). Incluímos também dropout de 30% entre todas as camadas, vimos que é uma técnica amplamente utilizada que desativa aleatoriamente algumas unidades durante o treinamento, o que ajuda a evitar o overfitting. Em geral as funções de ativação são as mesmas entre todas as camadas e em todas aplicamos dropout como dito anteriormente, mas no final há uma quebra de lógica, porque as camadas vão de 1024 para 512, de 512 para 256, e de 256 para 128 de novo que é a dimensão da imagem, mas ao invés de fazer isso construindo uma camada linear como as outras, aplicamos a função de ativação sigmoid para retornar um número do intervalo [0,1], que nos diz exatamente a probabilidade que queríamos desde o começo como resposta do discriminador.

Bem, mas após uma leitura mais aprofundada descobrimos que poderia melhorar a arquitetura em um aspecto, mudar a função de ativação ReLU de todas as camadas por LeakyReLU (Rectified Linear Unit com vazamento), diferente da função ReLU que recebe um x e retorna x se ele for positivo e 0 se ele for negativo, essa função recebe um x e retorna x se ele for positivo e px se ele for negativo, onde p é um parâmetro da função de ativação chamado leakage rating, que determina quanto a função irá vaziar quando o input é negativo. Um valor muito comum de p era 0.2, que foi o que usamos, e estava nos nossos planos testar outros valores de p mas como é muito demorado o tempo de treinamento com imagens e como a melhoria esperada por nós ajustando esse p seria mínima, só não fizemos esses testes, mas deixamos como registro aqui que é algo a se fazer quando se busca perfeição em trabalhos com imagem. A vantagem da função LeakyReLU é que ela ajuda a mitigar o chamado "problema de neurônios mortos" (dead neurons problem), que ocorre quando a função ReLU deixa de ativar as unidades em uma rede neural devido a um valor negativo de entrada. O vazamento introduzido pela Leaky ReLU permite que a informação flua mesmo quando a entrada é negativa, evitando a inativação completa dos neurônios. O que era um problema que seria muito grave porque além de ReLU, usávamos dropout junto, e aí poderíamos com a morte excessiva de neurônios gerar imagens consistentes e ruidosas, com repetição de pixels pretos (valor de preto são os 3 canais 0) onde não deveriam.

4.1.2. Aplicando convolução a rede discriminativa

Após achar outros códigos-exemplo decidimos que o impulso final para o melhor desempenho do nosso programa era usar camadas convolucionais, as camadas de

convolução são eficazes na extração de características espaciais das imagens. Elas são capazes de aprender padrões locais, como bordas, texturas e formas, através da aplicação de filtros em diferentes regiões da imagem. Outro ponto importante é a redução da dimensionalidade, as camadas de convolução têm a capacidade de reduzir a dimensionalidade dos dados de entrada. À medida que o kernel desliza sobre a imagem com um stride maior que 1, a saída da convolução é reduzida em tamanho. Essa redução da dimensionalidade ajuda a comprimir as informações relevantes da imagem em uma amostra do espaço latente mais compacto, o que facilita a aprendizagem e o treinamento da rede. Outra coisa boa é quanto a invariância a translações, as camadas de convolução são invariantes a translações, o que significa que elas podem identificar as mesmas características independentemente da sua localização na imagem. Essa propriedade é especialmente útil em GANs, onde o objetivo é gerar imagens realistas que sejam indistinguíveis das imagens reais. A invariância a translações permite que a rede aprenda características globais e representações robustas, independentemente da posição dos objetos nas imagens. Outra coisa importante que convolução nos dá é o aprendizado hierárquico, as camadas de convolução permitem o aprendizado hierárquico de características. À medida que a informação flui através de várias camadas convolucionais, a rede é capaz de aprender características cada vez mais complexas e abstratas. Isso é importante para a geração de imagens de alta qualidade, pois a rede pode aprender a representar detalhes finos e estruturas de alto nível, inclusive uma coisa que não foi feita nesse trabalho e vai ser incluída em trabalhos futuros é o uso de mais camadas, porque mais camadas (obviamente não criando exageros) tendem a aumentar o aprendizado de características mais abstratas e vão gerar resultados certamente melhores que os atuais, mas isso também implica em um treinamento mais demorado, o que acabou nos impedindo de fazer essa melhoria nesse trabalho.

Dado todos esses ganhos de qualidade ao nosso modelo, não pensamos duas vezes, e fomos implementar convolução ao nosso modelo discriminativo. Basicamente trocamos as camadas lineares por camadas de convolução 2D, que recebem como parâmetros o número de canais de entrada, número de canais de saída, dimensão do kernel, stride e padding, onde kernel é uma matriz de pesos que desliza sobre a imagem de entrada durante a operação de convolução, stride define a quantidade de pixels pelo qual o kernel se move a cada passo durante a convolução e o padding refere-se à adição de zeros ao redor da imagem de entrada antes da operação de convolução. Ele é usado para controlar o tamanho da saída da convolução e preservar as informações nas bordas da imagem. Cada uma dessas camadas se conectam com a próxima por meio de uma função de ativação LeakyReLU e passam também por uma camada de normalização por batch 2D ela normaliza os valores de saída da camada anterior para ajudar na estabilização e acelerar o treinamento, no PyTorch essa funcionalidade se chama BatchNorm2d. E novamente nós repetimos essas sequências de camadas 4 vezes até que chegamos na função sigmoid no final que vai nos retornar um escalar, assim como previa na nossa definição de D lá no começo.

Então essa foi a nossa arquitetura final da rede discriminativa, escolhemos convolução por conta dos diversos benefícios supracitados, LeakyRelu porque mitiga o problema dos neurônios mortos, e output passa por uma ativação sigmoid porque nosso objetivo é

uma probabilidade, e no meio é feito normalizações para estabilização e acelerar o treinamento.

4.2.1. Evolução da rede generativa sem convolução

De forma análoga ao processo de criação da nossa rede discriminativa, primeiro testamos a combinação mais segura e ingênua que poderíamos, segura no sentido de que já nos daria algum resultado para começarmos a ter uma intuição de como seria nosso output das imagens. E essa primeira arquitetura que criamos foi basicamente 4 camadas, que recebíamos uma entrada de tamanho 100 e transformávamos para 256, depois 512, depois 1024, e depois enfim o output de tamanho $3 * \text{dimensão} * \text{dimensão}$. Essa entrada de tamanho 100 é arbitrária e é a nossa amostra do espaço latente z gerado aleatoriamente só para termos uma distribuição qualquer que vai tentar ser levada a distribuição das nossas imagens reais. Todas essas camadas que “transformam” o tamanho das camadas anteriores são lineares usando a função `nn.Linear` do PyTorch e usam como função de ativação ReLU. E a diferença interessante que temos nesse caso é nossa última camada que tem como ativação a função tangente hiperbólica que foi escolhida especialmente porque vai restringir a saída da rede entre o intervalo $[-1,1]$ que e isso é comumente usado para gerar imagens em GANs e em outras implementações como difusão estável, onde os valores dos pixels devem estar dentro desse intervalo.

Dito isso, vou apenas citar que todas mudanças como incluir dropout e mudar de ReLU para LeakyReLU seguem a mesma lógica do porque fiz essas mudanças para o discriminador, fora que esse é o padrão adotado pela maioria das pessoas que fizeram qualquer tipo de trabalho como esse envolvendo imagens e GANs.

4.2.2. Aplicando convolução na rede generativa

Seguindo as mesmas lógicas do porque adotei convolução na rede discriminativa, fiz igual para a rede generativa, elas ficaram inclusive com a mesma arquitetura, exceto a ativação final que uma é sigmoid no intuito de gerar uma probabilidade, e a outra é tangente hiperbólica porque tem o foco de gerar valor do pixel.

É importante repetir que a convolução melhorou drasticamente a redução de ruído e a nossa imagem agora só sofre do problema de rostos levemente deformados, talvez por estar rodando poucas épocas e termos “apenas” 200 mil imagens.

4.3 Escolha dos parâmetros e otimizadores

No nosso artigo, fizemos escolhas cuidadosas em relação aos parâmetros e otimizadores utilizados na nossa GAN. Primeiramente, definimos a taxa de aprendizado como 0.0001, o que influencia o tamanho do passo que os otimizadores darão ao ajustar os pesos da rede. Optamos por um valor baixo para garantir uma convergência mais suave durante o treinamento, passos muito grandes poderiam gerar maiores deformidades nas imagens.

Quanto à função de perda (`loss_function`), utilizamos a Binary Cross Entropy Loss (BCELoss), uma função de perda comumente empregada em tarefas de classificação binária, mas já tínhamos visto ela na nossa função que as redes jogam minimax.

Para otimizar os pesos do discriminador e do gerador, decidimos empregar o algoritmo Adam como otimizador. Utilizamos o Adam para ambos os componentes da GAN, configurando o learning rate como `lr`. O Adam é um otimizador eficiente que adapta a taxa de aprendizado individualmente para cada parâmetro, levando em consideração seus gradientes anteriores. Essa escolha visa melhorar a estabilidade e a velocidade de convergência do treinamento.

Uma das principais vantagens do Adam é que ele combina as melhores características de outros dois otimizadores amplamente utilizados: o algoritmo de descida do gradiente estocástico (SGD) e o otimizador com momento. E como sabemos, o SGD funciona bem com grandes conjuntos de dados, e o objetivo desse trabalho era ser algo introdutório para um projeto maior e que iria envolver diversas imagens no banco de dados.

5. Treinamento

Já explicamos anteriormente como as duas redes adversárias funcionam e como elas se comunicam para convergir a uma distribuição realista da rede generativa. Mas como isso é feito no loop de treinamento pode não ter ficado claro ainda, qualquer leitor deste artigo pode ir nas referências e ver nosso código final, e todos os loops de treinamento possuem um mesmo formato e vamos explicar nessa sessão.

No início do nosso código, após pré processarmos os dados fazendo transformações de redimensionamento, normalização e outras coisas, nós dividimos nosso dataset [7] em batches, que são lotes que contém partes iguais do dataset amostradas de forma aleatória e cada época de treinamento usa todos os batches para o treinamento mas treinam o nosso modelo com cada batch de forma separada. A ideia de separar o dataset em batches vem do mesmo sentido que vimos em aula com o K-Fold mas é mais profundo que isso, essa técnica que usamos é chamada de treinamento em mini-lotes (mini-batch training), essa técnica é frequentemente usada em problemas de aprendizado de máquina e no nosso caso em questão nos dá várias melhorias como eficiência computacional, treinar a GAN em mini lotes permite que o processo de atualização dos pesos da rede seja mais eficiente computacionalmente. Em vez de calcular os gradientes e atualizar os pesos para todas as 200 mil imagens de uma vez, o modelo atualiza os pesos com base em um subconjunto menor de imagens, o que requer menos memória e tempo de computação. Outra coisa boa é a regularização, ao utilizar mini lotes, ocorre uma forma de regularização chamada "regularização estocástica". Isso ocorre porque, a cada iteração, o modelo é exposto a diferentes subconjuntos de dados, introduzindo uma fonte de aleatoriedade e ajudando a evitar o overfitting.

Então nosso algoritmo faz iterações para cada época, e cada época inteira em cima de cada lote, que normalmente têm tamanhos de potência de dois, por motivos de eficiência computacional, uso de memória e paralelização, pesquisamos e descobrimos que vários frameworks como o PyTorch que usamos, possui algoritmos de aprendizado otimizados

para batchs com tamanho potências de dois. E dentro de cada uma dessas iterações de batch, fazemos basicamente o algoritmo que já comentamos acima do jogo do minimax.

Nós em geral criamos tensores com amostras reais vindas do batch e geramos amostras novas usando a amostra do espaço latente aleatório, juntamos essas duas informações em uma só e damos como entrada do nosso discriminador, ele vai gerar um output de o que ele acha ser real e o que ele acha ser falso, lembrando que nessa informação que estamos dando como entrada para ele, tem ali imagens verdadeiras e imagens falsas geradas pelo nossa rede generativa. Usamos esse output da rede discriminativa e o que deveria ser correto para dar como saída, e jogamos como entrada na nossa função de perda, que é a entropia cruzada que estamos usando como função V do jogo minimax e usamos backpropagation, para preparar o modelo para dar o próximo passo, maximizando os parâmetros da rede discriminativa. Uma coisa importante que melhorou nossos resultados em geral, foi alterar o loop de treinamento, começamos a fazer o discriminador criticar as imagens da rede generativa apenas de 5 em 5 épocas, esse método reduz ruído e atrasa a evolução do nosso discriminador, já que é muito mais fácil criticar a imagem do que fazer ela. Depois disso, fazemos justamente o passo de minimizar a rede generativa, fazendo $G(z)$ e depois $D(G(z))$ e jogando o output de novo na função de perda e fazendo backpropagation, dando fim ao loop e repetindo isso para cada lote e depois repetindo tudo isso para cada época.

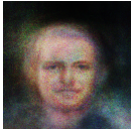



6. Experimentos realizados

Como se trata de imagens, a definição de desempenho do modelo pode ser medida observando as curvas da função de perda, mas analisamos os resultados dos experimentos no olho, basicamente rodamos o código, olhávamos a imagem e se não estivesse satisfatória, buscávamos algo que pudesse melhorar o código.

Começamos rodando os nossos experimentos com 15 mil imagens apenas, para conseguir resultados com uma frequência maior. Isso implicou em resultados ruins, aparentemente para essa tarefa, 15 mil imagens é muito pouco, mas serviu para conseguirmos testar várias vezes seguidas os outputs.



É importante citar que usamos duas máquinas diferentes para rodar o mesmo código, alguns experimentos foram realizados em um MacBook Air 2022 que não possui GPU, mas possui o chip M2 com o Metal Performance Shader, que processa imagens basicamente 40% mais devagar do que seria em uma GPU, mas ainda assim consideravelmente melhor que rodar na CPU. E outros experimentos foram rodados em um computador com uma placa de vídeo do modelo ASUS GEFORCE GTX 1060 DUAL OC 6GB, que é uma GPU que processava os experimentos muito mais rapidamente que o MacBook. Abaixo segue uma tabela com as imagens geradas dos experimentos e com informações como número de imagens que usamos para o treino e duração do tempo de treino.

6.1 Experimentos com 50 Épocas e sem Convolução

Nº Imagens	Duração (T1)	Resultado (T1)	Duração (T2)	Resultado (T2)
15000	42m 19.6s		41m 40.1s	
200599	237m 54.9s		224m 41.7s	


Para T1, o teste usando uma rede geradora com função ReLU de ativação entre camadas e T2, o teste usando uma rede geradora com função LeakyReLU de ativação entre camadas.

6.2 Experimentos com 50 Épocas e Convolução

Nº Imagens	Duração	Resultado
15000	9m 14.0s	
200599	315m 55.2s	

Nos testes com convolução, utilizamos a uma rede geradora com função ReLU de ativação entre camadas, conforme sugerido no guia [9].

6.3 Experimento Final com 400 Épocas e Convolução

Nº Imagens	Duração	Resultado
200599	2357m 26.5s	

7. Discussão dos resultados

Nossa metodologia basicamente foi aplicar mudanças no nosso modelo inicial ingênuo até ele evoluir para um mais complexo, não em quesito quantidades de linha de código nem nada do tipo, mas sim por ser mais robusto, por conter melhores escolhas de função de ativação, arquitetura das camadas mais bem desenvolvidas, normalizações de valores e parâmetros. E isso foi feito usando sempre a comparação com mesmo número de épocas, que foi 50 e com 15 mil imagens, esse foi um número escolhido porque foi o valor em que o nosso modelo mais ingênuo convergiu para algo que parecia um rosto, e como ele conseguiu convergir para imagem de um rosto, na teoria todos os outros conseguiriam também. Não provamos em nenhum momento ao longo desse artigo que essa ordem de qualidade de ReLU sem convolução < LeakyReLU sem convolução < ReLU com convolução se estende conforme aumentamos nosso número de épocas e imagens na base de dados, mas isso está implícito na nossa intuição que formamos por conhecer como funciona aprendizado em máquina em si, e nosso conhecimento de conceitos estatísticos como Lei dos Grandes Números, seria fácil provar isso, bastava incluímos experimentos iguais desses 3 tipos de combinações por quantidade de imagens na base de dados e por número de épocas, mas considerando que a média desses treinamentos foi 1 hora, não teríamos tempo para rodar esses outros experimentos, ficamos sem tempo por ter começado o trabalho uma semana antes do prazo e estarmos também ocupados com outras disciplinas. Mas então concluímos que incluir convolução, por analisar melhor características locais específicas de cada imagem e outros benefícios, usar LeakyReLU para evitar o problema da morte exagerada de neurônios, adotar Dropout para evitar overfitting e incluir a técnica de ciclar de quantos em quantas épocas o discriminador vai criticar a imagem gerada são as melhores técnicas que conseguimos aprender para melhorar o desempenho da nossa GAN.

8. References

- [1] Goodfellow, I. et al., 2014. Generative adversarial nets. In Advances in neural information processing systems. pp. 2672–2680
- [2] Larabel, Michael (February 10, 2019). "NVIDIA Opens Up The Code To StyleGAN - Create Your Own AI Family Portraits". Phoronix.com. Retrieved October 3, 2019.
- [3] Elad Richardson, Yuval Alaluf, Or Patashnik, Yotam Nitzan, Yaniv Azar, Stav Shapiro, Daniel Cohen-Or, Penta-AI, Tel-Aviv University (April 21, 2021). Encoding in Style: a StyleGAN Encoder for Image-to-Image Translation.
- [4] Tero Karras (NVIDIA), Timo Aila (NVIDIA), Samuli Laine (NVIDIA), Jaakko Lehtinen (NVIDIA and Aalto University), 2018. Progressive Growing Of GANs for Improved Quality, Stability, and Variation.
- [5] Jonathan Hui (June 22, 2018). "GAN — Some cool applications of GAN", Medium, <https://jonathan-hui.medium.com/gan-some-cool-applications-of-gans-4c9ecca35900>
- [6] Hunter Heidenreich (August 18, 2018), "What is a Generative Adversarial Network?", <http://hunterheidenreich.com/blog/what-is-a-gan/>
- [7] Jessica Li (June 01, 2018), "CelebFaces Attributes (CelebA) Dataset", Kaggle, <https://www.kaggle.com/datasets/jessicali9530/celeba-dataset>
- [8] Renato Candido (July 27, 2020), "Generative Adversarial Networks: Build Your First Models", Real Python, <https://realpython.com/generative-adversarial-networks/>
- [9] Nathan Inkawhich (August 29, 2018), "DCGAN Tutorial", PyTorch, https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html